Chapter 22
ADS and MDAC, OLE DB, ADO, and Visual Basic

Note: This chapter accompanies the book Advantage Database Server: A Developer's Guide, 2nd Edition, by Cary Jensen and Loy Anderson (2010, ISBN: 1453769978). For information on this book and on purchasing this book in various formats (print, e-book, etc), visit: http://www.JensenDataSystems.com/ADSBook10

Note: Visual Basic, though still used widely, has not seen an update since 1998 (from a development standpoint, Visual Basic for .NET is a separate language, one more closely aligned to the .NET framework). As a result, this chapter appears as it did in previous editions of this book, with only minor modifications.

We want to thank Chris Franz and Lance Schmidt from the Advantage team at Sybase. Chris provided invaluable help with all of the Visual Basic examples used here and in previous editions of this book. Lance also gave us technical assistance on Visual Basic.

ADO (ActiveX data objects) provides the API (application programming interface) part of MDAC (Microsoft Data Access Components). MDAC is the implementation of Microsoft's Universal Data Access strategy, which is designed to provide a high-performance, language-independent data access layer in the Windows operating system.

This chapter provides you with an introduction to MDAC, and ADO in particular. Because ADO is language neutral, the examples presented in this chapter could have been produced using any one of a wide variety of programming languages. Due to its popularity, and heavy reliance on ADO for data access, we have chosen Visual Basic 6.

This chapter begins with a high-level introduction to ADO and OLE DB. It then continues by showing you how to access your Advantage data using ADO.

ADS and MDAC, OLE DB, ADO, and Visual Basic

There are three layers to Universal Data Access, Microsoft's ambitious initiative to build a data access layer into the Windows operating system. At the lowest layer are data-providing applications and services. In most cases, such as with Advantage, these are

client/server database servers. But in practice, they can be almost any type of application imaginable. Nonetheless, the one characteristic that all data providers and services share is that they provide access to information.

Above this lowest layer is OLE DB, which consists of a series of COM (component object model) interfaces. OLE DB provides a layer of abstraction between the data providers and the data consumers, which are your client applications in the traditional client/server architecture.

COM interfaces are simply API templates, which alone are useless unless they are implemented by objects. This is where OLE DB providers come in. OLE DB providers are the objects that implement the OLE DB interfaces, and which perform the physical communication with the data-providing applications and services. In the MDAC scheme of things, both OLE DB and OLE DB providers reside in this middle layer.

While it is conceivable for an application developer to program directly to the OLE DB API, doing so would be both time-consuming and complicated. This is because OLE DB, being a low-level interface, was not designed as a developer API. And this is where the third and highest layer comes in, ADO.

ADO consists of a collection of ActiveX data objects that encapsulate calls to OLE DB. By comparison to OLE DB, the ADO API is simple, providing client application developers with convenient access to the data supplied by the data providers and services.

MDAC is normally available on all Windows machines (only Windows 95, which has not been supported by Microsoft for some time, did not come with MDAC already installed). Furthermore, the latest version of MDAC can be freely downloaded from Microsoft's Web site at http://msdn.microsoft.com/data/mdac/downloads/. The licensing for MDAC specifically permits you to distribute it with your applications, but that is rarely necessary (unless you are installing your applications on obsolete machines).

MDAC consists of all of the ActiveX data objects, as well as a collection of OLE DB providers. The standard providers include the Microsoft Jet 4.0 OLE DB Provider, the Microsoft OLE DB Provider for SQL Server, the Microsoft OLE DB Provider for Oracle, and the Microsoft OLE DB Provider for ODBC, just to name a few.

There are two critical characteristics of Microsoft's Universal Data Access that make this an appealing data access solution. First, regardless of which OLE DB provider you want to use, you access it through the one set of ActiveX data objects. They provide the common API.

The second is that you are not limited to using just the standard OLE DB providers that ship with MDAC. Any COM objects that correctly implement the necessary OLE DB interfaces can be installed on a Windows computer and executed through ADO. The Advantage OLE DB Provider is an example of such an implementation. After installing the Advantage OLE DB Provider, which automatically registers this provider with COM, you can use ADO to access your Advantage data.

There is one final issue that deserves mention before turning our attention to using Advantage through ADO: client-side cursors versus server-side cursors. In ADO, when

you execute a command that returns a result set, you specify where the result set will reside using the CursorLocation property of a Connection or Recordset object. If you set CursorLocation to adUseClient, all records from the result set are downloaded from the server and stored in-memory on the client workstation. By comparison, if you set CursorLocation to adUseServer, the Advantage OLE DB Provider manages the access of data using ADS, retrieving to the client only those records required by your application. The default value of the CursorLocation property of Connection and Recordset objects is adUseServer.

When you load your data into a client-side cursor, operations such as sorting, finding, filtering, and the like are performed by the ADO client cursor engine, and do not involve ADS until you are ready to write changes back to the server. By comparison, when you use server-side cursors, you are leveraging the power and performance of ADS in operations that involve filters, indexes, seeks, and navigation. As a result, the examples discussed in this chapter make use of server-side cursors. If you are interested in learning more about client-side cursors and their features, refer to a book on ADO.

Note: Because client-side cursors require that all records be downloaded from the server into memory on your workstation before you can work with your data, you may experience significant delays when loading large result sets using client-side cursors. In most cases, you will achieve excellent performance with ADS and server-side cursors, making them a better solution—particularly when your result sets are large.

The remainder of this chapter shows you how to access Advantage using Visual Basic 6. These discussions are divided into three parts. The first part describes common tasks, such as connecting to Advantage and accessing data. The second part shows you how to perform simple navigation using ADO. The third and final part demonstrates several basic administrative tasks, such as creating tables and granting rights to them.

Code Download: The VB project VB_ADS.vbp can be found on this book's code download (see Appendix A).

All of the examples presented here can be found in the VB_ADO.vbp Visual Basic project. Figure 22-1 shows the main form of this project in Visual Studio.

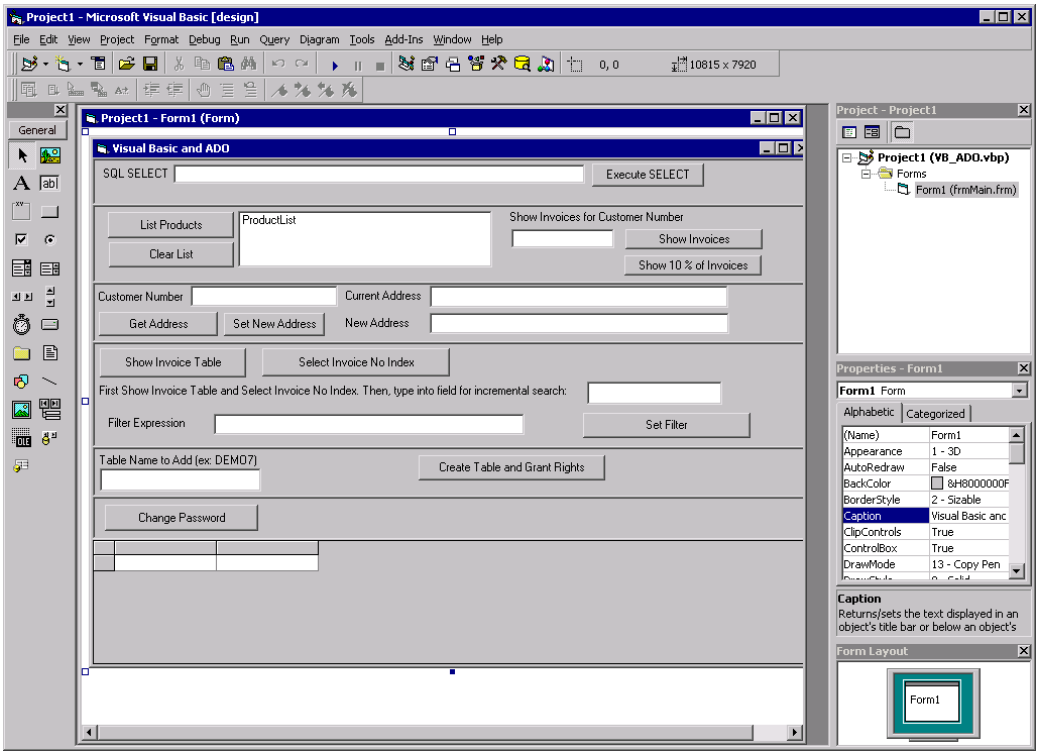

Figure 22-1: The VB_ADO project in Visual Studio

Note: The VB_ADO project with the code download for this book was compiled with MDAC 2.8. If you are not using MDAC 2.8, you will get an error when you first try to run this project. If this happens, select Project | References. Use the displayed dialog box to uncheck the Microsoft ActiveX Data Objects Library 2.8 (if necessary). Then, scroll to find the version of the Microsoft ActiveX Data Objects Library that you want to use (this must be version 2.1 or later), and add a checkmark to it. Click OK when you are done.

As is the case with all data access mechanisms described in Part III, the following discussion of Advantage programming with Visual Basic touches on just a few of the available techniques. For a more comprehensive discussion of ADO programming, you may want to pick up a book on ADO programming.

If you are creating a new project that uses ADO, you must add a reference to the Microsoft Data Access Objects library before you can use the Advantage OLE DB Provider with Visual Studio. To do this, use the following steps:

1. From Visual Studio, select Projects | References. Visual Studio displays the References dialog box as shown in Figure 22-2.
2. Scroll the Available References list until you see Microsoft ActiveX Data Objects Library. Place a checkmark next to the version with the highest major and minor version, as shown in Figure 22-2.
3. Click OK to close the References dialog box.

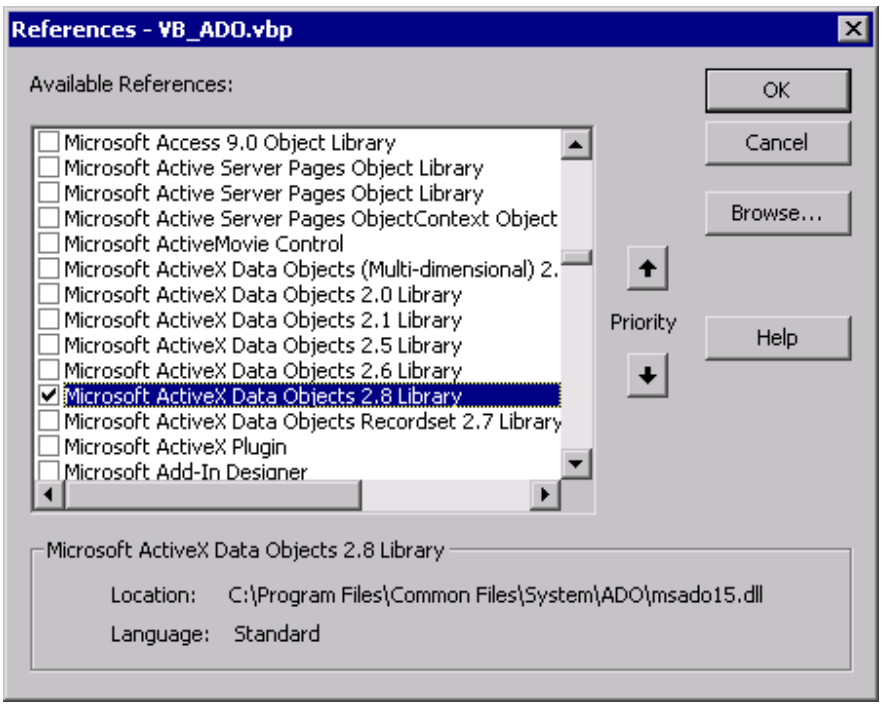

Figure 22-2: Adding a reference to the Microsoft Data Access Objects library

Performing Basic Tasks with Advantage and ADO

This section describes some of the more common tasks that you can perform with ADO. These include connecting to a data dictionary, executing a query, using a parameterized query, retrieving and editing data, and executing a stored procedure.

Connecting to Data

You connect to a data dictionary or a directory in which free tables are located using a Connection object found in the ADODB namespace. At a minimum, you must provide the Connection object with sufficient information to locate your data and configure how the data should be accessed. This can be done either with the Parameters collection property or the ConnectionString property. Both of these properties accept name/value pairs using the parameters listed in Table 22-1. If you use the ConnectionString property, and use more than one name/value pair, separate them with semicolons.

Parameter	Description
CharType	Set to the character set type for DBF files. Valid values are ADS_ANSI and ADS_OEM. The default value is ADS_ANSI.
Compression	Set to ALWAYS, INTERNET, NEVER, or empty. If left empty (the default), the ADS.INI file will control the compression setting. This parameter is not used by ALS.

CommType	The communication protocol to use to connect to ADS. Under Windows and Linux, the default is UDP_IP. For Novel Netware, the default is IPX. To use TCP/IP, set CommType to TCP_IP.
Data Source	The path to your free tables or data dictionary. If you are using a data dictionary, you must include the data dictionary filename in this path. It is recommended that this path be a UNC path. Data Source is a required parameter.
DbfsUseNulls	Set to TRUE to return empty fields from DBF files as NULL values. If set to FALSE, empty fields are returned as empty data values. The default is FALSE.
EncryptionPassword	Set to an optional password to use for accessing encrypted free tables. If using less than a 20-letter password, a semicolon should be included directly after the password so the Advantage OLE DB Provider knows when the password ends. This parameter is ignored for data dictionary connections.
FilterOptions	Set to IGNORE_WHEN_COUNTING or RESPECT_WHEN_COUNTING. When set to IGNORE_WHEN_COUNTING, the RecordCount property of a Recordset may not accurately reflect the number of records in a result set. Set to RESPECT_WHEN_COUNTING for accurate record counts. Requesting accurate record counts can reduce performance significantly, and should be used only if accurate counts are needed. The default is IGNORE_WHEN_COUNTING.
IncrementUsercount	Set to TRUE to increment the user count when the connection is made. Set to FALSE to make a connection without incrementing the user count. The default is FALSE.
Initial Catalog	Optional name of a data dictionary if the data dictionary is not specified in the Data Source parameter.
LockMode	Set to ADS_PROPRIETARY_LOCKING or ADS_COMPATIBLE_LOCKING to define the locking mechanism used for DBF tables. Use ADS_COMPATIBLE_LOCKING when your connection must share data with non-ADS applications. The default is ADS_PROPRIETARY_LOCKING.
Password	When connecting to a data dictionary that requires logins, set to the user's password.
Provider	This required parameter must be set to either Advantage OLE DB Provider or Advantage.OLEDB.1.
SecurityMode	Set to ADS_CHECKRIGHTS to observe the user's network access rights before opening files, or ADS_IGNORERIGHTS to access files regardless of the user's network rights. The default is ADS CHECKRIGHTS. This property applies only

	to free table connections.
ServerType	Set to the type of ADS server you want to connect to. Use ADS_LOCAL_SERVER, ADS_REMOTE_SERVER, or ADS_INTERNET_SERVER. To attempt to connect to two or more types, separate the server types using a vertical bar (|). This is demonstrated in the ConnectionString shown later in this chapter.
ShowDeleted	Set to TRUE to include deleted records in DBF files. Set to FALSE to suppress deleted records. The default is FALSE.
StoredProcedure Connection	Set to TRUE if connecting from within a stored procedure. When set to TRUE, the connection does not increment the user count. The default is FALSE.
TableType	Set to ADS_ADT, ADS_CDX, ADS_VFP, or ADS_NTX to define the default table type. The default is ADS_ADT. This parameter is ignored for data dictionary connections.
TrimTrailingSpaces	Set to TRUE to trim trailing spaces from character fields. Set to FALSE to preserve trailing spaces. The default is FALSE.
User ID	If connecting to a data dictionary that requires logins, set to the user's user name.

Table 22-1: ADO Connection String Parameters

For any of the optional connection string parameters that you fail to provide, the Advantage OLE DB Provider will automatically insert the default parameters. Furthermore, instead of supplying a connection string with your connection information, you can set the connection string to the following pattern:

```
FILE NAME=path\filename.udl
```

where *path* is the physical or UNC path to a directory in which a file with the .udl file extension resides, and *filename* is the name of a UDL (universal data link) file. UDL files are INI-style files that contain ADO connection information. Under the most recent versions of Windows, UDL files are stored in:

```
C:\Program Files\Common Files\System\Ole DB\Data Links
```

You do not even need to know how a UDL file is structured to create one. Simply create a new empty file in the preceding directory using the UDL file extension. Then, using the Windows Explorer, right-click this filename and select Properties. Use the displayed properties dialog box, shown in Figure 22-3, to configure the connection information. At runtime, when ADO processes the connection string containing FILE NAME=*path**filename*.udl, it will expand the connection string, populating it with the definitions located in the UDL file.

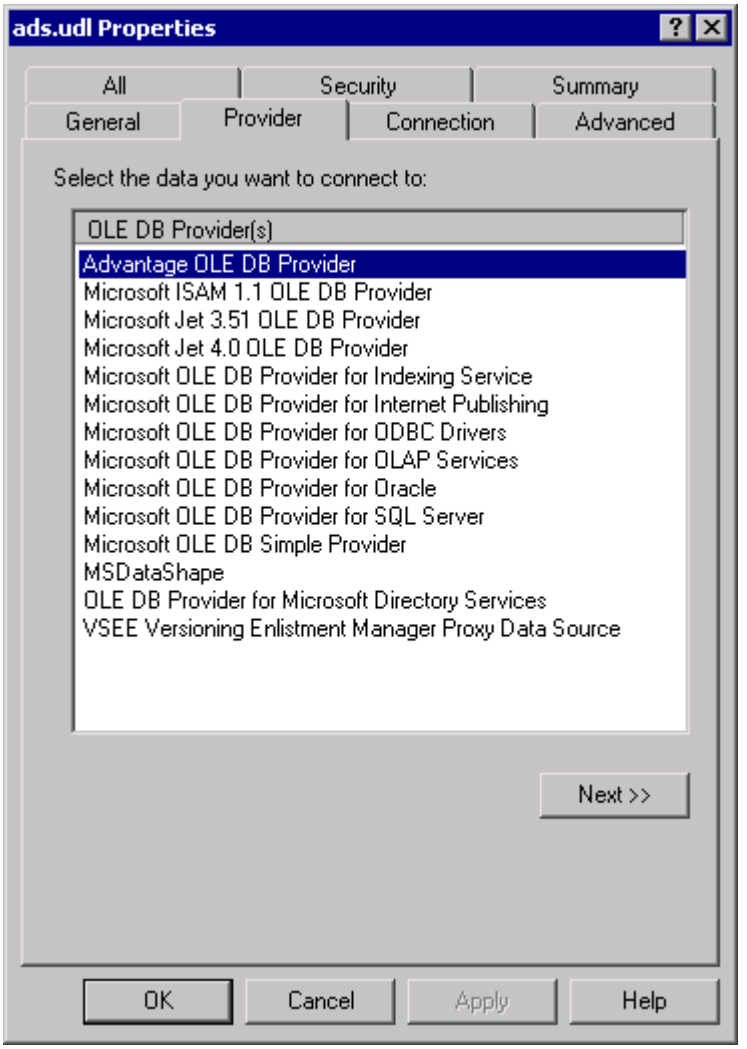

Figure 22-3: Setting the connection string properties using a UDL file

Because the Connection object that is used by this project must be used by a number of subprocedures on the form, the AdsConnection variable and several other variables that must be repeatedly referenced are declared as form-level variables. The following code segment shows this declaration. Note also in the following code segment that the data source location of the data dictionary is declared as a constant.

```
'Require explicit variable declarations
Option Explicit
Dim AdsConnection As ADODB.Connection
Dim AdsCommand As ADODB.Command
Dim AdsRecordset As ADODB.Recordset
Dim AdsParamQueryCommand As ADODB.Command
Dim AdsParamQueryRecordset As ADODB.Recordset
Dim AdsParameter As ADODB.Parameter
Dim AdminConnection As ADODB.Connection
```

```
Dim AdminCommand As ADODB.Command
Const DataPath = "\\server\share\ADSBook\DemoDictionary.add"
```

This OLE DB Connection, named AdsConnection, is created, configured, and opened from the Load event of the form, along with several other Command, Recordset, and Connection objects. The relevant portion of this subprocedure is shown in the following code:

```
Private Sub Form_Load()
  On Error GoTo ErrorHandler
  Set AdsConnection = New ADODB.Connection
  Set AdsCommand = New ADODB.Command
  Set AdsRecordset = New ADODB.Recordset
  Set AdminConnection = New ADODB.Connection
  Set AdminCommand = New ADODB.Command
  'Setup the connection
  AdsConnection.ConnectionString = _
    "Provider=Advantage OLE DB Provider;" + _
    "Data Source=" + DataPath + ";user ID=adsuser;" + _
    "password=password;" + _
    "ServerType=ADS_LOCAL_SERVER | ADS_REMOTE_SERVER;" + _
    "FilterOptions=RESPECT_WHEN_COUNTING;" + _
       TrimTrailingSpaces=True"
  AdsConnection.Open

  Set AdsCommand.ActiveConnection = AdsConnection
  'Additional code not shown follows
  Exit Sub
ErrorHandler:
   MsgBox "Error: " & Err.Number & vbCrLf & _
     "Description: " & Err.Description
  Exit Sub
End Sub
```

As you inspect this code, you will notice that all errors are handled by displaying the error code and message of the error. This type of error handler is present in every subprocedure in this Visual Basic project. In order to reduce redundancy in this chapter, the error handling block, as well as the subprocedure declaration, is omitted from the remaining subprocedure listings in this chapter.

Note: If you have difficulty connecting, it might be because you have other client applications, such as the Advantage Data Architect, connected using a local connection. Ensure that all clients connected to the database use the same type of connection.

Executing a Query

You execute a query that returns a result set by calling the Open procedure of a Recordset. This procedure has five optional parameters. The first is the command you want to execute. This can be either a Command object, the name of a table or stored procedure, or (as in the case in the following code segment) the actual text of the query. The second parameter is the connection over which the query will be executed.

The third parameter identifies the type of cursor that you want returned, and the fourth specifies the type of record locking you want. The fifth and final parameter identifies what kind of command you pass in the first parameter. If you pass a table name in the first parameter, you can pass the value adCmdTable in this fifth parameter, and a SELECT * FROM query will be generated. If you pass the name of a stored procedure that takes no input parameters in the first parameter, an EXECUTE PROCEDURE statement is generated if adCmdStoredProc is given as the fifth parameter.

The following code demonstrates the execution of a query entered by the user into the TextBox named SELECTText. This subprocedure is associated with the Execute SELECT button shown in Figure 22-1:

```
If AdsRecordset.State = adStateOpen Then
    AdsRecordset.Close
End If

AdsRecordset.Open SELECTText.Text, AdsConnection, _
  adOpenDynamic, adLockPessimistic, adCmdText
Set DataGrid1.DataSource = AdsRecordset
```

This code begins by verifying that the Recordset is not currently open, by checking its State property. Next, the query is executed and the returned records are assigned to the Recordset. Finally, the Recordset is assigned to the DataSource property of the DataGrid. The effects of executing this code are shown in Figure 22-4.

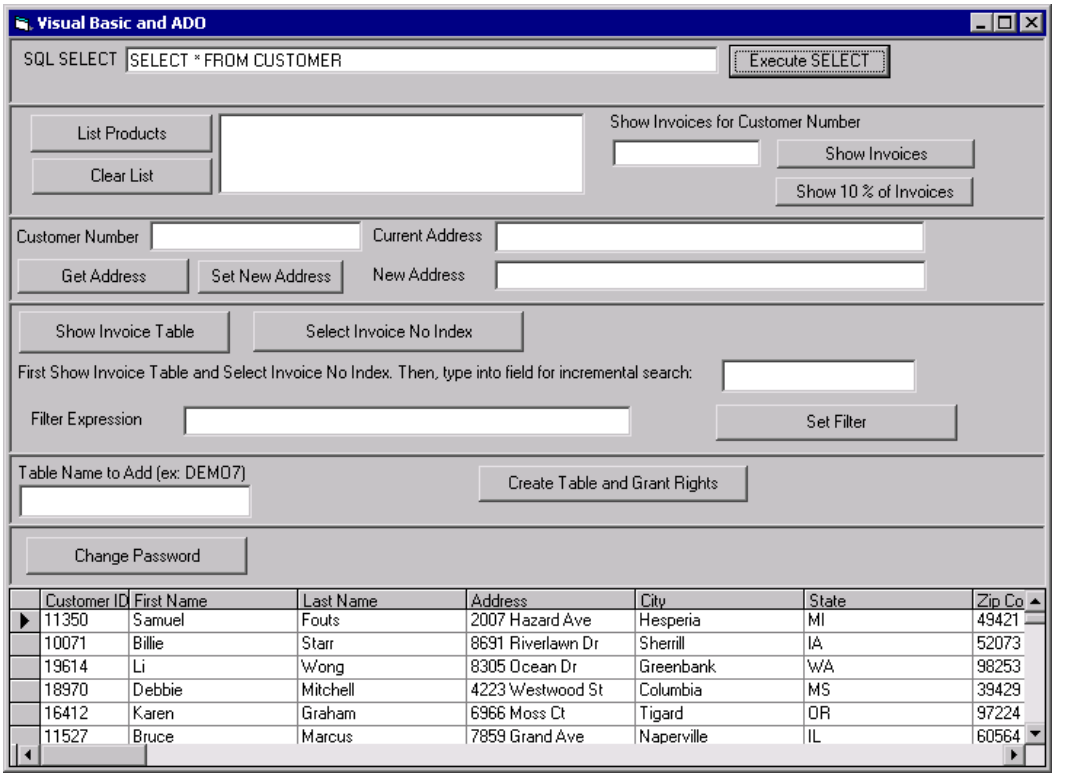

Figure 22-4: Records returned from a query are displayed in a data grid

If you need to execute a query that does not return a Recordset, use a Command object. The use of a Command object to execute a query that does not return a Recordset is demonstrated later in this chapter.

Using a Parameterized Query

Parameterized queries are defined using a Command object. This command object can then be executed directly, so long as the query does not return a result set. If the parameterized query returns one or more records, you can execute it using the Open method of a Recordset, just as you can with a query that takes no parameters.

Before you can invoke a parameterized query, you must create one Parameter object for each of the query's parameters, and then associate each Parameter with the Command holding the parameterized query.

The definition of a parameterized query, including the creation and configuration of a parameter, is shown in the following code segment. This code segment is part of the Load event for the form object, and was omitted from the code listing shown earlier (in the section "Connecting to Data"):

```
'Set up the parameterized query that will be reused
Set AdsParamQueryCommand = New ADODB.Command
Set AdsParamQueryRecordset = New ADODB.Recordset
Set AdsParamQueryCommand.ActiveConnection = AdsConnection
AdsParamQueryCommand.CommandText = _
  "SELECT * FROM INVOICE WHERE [Customer ID] = ?"
Set AdsParameter = AdsParamQueryCommand.CreateParameter
AdsParamQueryCommand.Prepared = True
AdsParamQueryCommand.NamedParameters = False
AdsParameter.Type = adInteger
AdsParamQueryCommand.Parameters.Append AdsParameter
```

Once a Parameter has been created, configured, and associated with the Command holding the parameterized query statement, there is only one more step necessary before the query can be executed. You must bind data to each parameter. This is shown in the following click event of the DoParamQuery button (the button labeled Show Invoices in Figure 22-1):

```
If IsNumeric(ParamText.Text) = False Then
  MsgBox "Invalid customer number"
  Exit Sub
End If
If AdsParamQueryRecordset.State = adStateOpen Then
  AdsParamQueryRecordset.Close
End If
AdsParameter.Value = CInt(ParamText.Text)
AdsParamQueryRecordset.Open AdsParamQueryCommand, , _
    adOpenDynamic, adLockPessimistic, adCmdText
If AdsParamQueryRecordset.BOF And _
  AdsParamQueryRecordset.EOF Then
  MsgBox "No invoices for customer ID"
```

```
  Set DataGrid1.DataSource = Nothing
Else
  Set DataGrid1.DataSource = AdsParamQueryRecordset
End If
```

As you can see from this code, after verifying that a numeric value has been entered into the Customer ID field, the entered data is assigned to the Value property of the parameter and the query is executed.

This example is actually a classic example of how parameterized queries are used. Specifically, the query text is defined only once, but can be executed repeatedly. And by changing only the value of the parameter, a different result set can be returned upon each execution.

Note: The Advantage OLE DB Provider only supports positional parameters—named parameters are not supported. As a result, if you have more than one parameter, it is important to keep track of the position in which each parameter appears.

Reading and Writing Data

You read data from fields of a Recordset by using the Recordset's Fields property, which is a collection property. The Fields property takes a single parameter that identifies which field's value you want to read. This value can either be an integer identifying the ordinal position of the field in the table's structure (this value is zero-based) or it can be a string identifying the field's name. The Value property of the identified field holds the field's data.

Reading data from a Recordset is demonstrated in the following Click subprocedure associated with the Get Address button shown in Figure 22-1:

```
Dim AdsGetCustCommand As ADODB.Command
Dim AdsGetCustRecordset As ADODB.Recordset
Dim AdsGetCustParameter As ADODB.Parameter

If CustNoText.Text = "" Or Not IsNumeric(CustNoText.Text) Then
  MsgBox "Please supply a valid customer ID number"
  Exit Sub
End If
Set AdsGetCustCommand = New ADODB.Command
Set AdsGetCustRecordset = New ADODB.Recordset
Set AdsGetCustCommand.ActiveConnection = AdsConnection
AdsGetCustCommand.CommandText = _
  "SELECT * FROM CUSTOMER WHERE [Customer ID] = ?"
Set AdsGetCustParameter = AdsGetCustCommand.CreateParameter
AdsGetCustCommand.Prepared = True
AdsGetCustCommand.NamedParameters = False
AdsGetCustParameter.Type = adInteger
AdsGetCustCommand.Parameters.Append AdsGetCustParameter
AdsGetCustParameter.Value = CInt(CustNoText.Text)
AdsGetCustRecordset.Open AdsGetCustCommand, , _
  adOpenDynamic, adLockPessimistic, adCmdText
```

```
If AdsGetCustRecordset.BOF And _
  AdsGetCustRecordset.EOF Then
  MsgBox "No records for customer ID"
  Set DataGrid1.DataSource = Nothing
Else
  Set DataGrid1.DataSource = AdsGetCustRecordset
  OldAddressText.Text = _
    AdsGetCustRecordset.Fields("Address").Value
End If
```

So long as your Recordset contains a dynamic (live) cursor, you can make changes to a Recordset by assigning data to one or more of the Recordset's Fields Value properties, where you identify the field you are writing to by using the same technique that you use to read from a field. After changing one or more fields, you call the Update method of the Recordset to write those changes to Advantage. Alternatively, you can call the Recordset's Update, passing to it either a field name/value pair or an array of field name/value pairs. This second approach writes one or more updated fields to ADS in a single command.

The following code demonstrates one way to update a Recordset. This code can be found for the click procedure associated with the button labeled Set New Address shown in Figure 22-1:

```
If CustNoText.Text = "" Or Not IsNumeric(CustNoText.Text) Then
  MsgBox "Please supply a valid customer ID number"
  Exit Sub
End If
If InStr(1, TableNameText.Text, ";", vbTextCompare) <> 0 Then
  MsgBox "Customer ID may not contain a semicolon"
  Exit Sub
End If
Dim AdsGetCustRecordset As ADODB.Recordset
Set AdsGetCustRecordset = New ADODB.Recordset
AdsGetCustRecordset.Open "SELECT Address FROM CUSTOMER " + _
  "WHERE [Customer ID] = " + CustNoText.Text, _
  AdsConnection, adOpenDynamic, adLockPessimistic, adCmdText
If AdsGetCustRecordset.BOF And _
  AdsGetCustRecordset.EOF Then
  MsgBox "Customer ID not found"
Else
  AdsGetCustRecordset.Fields("Address").Value = _
    NewAddressText.Text
  AdsGetCustRecordset.Update
End If
MsgBox "Address for customer " + CustNoText.Text + " " + _
  "has been updated"
Exit Sub
```

Of course, a SQL UPDATE query can also be used to achieve a similar result.

Calling a Stored Procedure

Calling a stored procedure is no different than executing any other query. If your stored procedure does not require input parameters, you can define the query text using a

Command object or by passing the call to EXECUTE PROCEDURE in the first parameter of a Recordset Open invocation. Alternatively, you simply pass the name of the stored procedure object in the CommandText, and set the CommandType property (of a Command object) or CommandType parameter (of the Resultset's Open method) to adCmdStoredProc (this second technique is demonstrated in the following example). Typically, you use a Command object—executing it directly—when the stored procedure does not return records, and use the Open method of a Recordset when your stored procedure returns one or more records.

Invoking a stored procedure that takes one input parameter is demonstrated by the following code associated with the click event for the Show 10% of Invoices button (shown in Figure 22-1). The stored procedure referenced in this code is the SQL stored procedure created in Chapter 7. If you did not create this stored procedure, but created one of the other AEPs described in that chapter, substitute the name of the stored procedure object in your data dictionary in the CommandText property of the Command object.

```
Dim AdsSPCommand As ADODB.Command
Dim AdsSPRecordset As ADODB.Recordset
Dim AdsSPParameter As ADODB.Parameter

If ParamText.Text = "" Or Not IsNumeric(ParamText.Text) Then
  MsgBox "Please supply a valid customer ID number"
  Exit Sub
End If
Set AdsSPCommand = New ADODB.Command
Set AdsSPRecordset = New ADODB.Recordset
Set AdsSPCommand.ActiveConnection = AdsConnection
AdsSPCommand.CommandText = "Get10PercentSQL"
Set AdsSPParameter = AdsSPCommand.CreateParameter
AdsSPCommand.Prepared = True
AdsSPCommand.NamedParameters = False
AdsSPParameter.Type = adInteger
AdsSPCommand.Parameters.Append AdsSPParameter
AdsSPParameter.Value = CInt(ParamText.Text)
AdsSPRecordset.Open AdsSPCommand, , adOpenDynamic, _
  adLockPessimistic, adCmdStoredProc
Set DataGrid1.DataSource = AdsSPRecordset
Exit Sub
```

Navigational Actions with Advantage and ADO

ADO supports a number of navigational operations on populated Recordsets, permitting you to leverage Advantage's support for both navigational and set-based SQL data access. This section describes the navigational options made available through server-side cursors. These operations can be performed on any SQL result set that returns a live cursor or any table opened directly.

Unlike most remote relational database servers, Advantage also supports a non-SQL technique for working with a server-side cursor. It involves opening a table directly. When you open a table directly, the Advantage OLE DB Provider obtains a table handle, which permits Advantage to use its high-performance indexes, Advantage Optimized

Filters, and read-ahead record caching to supply data to your client application. Fortunately, with live cursors, Advantage also uses these high-performance features.

You open a table directly by setting the CommandText property of a Command object or the Source parameter of a Recordset's Open method to the table name. You then set the CommandType property of the Command, or the Options parameter of the Recordset's Open method, to adCmdTableDirect. This is shown in the following code segment, which is taken from the ShowInvoiceBtn click event:

```
If AdsRecordset.State = adStateOpen Then
  AdsRecordset.Close
End If
AdsRecordset.Open "INVOICE", AdsConnection, _
  adOpenDynamic, adLockPessimistic, adCmdTableDirect
Set DataGrid1.DataSource = AdsRecordset
```

It's worth noting that there are some similarities, but also some significant differences between using a CommandType of adCmdTable and adCmdTableDirect. Just as you do when you set CommandType (or Options) to adCmdTableDirect, when you use adCmdTable, you assign the name of the table to the CommandText property of a Command object, or pass the table name in the Source parameter of a Recordset's Open method. In response, the Command or Recordset object generates a SELECT * FROM query. Queries like these return a live, server-side cursor (so long as you did not specifically request a client-side cursor), which enables the use of the table's indexes for searching, filtering, and the like.

By comparison, when you set CommandType (or Options) to adCmdTableDirect, the Advantage SQL engine is bypassed altogether, instead opening the table using an OLE DB Rowset object. Opening a table this way enables additional capabilities, including being able to obtain an exclusive lock on the table, which cannot be done through a SQL SELECT statement.

Performing navigational actions on server-side cursors is described in the following sections.

Note: The Advantage OLE DB Provider also supports high-performance bookmarks on server-side cursors. For information on using bookmarks, see the Advantage help.

Setting an Index

From within an ADO application, you select an available index for one of two reasons. Either you want to sort the records in your Recordset based on the index expression, or you want to use the index to enable high-speed searches using a Recordset.

Fortunately, selecting an index that you have defined in a table's index file is straightforward. You assign the name of an index order to the Index property of a Recordset that returns either a dynamic (live) cursor or a table that is opened directly

using the adCmdTableDirect command type. If you are connected to a data dictionary, the index order name can be in any of your table's auto-open indexes.

You return to the natural index order by setting the Index property of the Recordset to an empty string. Setting an index is demonstrated in the following code segment, which is associated with the Select Invoice No Index button shown in Figure 22-1:

```
If AdsRecordset.State <> adStateOpen Then
    MsgBox "Please open Invoice table before setting index"
    Exit Sub
  End If
AdsRecordset.Index = "Invoice No"
```

Note that you cannot set an index if the Recordset is already actively employing a filter. However, you can apply a filter on a Recordset that is using an index.

Finding a Record Based on Data

ADO supports two methods for searching for data in a Recordset. One of these, Find, performs a record-by-record search for data. When used with server-side cursors, these sequential reads are performed by ADS, but nonetheless the sequential nature of this operation means that it is often relatively slow—especially when the result set is large.

The second method, Seek, is only supported in server-side cursors by OLE DB providers that also support indexes and fortunately, the Advantage OLE DB Provider is one of them. Unlike Find, Seek uses Advantage indexes on server-side cursors to quickly locate records. Compared to Find, Seek is typically much faster at finding records with server-side cursors.

Before you can call Seek on a Recordset, you must set an index that you will use to find the record you are looking for. Once the index is set, you call Seek, passing either a single value or an array of values. If you pass a single value, ADS will search the current index for that value in the first field of the index.

You pass an array of values to Seek when you want to search on more than one expression of a multisegment index order. (A multisegment index order is based on two or more fields or expressions.) In this case, Seek searches for the first array element in the first field or expression of the current index, then searches the second array element, if present, in the second field or expression of the current index, and so on.

The second, optional parameter, SeekOption, defines how the Seek is performed. Valid values for this second parameter include adSeekAfter, adSeekAfterEQ, adSeekBefore, adSeekBeforeEQ, adSeekFirstEQ, and adSeekLastEQ. The default value is adSeekFirstEQ.

If a matching record is found, based on the SeekOption, the record associated with the value or values is made the current record. If the value or values are not found, the Recordset will point to the end-of-file marker.

The use of Seek is demonstrated in the following code segment. This code is associated with the change event of the TextBox named SearchText. After clicking the

Show Invoice Table and Set Invoice No Index buttons, this code permits an incremental search through the INVOICE table:

```
If AdsRecordset.State <> adStateOpen Then
  MsgBox "Please open Invoice table before searching
    for an invoice"
  Exit Sub
End If
If AdsRecordset.Index <> "Invoice No" Then
  MsgBox "You must set the Invoice No index " + _"
    "before searching"
  Exit Sub
End If
AdsRecordset.Seek SearchText.Text, adSeekAfterEQ
If AdsRecordset.EOF Then
  MsgBox "End of file"
End If
```

Setting a Filter

You use a filter to limit a Recordset to a subset of records. When executed on a server-side Recordset, Advantage produces an AOF (Advantage Optimized Filter), after which it repopulates the Recordset based on the filtered view.

You set a filter by assigning a filter expression to a Recordset's Filter property. You drop a filter by setting the Filter property to an empty string.

Although the filter expressions that you can assign to the Filter property of a Recordset are similar to those that you can set to Advantage using other mechanisms (such as using the Advantage Data Architect or the Advantage TDataSet Descendant), there is one important difference. If you include a field name that contains embedded spaces, you must enclose the field name in square brace delimiters.

Setting and dropping a filter is demonstrated in the following code, which is located in the click subprocedure for the Set Filter button shown in Figure 22-1:

```
If AdsRecordset.State <> adStateOpen Then
  MsgBox "Please open Invoice table before setting a filter"
  Exit Sub
End If
If FilterBtn.Caption = "Drop Filter" Then
  AdsRecordset.Filter = ""
  FilterBtn.Caption = "Set Filter"
Else
  AdsRecordset.Filter = FilterText.Text
  FilterBtn.Caption = "Drop Filter"
End If
Set DataGrid1.DataSource = AdsRecordset
```

If you run this project, click the Show Invoice Table button and then enter the following filter expression:

```
[Customer ID] = 12037 and [Employee ID] = 89
```

Once you click the Set Filter button, the DataGrid at the bottom of the form will display only two records, as shown in Figure 22-5.

Note: Although a filter does not rely on the current index, the speed with which a filter can be applied is directly related to the available indexes on the table. Specifically, filters can be applied quickly when the expressions in the filter expression map to available indexes on the underlying table.

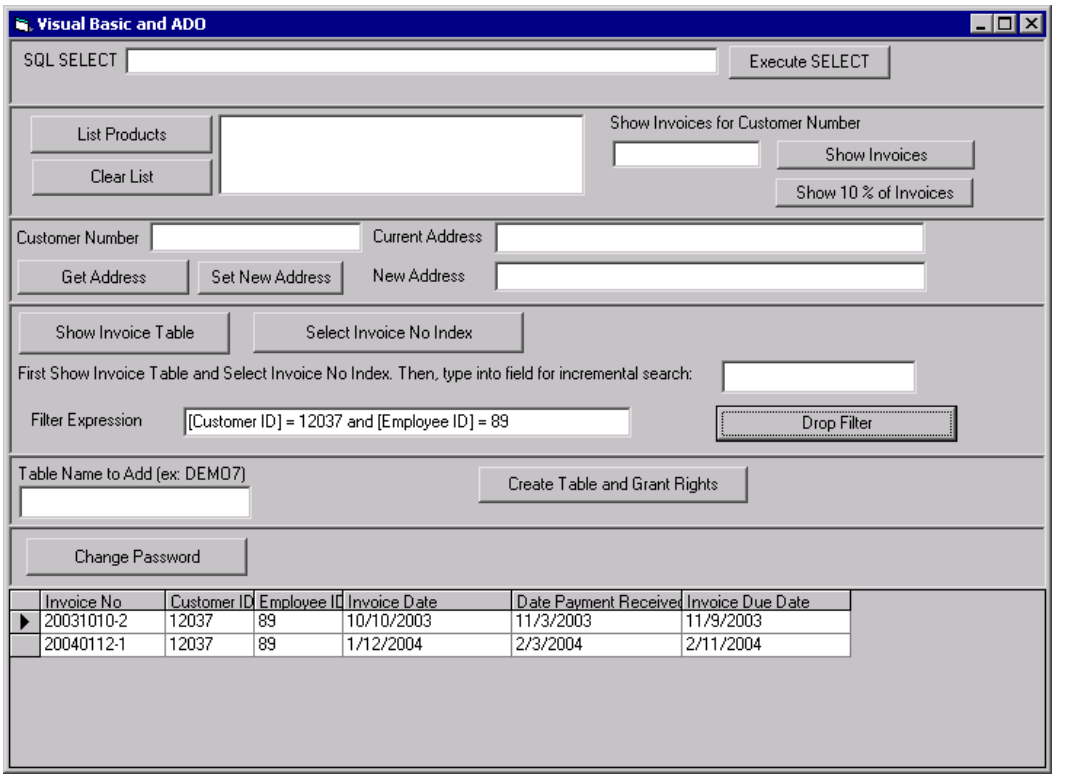

Figure 22-5: A filter has been applied to a Recordset

Scanning a Result Set

Scanning is the process of sequentially reading every record in a result set, or every record in the filtered view of the result set if a filter is active. In most cases, scanning involves an initial navigation to the first record of the result set, followed by repeated calls to advance one record until all of the records have been visited.

Although scanning is a common task, it is important to note that it necessarily requires the client application to retrieve all of the records in the result set.

When using a client-side cursor, all records must be retrieved to the client before any action can be taken. However, once retrieved, the scanning process itself is very fast. By

comparison, when using a server-side cursor, the records are read to the client during the scanning process. Consequently, scanning can initiate faster but may take longer when using a server-side cursor.

Tip: If you are using ADS, and you must scan a large number of records, implement the operation using a stored procedure as described in Chapter 7, "Creating Stored Procedures." When the server and the data reside on the same machine, scanning from a stored procedure installed on ADS requires no network resources.

The following code demonstrates scanning records in a Recordset. This code, associated with the click subprocedure of the List Products button (shown in Figure 22-1), navigates the entire PRODUCTS table, assigning data from each record to the ProductList list box:

```
If AdsRecordset.State = adStateOpen Then
  AdsRecordset.Close
End If
AdsRecordset.Open "SELECT * FROM PRODUCTS", AdsConnection, _
  adOpenDynamic, adLockPessimistic, adCmdText
ProductList.Clear
AdsRecordset.MoveFirst
While Not AdsRecordset.EOF
  ProductList.AddItem (AdsRecordset.Fields(0).Value & _
    vbTab & AdsRecordset.Fields(1).Value)
  AdsRecordset.MoveNext
Wend
AdsRecordset.Close
```

Note: You can improve the performance of scanning operations by using ForwardOnly cursors, which are optimized for forward navigation.

Administrative Operations with Advantage and ADO

While Advantage requires little in the way of periodic maintenance to keep it running smoothly, many applications need to provide administrative functionality related to the management of users, groups, and objects.

This section is designed to provide you with insight into exposing administrative functions in your client applications. Two related, yet different, operations are demonstrated here. In the first, a new table is added to the database and all non-default groups are granted access rights to it. This operation requires that you establish an administrative connection. The second operation involves permitting individual users to modify their own passwords. Especially in the security-conscious world of modern database management, this feature is often considered an essential step to protecting data.

Creating a Table and Granting Rights to It

The VB_ADO.vbp project permits a user to enter the name of a table that will be created in the data dictionary, after which all non-default groups will be granted rights to the table. This operation is demonstrated in the following subprocedure, which is associated with the click event of the Create Table and Grant Rights button shown in Figure 22-1:

```
If TableNameText.Text = "" Then
  MsgBox "Please enter the name of the table to create"
  Exit Sub
End If
'Check for semicolon hack
If InStr(1, TableNameText.Text, ";", _
    vbTextCompare) <> 0 Then
  MsgBox "Table name may not contain a semicolon"
  Exit Sub
End If
If AdsRecordset.State = adStateOpen Then
  AdsRecordset.Close
End If
AdminConnection.ConnectionString = _
  "Provider=Advantage OLE DB Provider;" + _
  "Data Source=" + DataPath + ";user ID=adssys;" + _
  "password=password;" + _
  "ServerType=ADS_LOCAL_SERVER | ADS_REMOTE_SERVER;" + _
    "FilterOptions=RESPECT_WHEN_COUNTING;" + _
    "TrimTrailingSpaces=True"
AdminConnection.Open
Set AdminCommand.ActiveConnection = AdminConnection
AdsRecordset.Open "SELECT COUNT(*) FROM system.tables " + _
  "WHERE UCASE(Name) = UCASE(TableName.Text)", _
  AdminConnection, adOpenDynamic, adLockPessimistic, _
    adCmdText
AdsRecordset.MoveFirst
If AdsRecordset.Fields(0).Value = 1 Then
  MsgBox "This table already exists. Cannot create"
  Exit Sub
End If
AdminCommand.CommandText = "CREATE TABLE " + TableNameText.Text + _
  "([Full Name] CHAR(30)," + _
  "[Date of Birth] DATE," + _
  "[Credit Limit] MONEY, " + _
  "Active LOGICAL)"
AdminCommand.Execute
AdsRecordset.Close
AdsRecordset.Open "SELECT * FROM system.usergroups " + _
  "WHERE Name NOT LIKE 'DB:%'", _
  AdminConnection, adOpenDynamic, adLockPessimistic, adCmdText
If AdsRecordset.BOF And AdsRecordset.EOF Then
  MsgBox "No groups to grant rights to"
  Exit Sub
End If
AdsRecordset.MoveFirst
While Not AdsRecordset.EOF
  AdminCommand.CommandText = "GRANT ALL ON " + _
```

```
    TableNameText.Text + " TO """ + _
    AdsRecordset.Fields(0).Value + """"
  AdminCommand.Execute
  AdsRecordset.MoveNext
Wend
AdminConnection.Close
MsgBox "The " + TableNameText.Text + " table has been " + _
  "created, with rights granted to all groups"
```

This subprocedure begins by verifying that the table name does not include a semicolon, which could be used to convert the subsequent GRANT SQL statement into a SQL script. Since this value represents a table name, a parameterized query is not an option.

Next, this code verifies that the table does not already exist in the data dictionary. Once that is done, a new connection is created using the data dictionary administrative account. This connection is then used to call CREATE TABLE to create the table, and then to call GRANT for each non-default group returned in the system.usergroups table.

Note: The administrative user name and passwords are represented by string literals in this code segment. This was done for convenience, but in a real application, either you would ask for this information from the user or you would obfuscate this information so that it could not be retrieved from the executable.

Changing a User Password

A user can change the password on their own connection, if you permit this. In most cases, only when every user has a distinct user name would you expose this functionality in a client application. When multiple users share a user name, this operation is usually reserved for an application administrator.

The following subprocedure, associated with the Change Password button (shown in Figure 22-1), demonstrates how you can permit a user to change their password from a client application:

```
Dim UserName As String
Dim OldPass As String
Dim NewPass1 As String
Dim NewPass2 As String
If AdsRecordset.State = adStateOpen Then
  AdsRecordset.Close
End If
AdsRecordset.Open "SELECT USER() FROM system.iota", _
  AdsConnection, adOpenDynamic, adLockPessimistic, adCmdText
UserName = AdsRecordset.Fields(0).Value
AdsRecordset.Close
OldPass = InputBox("Enter your current password")
If OldPass = "" Then Exit Sub
If Not CheckPass(UserName, OldPass) Then
  MsgBox "Cannot validate your current password. " + _
    "Cannot change password"
```

```
  Exit Sub
End If
'Get new password
NewPass1 = InputBox("Enter your new password")
If NewPass1 = "" Then
  MsgBox "Password cannot be blank. Cannot change password"
  Exit Sub
End If
'Check for semicolon hack
If InStr(1, NewPass1, ";", vbTextCompare) <> 0 Then
  MsgBox "Password may not contain a semicolon"
  Exit Sub
End If
NewPass2 = InputBox("Confirm your new password")
If NewPass1 <> NewPass2 Then
  MsgBox "Passwords did not match. Cannot change password"
  Exit Sub
End If
'Green light to change password
AdsCommand.CommandText = _
  "EXECUTE PROCEDURE sp_ModifyUserProperty('" + UserName + _
  "', 'USER_PASSWORD', '" + NewPass1 + "')"
AdsCommand.Execute
MsgBox "Password successfully changed. " + _
  "New password will be valid next time you connect"
```

A number of interesting tricks are used in this code. First, the user name is obtained by requesting the USER scalar function from the system.iota table. USER returns the user name on the connection through which the query is executed. Next, the user is asked for their current password, and the user name and password are used to attempt a new connection, which, if successful, confirms that the user is valid. This validation occurs in a subfunction named CheckPass. The following code is found in this function:

```
Private Function CheckPass(UName As String, _
  Pass As String) As Boolean
  Dim TempConnection As ADODB.Connection
  On Error GoTo ErrorHandler
  'Try to make a new connection using this password
  Set TempConnection = New ADODB.Connection
  TempConnection.ConnectionString = _
    "Provider=Advantage OLE DB Provider;" + _
    "Data Source=" + DataPath + ";user ID=" + UName + ";" _
    + "password=" + Pass + ";" +
    "ServerType=ADS_LOCAL_SERVER | ADS_REMOTE_SERVER;"
  TempConnection.Open
  'Password must be ok. Close TempConnection
  TempConnection.Close
  CheckPass = True
  Exit Function
ErrorHandler:
  CheckPass = False
End Function
```

Finally, the user is asked for their new password twice (for confirmation). If all is well, the sp_ModifyUserProperty stored procedure is called to change the user's password. This password will be valid once the user terminates all connections on this user account.

*Note: If you run this code, and change the password of the adsuser account, you should use the Advantage Data Architect to change the password back to **password**; otherwise, you will not be able to run this project again since the password is hard-coded into the connection string.*