Introduction to ADO

By Doug Hennig

Microsoft's latest data access strategy is focused on ActiveX Data Objects (ADO), a COM-based data engine that acts as a front end for OLE DB. OLE DB is bound to replace ODBC as the defacto data connectivity standard and ADO is the interface that most applications will use to access data functionality. ADO provides an object-based approach to data access and through COM allows the ability to pass data as objects between components. A subset of ADO – Remote Data Service (RDS) – also allows access to ADO recordsets over an HTTP connection. This document discusses the technology involved with ADO and provides examples of how you can implement and use ADO in a VFP environment.

OLE DB and ADO

OLE DB is part of Microsoft's Universal Data Access strategy, in which data of any type stored anywhere in any format, not just in relational databases on a local server, can be made available to any application requiring it. It will ultimately replace ODBC as the preferred mechanism for accessing data.

OLE DB providers are similar to ODBC drivers: they provide a standard, consistent way to access data sources. A variety of OLE DB providers are available today (SQL Server, Oracle, Access/Jet, etc.) and more will become available with time. In addition, Microsoft provides an OLE DB provider for ODBC data sources, so anything you can access with ODBC today can be accessed from OLE DB.

Because OLE DB is a set of low-level COM interfaces, it's not easy to work with in languages like Visual FoxPro and Visual Basic. To overcome this, Microsoft created ActiveX Data Objects, or ADO, a set of COM objects that provides an object-oriented front-end to OLE DB. As a result, this session will focus on ADO rather than OLE DB, since that's what we'll be working with directly.

Why Use ADO in VFP

ADO isn't a replacement for native data access in VFP. Manipulating VFP tables and cursors is faster and more feature-rich than using ADO in a VFP application. However, ADO does have its place in VFP components:

- Only VFP knows how to open a VFP table while ADO provides a DBMS-independent mechanism. If your VFP component needs to send data to or receive data from non-VFP components, ADO is the logical choice.

- A VFP cursor only "lives" in the datasession it was opened in. Not only can't you pass a cursor to a non-VFP object, you can't even pass it to a VFP object in another VFP session or the same VFP session but a different datasession.

- ADO makes it easy to pass data over the Internet. While you can also do this with VFP tables (for example, using tools provided in West Wind Technologies' West Wind Web Connection or described in Rick Strahl's book "Internet Applications with Visual FoxPro 6.0"), it's a lot more work.

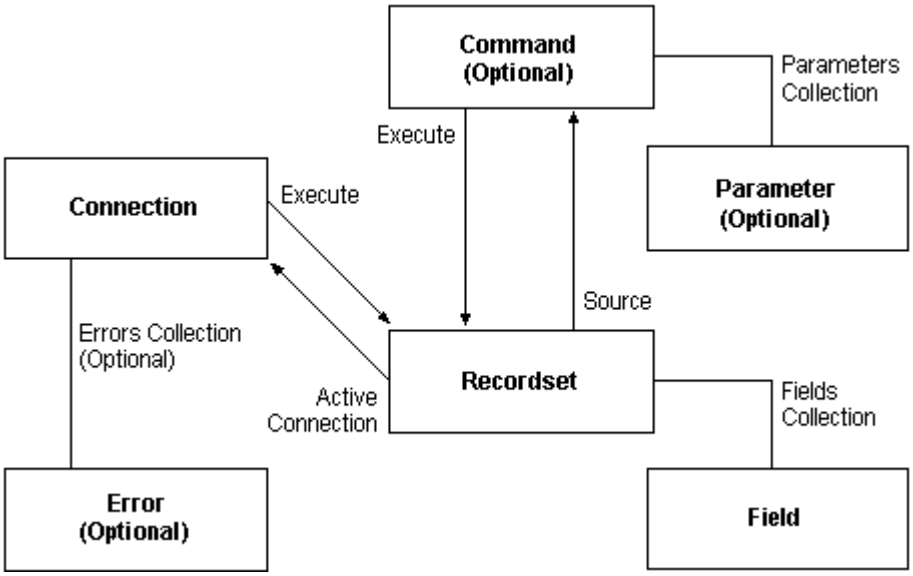
When VFP is used in an n-tier application, ADO makes it easy to transfer data between the tiers. Also, ADO provides an object-oriented interface to data, something VFP developers have asked for almost since VFP was released.

Installing ADO

ADO is included with the Microsoft Data Access Components (MDAC), which are automatically installed by a number of applications, including Microsoft Visual Studio and Microsoft Office 2000. You can also download the latest version of MDAC (MDAC_TYP.EXE) from www.microsoft.com/data. Other tools and documentation can also be downloaded from there.

ADO Object Model

The diagram below shows the object model for ADO. The main objects we work with are Connection and RecordSet, although Command is frequently used as well. The other objects are collections that belong to the main objects, and get used as required.



ADO uses a lot of enumerated properties and parameters (those which accept a pre-defined range of values; the DataSession property of VFP forms, which can be either 1 for default or 2 for private, is an example of an enumerated property). Enumerated values are often defined as constants in the type library of a COM object. Languages like VB can read the type library and automatically allow the use of these constants. VFP, on the other hand, needs an include file of constants if you want to use them in your code. A file accompanying this document, ADOVFP.H, provides the constants used in the ADO object model.

The Connection Object

The ADO Connection object provides the mechanism to access a data source. Where you instantiate a Connection object depends on how you'll use it. In a PRG, you'll instantiate the object into a variable. In a form or class, where you likely want the Connection object to persist beyond a single method, you'll instantiate it into a form property. You might also want to create an ADO manager object that maintains one or more Connection objects for all the components in your application. Here are a couple of examples:

```
oConn = createobject('ADODB.Connection')
This.oConnection = createobject('ADODB.Connection')
```

Connecting to a Data Source

To connect to a data source, use the Open method of the Connection object. There are several ways you can do this.

To use an ODBC data source (which means using the OLE DB Provider for ODBC), pass the Open method the DSN (data source name), user name, and password:

```
oConn.Open('Northwind')
oConn.Open('MyDataSource', 'MyUserName', 'MyPassword')
```

If an ODBC data source doesn't exist, you don't want to create one on every machine that'll run your application, or you want to use an OLE DB provider for a specific database, pass a connection string to the Open method (this is sometimes called a "DSN-less" connection):

```
oConn.Open('Provider=MSDASQL.1;Data Source=northwind')
```

How do you know what to pass for the connection string? There are several ways you can determine it:

- If you're using ODBC, create an ODBC data source (even just temporarily), connect to it, then look at the ConnectionString property of the Connection object. For example, when I connect to the sample NWind database (based on the Microsoft Jet engine) that comes with VB through a DSN called "Northwind", this is what the ConnectionString property contains:

```
Provider=MSDASQL.1;
Data Source=northwind;
Extended Properties="DSN=northwind;
DBQ=D:\Program Files\Microsoft Visual Studio\VB98\NWIND.MDB;
DriverId=281;
FIL=MS Access;
MaxBufferSize=2048;
PageTimeout=5;"
```

- If you have VB, fire it up, create a DataProject, select the DataEnvironment from the Project window, select the Connection1 object, right-click and select Properties, and step through the process of specifying the properties for the Connection, selecting the appropriate OLE DB provider and database. Then look at the ConnectionSource property in the Properties window. When I connect to the NWind database using the Microsoft Jet 4.0 OLE DB Provider, this is what the ConnectionSource property contains:

```
Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=D:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb;
Persist Security Info=False
```

- You can create a data link file, which contains the connection information for a data source. To create a data link file, which has a UDL extension, right-click in a folder in Windows Explorer, select New, and choose Microsoft Data Link. If this option doesn't appear in the menu, create a text file and give it a UDL extension, then right-click on it and choose Properties. In either case, you'll get a dialog in which you can select the desired provider and data source. After you close the dialog, you can either open the file in Notepad and copy the information from this file or tell the Connection object to use it with the Open method:

```
oConn.Open('mydata.udl')
```

The connection string determined using these techniques often has more information than is required to open a connection, so you can either use it as is or use trial and error to determine the minimum information to pass. Here's an example of using the connection string determined from VB:

```
oConn.Open('Provider=Microsoft.Jet.OLEDB.4.0;' + ;
       'Data Source=D:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb')
```

As an alternative to passing a connection string to the Open method, you can set the Connection object's Provider and ConnectionString properties, and then call Open with no parameters:

```
oConn.Provider = 'Microsoft.Jet.OLEDB.4.0'
oConn.ConnectionString = 'Data Source=D:\Program Files\Microsoft ' + ;
       'Visual Studio\VB98\Nwind.mdb'
oConn.Open()
```

Error Handling

What if the connection fails for some reason? ADO doesn't return an error code; instead, it throws an exception, which causes error number 1429 to occur in VFP. So, you must be sure your error handler handles errors coming from ADO.

In addition to the usual information about the error you can get from the AERROR() function, you can also use the Errors collection of the Connection object. The Errors collection consists of Error objects, which have several properties containing information about the errors that occurred. Here's a simple program that demonstrates this (ADOERRORS.PRG in the sample files for this session):

```
on error do ErrHandler
```

```
oConn = createobject('ADODB.Connection')
oConn.Open('Provider=Some bad provider;Data Source=Some non-existent file')
oConn.Open('Provider=Microsoft.Jet.OLEDB.4.0;' + ;
       'Data Source=Some non-existent file')
on error

procedure ErrHandler
aerror(laErrors)
clear
display memory like laErrors
?
? oConn.Errors.Count, 'error(s) occurred'
for each loError in oConn.Errors
       ? 'Error #: ' + transform(loError.Number, '9999999999999999')
       ? 'Description: ' + loError.Description
       ? 'Native error #: ' + transform(loError.NativeError, '9999999999999999')
       ? 'Source: ' + loError.Source
       ? 'SQL state: ' + transform(loError.SQLState, '9999999999999999')
next loError
wait
```

Properties

Besides ConnectionString and Provider, some of the other properties of the Connection object are:

- CommandTimeout: the timeout value (in seconds) for a command to execute.
- ConnectionTimeout: the timeout value for a connection to be made.
- CursorLocation: we'll discuss cursor locations when we discuss the RecordSet object.
- Version: the ADO version number (as of this writing, 2.5 is the latest release version).

See the ADO help file for details on these and other properties.

Methods

Besides Open, some of the other methods of the Connection object are:

- BeginTrans, CommitTrans, and RollbackTrans: similar to the BEGIN TRANSACTION, END TRANSACTION, and ROLLBACK commands in VFP, these methods provide control over transaction processing in ADO. Not all OLE DB providers support transaction processing. We'll look at these in more detail later.
- Close: closes the connection.
- Execute: creates a Command object (we'll talk about these objects later) and has it execute a SQL statement, stored procedure, or other command.
- OpenSchema: creates a RecordSet object containing meta data about the data source. We'll discuss this method in more detail later.

See the ADO help file for details on these and other methods.

The RecordSet Object

Suppose you have a VFP form with a private datasession that opens a cursor. Objects outside the form can't work with the cursor directly, but they can call methods of the form to move the record pointer in the cursor, find a specific record in the cursor, filter or sort the cursor, etc. Properties of the form would hold the values of the fields in the cursor for the current record; changing the values of the properties and then calling an Update method of the form would save the changes made.

This is kind of what an ADO RecordSet is like. It exposes an object-oriented interface to a set of records. Like the analogy above, there's an underlying cursor that we can't access directly (in fact, believe it or not, this cursor is based on the VFP cursor engine!) but instead through properties and methods of the RecordSet object.

Like a Connection object, you first instantiate a RecordSet object, then call the Open method to create the cursor. Here's the syntax:

```
oRS = createobject('ADODB.RecordSet')
oRS.Open(Source, Connection, CursorType, LockType, Options)
```

In this example, Source is the source of data (such as a SQL SELECT statement, although we'll see other types later) and Connection is usually a Connection object (although other things can be passed, such as a Command object or a connection string). Instead of passing the Connection object as a parameter, you could set the ActiveConnection property of the RecordSet object before calling Open.

Let's take a look at various options for opening a recordset (by the way, in this document, I'll use the terms "recordset" and "cursor" interchangeably, whereas "RecordSet" refers to an ADO RecordSet object).

Cursor Types

ADO supports four types of cursors, shown in the table below (the Constant column shows the constant defined in ADOVFP.H). The type can be specified as a parameter in the RecordSet object's Open method or by setting the object's CursorType property prior to calling Open. You'll get an error if you try to change the cursor type after the recordset is open.

Cursor Type	Constant	Value	Description
Forward-only	adOpenForwardOnly	0	The recordset can only be navigated in a forward direction, so it's typically used to fill a control (such as a combobox) or when processing records in a single pass. It provides the fastest access, and is the default cursor type.
Keyset	adOpenKeyset	1	The recordset can be freely navigated. The number of records doesn't change (hence the term "keyset"), but the data in them can, so changes to records, but not additions or deletions, made by other users are visible.
Dynamic	adOpenDynamic	2	The recordset can be freely navigated. Both the number of records and the data in them can change (hence the term "dynamic"), so all changes (including additions and deletions) made by other users are visible.
Static	adOpenStatic	3	The recordset can be freely navigated. Both the number of records and the data in them are fixed, so no changes made by other users are visible.

Cursor Location

The cursor can reside on either the client or the server, as shown in the table below. The location is specified by setting the CursorLocation property of the Connection or RecordSet objects prior to calling Open. You can change the cursor location after opening a connection, but you'll get an error if you try to change the cursor location after the recordset is open.

Location	Constant	Value	Description
Server	adUseServer	2	The cursor resides on the server. This is the default setting.

Client	adUseClient	3	The cursor resides on the client workstation. Many operations on recordsets, including filtering, sorting, and disconnecting the recordset from the server, only work with client-side recordsets.

For client-side cursors, the cursor type is automatically forced to static.

Record Locking

ADO supports four types of record locking, shown in the table below. The locking type can be specified as a parameter in the RecordSet object's Open method or by setting the object's LockType property prior to calling Open. You'll get an error if you try to change the locking type after the recordset is open.

Lock Type	Constant	Value	Description
Read-only	adLockReadOnly	1	The recordset is read-only. This type provides the fastest access, and is the default type.
Pessimistic	adLockPessimistic	2	Like VFP's pessimistic row buffering, a lock is attempted as soon as the record is edited. This locking type is only available for server-side recordsets.
Optimistic	adLockOptimistic	3	Like VFP's optimistic row buffering, a lock in attempted when the record is updated.
Batch Optimistic	adLockBatchOptimistic	4	Like VFP's optimistic table buffering, a lock is attempted on each modified record when the entire batch is updated.

Command Type

You can use one of the six command types shown in the table below to open a recordset. The command type tells ADO how to interpret the command text. It can be specified as a parameter in the recordset object's Open method, as a parameter in a Command object's Execute command, or by setting the Command object's CommandType property prior to calling its Execute method (we'll look at the Command object later). You'll get an error if you try to change the command type after the recordset is open.

Command Type	Constant	Value	Description
SQL SELECT	adCmdText	1	The command text is a SQL SELECT statement.
Table	adCmdTable	2	The command text specifies the name of a table to open using SELECT * FROM [tablename].
Stored Procedure	adCmdStoredProc	4	The command text is the name of a stored procedure to execute.
Unspecified	adCmdUnknown	8	The command type isn't specified but is assumed to be a SQL SELECT statement.
File	adCmdFile	256	The command text is the filename of a disconnected recordset.
Table direct	adCmdTableDirect	512	The command text specifies the name of a table to open directly.

Opening a Recordset

OK, with all those issues out of the way, let's look at some examples of opening a recordset. This one creates a recordset of all records in the NWind CUSTOMERS table (this assumes you've already instantiated Connection and RecordSet objects):

```
oConn.Open('Provider=Microsoft.Jet.OLEDB.4.0;' + ;
       'Data Source=D:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb')
oRS.Open('select * from customers', oConn)
```

Alternatively:

```
oRS.ActiveConnection = oConn
oRS.Open('select * from customers')
```

Here's a different way of doing the same thing:

```
oConn.Open('Provider=Microsoft.Jet.OLEDB.4.0;' + ;
       'Data Source=D:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb')
oRS = oConn.Execute('select * from customers')
```

Rather than creating a Connection object, you can pass a connection string as the second parameter to the Open method. ADO will still create a Connection object; the ActiveConnection property of the RecordSet object contains a reference to it. The downside to this approach is that ADO will create a new connection every time you do this, whereas creating your own Connection object allows you to share that connection.

Fields in a Recordset

The fields in a recordset are accessed through the Fields collection, a collection of Field objects that contain the data for a record. You can access members of the collection either by their index number (which is zero-based, so the first item in the collection has index 0, not 1) or by name. Using index numbers is more convenient when you're processing all fields (although you can also use the FOR EACH syntax) while using field names is easier when working with a specific field. For example:

```
? oRS.Fields(0).Value
? oRS.Fields('Company').Value
```

would display the same thing if the first field in the collection is Company. Note that unlike a field in a VFP cursor, which you can get the value of by simply referring to the field, you must request the Value property of the field object to get the value for the field in the current record. Here are the more common properties of the Field object:

- Name: the name of the field.
- Value: the value of the field in the current record.
- OriginalValue: like the VFP OLDVAL() function, this property contains the value of the field as it was before any changes were made.
- UnderlyingValue: like the VFP CURVAL() function, this property contains the current value of the field in the data source. This is only applicable in a server-side cursor.
- Type: the data type of the field. Unlike VFP's data types, which are a single character, ADO's data types are a numeric value. See ADOVFP.H for a list of the constants and their values.
- DefinedSize: the size of the field.
- ActualSize: the value of this property is like using LEN(TRIM(FIELD)) in VFP; that is, it returns the length of the actual data, not the length of the field. However, the value you get might be double what you're expecting (for example, a value of 25 characters might give ActualSize of 50). I believe this is caused by Unicode.
- Precision: the number of digits in a numeric field.
- NumericScale: the number of decimal places.

- Properties: like the Properties collection of a Connection object, this is a dynamic collection of additional properties for a Field. Because not all data providers support all features, the properties in this collection will vary from provider to provider.

Navigating, Sorting, and Filtering a Recordset

Like a VFP cursor, the fields in an ADO recordset contain the values of the current record only. So, we need to navigate through the recordset to access more than one record.

The table below lists navigation-related properties of a RecordSet object. In the Cursor Type column, "F" means forward-only cursors, "K" means keyset cursors, "D" means dynamic cursors, and "S" means static cursors. In the Cursor Location column, "C" means client-side cursors. Unless otherwise specified, all properties are applicable to all cursor types and locations.

Property	Description	Cursor Type	Cursor Location
AbsolutePosition	The relative position of a record in the recordset. This can change as records are sorted and filtered. Like VFP's GO command, setting AbsolutePosition moves the record pointer to the desired record.	K, S	
BOF	Like VFP's BOF(), this property is .T. if the record pointer is moved before the first record in the recordset.		
Bookmark	Similar to VFP's RECNO(), except it doesn't necessarily start with 1 or increment by 1 when you're using server-side cursors. This value doesn't change for a record during the life of the recordset. Setting Bookmark is like using GO in VFP.	K, S	
EOF	Like VFP's EOF(), this property is .T. if the record pointer is moved past the last record in the recordset.		
Filter	Like VFP's SET FILTER command, setting this property to an expression filters the recordset.		C
RecordCount	Similar to VFP's RECCOUNT(), except it contains the count of records currently visible, so this number will change if a filter is set, for example.	K, S	
Sort	Determines how a recordset is sorted.	S	C

The next table shows some of the navigation-related methods of a RecordSet object. The Cursor Type and Cursor Location columns have the same meaning as they do in the previous table.

Method	Description	Cursor Type	Cursor Location
Find	Locates the next record matching the desired criteria (specified similar to the WHERE clause in a SQL SELECT statement).	K, D, S	
Move	Similar to VFP's GO command, but accepts two parameters: the number of records to move and the position from which to begin the move.		
MoveFirst	Like VFP's GO TOP, moves to the first record in the recordset.		
MoveLast	Like VFP's GO BOTTOM, moves to the last record in the recordset.	K, D, S	
MoveNext	Like VFP's SKIP 1 command, moves to the		

	next record in the recordset.		
MovePrevious	Like VFP's SKIP –1 command, moves to the previous record in the recordset.	K, D, S	
Requery	Like requerying a VFP view, this refreshes the recordset (losing any pending changes as a result) by in effect closing and reopening the recordset.		

Navigating

The Move methods, along with the EOF and BOF properties, are typically used to navigate through a recordset. Here's an example of processing all the records in a recordset:

```
oRS.MoveFirst()
do while not oRS.EOF
	* do something with the record
	oRS.MoveNext()
enddo while not oRS.EOF
```

If you need to process some records, then come back to one you were on previously, save the value of the Bookmark property (you can also use AbsolutePosition) to a variable and restore it later. Note that this property is only available for certain cursor types.

```
lnBookmark = oRS.Bookmark    && save the current record position
* do something navigating and processing
oRS.Bookmark = lnBookmark    && go back to the record
```

Interestingly, for forward-only cursors, MoveFirst is allowed (so you can move through the cursor more than once), but MoveLast is not.

Try this out in the ADO Test Bench form and see how different types and locations of cursors affect navigation abilities and how AbsolutePosition and Bookmark change as you navigate.

Sorting

How a recordset is sorted determines the order in which records are navigated. Set the Sort property of the recordset to a comma-delimited list of fields and sort order ("asc" for ascending or "desc" for descending). Here are some examples:

```
oRS.Sort = 'City, Company'
oRS.Sort = 'OrderDate desc'
```

Note that only client-side cursors can be sorted.

To try this out in the ADO Test Bench form, select a client-side cursor, enter the sort expression, and tab out of the Sort editbox. Then choose First and Next to see how the sort order has changed. Notice that AbsolutePosition now reflects a record's position within the sorted recordset.

Searching

ADO's Find method is similar to LOCATE. Pass Find an expression similar to the WHERE clause in a SQL SELECT statement to find the next record meeting that expression (to find the first record, call MoveFirst before Find). Note that only a single expression is allowed.

```
oRS.Find("Country = 'Canada'")
oRS.Find("City like 'B%'")
```

If the current record matches the condition, it stays the current record. If the search fails, the EOF property of the RecordSet object is .T.

To try this out in the ADO Test Bench form, select something other than a forward-only cursor, enter a search expression, and tab out of the Find editbox.

Note that ADO is picky about quotes. For example, searching for Country = "Canada" will fail while Country = 'Canada' works.

Filtering

Filtering an ADO recordset is very similar to filtering a VFP cursor. Set the Filter property to the filter expression; the recordset is automatically positioned to the first matching record.

```
oRS.Filter = "Country = 'Canada' and Title = 'Manager'"
```

Note that only client-side cursors can be filtered.

To try this out in the ADO Test Bench form, select a client-side cursor, enter a filter expression, and tab out of the Filter editbox. Notice that the RecordCount property now reflects the number of records that match the filter condition, and AbsolutePosition represents the position within the set of visible records.

Indexing

Indexes in ADO are used to optimize searches and sorts, not to determine the order records are accessed. Setting the Optimize property of a Field object creates a temporary index on that field. Optimize is one of the properties in the dynamic Properties collection of a field object; it's only available for cursor-side recordsets. Here's an example:

```
oRS.Fields('Company').Properties('Optimize').Value = .T.
```

Try this out in the ADO Test Bench form by select a client-side cursor, selecting a field, and checking the Optimize checkbox. Notice that the navigation order of the recordset doesn't change (actually, since the recordset is small, you won't really see any change in any operation).

Data Editing

Data editing with ADO recordsets works a lot (at least in concept) like it does with VFP tables. For example, you can bind controls in a form to fields in an ADO RecordSet. Typically, the RecordSet object will be referenced through a form property, so you'd set the ControlSource of a textbox to something like this:

```
Thisform.oRS.Fields('Company').Value
```

Be sure to set the InputMask or MaxLength properties of controls so the user can't enter a value longer than the field can hold; unlike VFP, which would simply truncate the data, ADO will give an error.

One exception to this is Grids: they can only be bound to VFP cursors, not ADO RecordSets. Fortunately, there are two solutions to this problem. One is to use Microsoft FlexGrid or Hierarchical FlexGrid controls, ActiveX controls that come with Visual Studio. These grids are more powerful than VFP's Grid (for example, you can select ranges of cells like a spreadsheet rather than just one cell at a time) and have a DataSource property you can set to an ADO RecordSet object. See FLEXGRID.SCX for an example. The other solution is to use the VFPCOM RSToCursor method to convert a RecordSet to a VFP cursor, and then bind the cursor to a VFP Grid. We'll talk about VFPCOM later.

You can add a new record to a recordset using the AddNew method, and delete a record using Delete. The Update and Cancel methods are like the VFP TABLEUPDATE() and TABLEREVERT() functions, respectively: they write or cancel changes to the current record. UpdateBatch and CancelBatch are the batch equivalents; they write or cancel changes to all modified records (these methods are only applicable when optimistic batch locking is used).

As in VFP, an update can fail for a variety of reasons, such as invalid values or contention issues. Your code must handle such failure just as it must with VFP data: detect the failure (rather than returning .F., the Update and UpdateBatch methods will cause an error to occur), check what went wrong (use the Errors collection of the Connection object), and try to resolve the problem (in the case of contention,

perhaps use the OriginalValue, Value, and UnderlyingValue properties of each field to determine if mutually exclusive changes occurred or not).

ADO supports transactions via the BeginTrans, CommitTrans, and RollbackTrans methods of the Connection object. As with VFP transactions, it's a good idea to wrap updates in transactions, especially if multiple records are involved in the update. Here's a simple example:

```
llError = .F.
on error llError = .T.
oConn.BeginTrans()
* do some changes to the data
oRS.UpdateBatch()
if llError
	oConn.RollbackTrans()
	oRS.CancelBatch()
else
	oConn.CommitTrans()
endif llError
```

You can test data editing in the ADO Test Bench form by setting the desired locking type, editing the data in the textboxes provided, and using the Update, Update Batch, Cancel, and Cancel Batch buttons. You'll notice that, as you might expect, when you make a change to a record when pessimistic locking (which is only available for server-side cursors) is used, the change is written to the record right away. If you use optimistic locking, the change is written when you choose Update or move off the record.

By default, ADO will return the entire recordset to the server when you update. However, you can set the MarshalOptions property to return only modified records. See ADOVFP.H for the values for these two choices.

Other Properties and Methods

The RecordSet object's Supports method is useful to determine what capabilities the recordset has so you don't try to perform some action that isn't allowed. These capabilities are based on both the type of recordset and the OLE DB provider used. Pass this method the appropriate value for the capability you're interested in (see the CursorOptionEnum section of ADOVFP.H) and it'll return .T. if the recordset supports that feature. As an example, the ADO Test Bench form only enables the Previous button when Thisform.oRS.Supports(adMovePrevious) returns .T.

The Connection object's OpenSchema method creates a recordset containing meta data for the open connection. This is useful for generic routines where, for example, you want to display field captions rather than field names to a user. Pass the appropriate value for the items of interest, such as adSchemaTables (20) for tables and adSchemaColumns (4) for fields. The ADO Test Bench form demonstrates this method.

The RecordSet object's GetRows method converts a recordset into an array. Such an array could be useful for a lot of things, such as providing the RecordSource for a listbox or combobox. Also, because VFP's array handling functions are somewhat limited, an interesting idea is to use an ADO RecordSet in place of an array. This allows you to filter and sort on multiple columns (and control ascending vs. descending order), and quickly convert it to an array for those things (like the RecordSource of a listbox) where VFP can't use a RecordSet object. You can even use a fabricated recordset (we'll talk about those in a moment) as the source of data for the array. One issue: the array returned by GetRows is in column-major rather than VFP's usual row-major order, so you'll need to flip it around to make it useful. See ADOGETROWS.PRG for an example of using GetRows and a function to flip the array.

The GetString method is similar to GetRows, but puts the records into a string rather than an array. You can specify both row and column delimiters.

Disconnected Recordsets

There are times when having a recordset connected to its data source may not be convenient:

- You're using SQL Server or some other database for which you pay license fees per seat. In this case, you want to minimize concurrent connections.

- You want to send the recordset to another computer that may not be connected to the data source (such as over the Internet).

- The typical "road warrior" scenario: you want a traveling salesperson to take a set of data with them so they can place orders, view or modify customer records, etc., and then update the head office data when they get back.

Disconnected ADO RecordSets are similar to offline views in VFP: you have a set of data, just no connection to the source of that data.

A RecordSet is disconnected from its data source by setting its ActiveConnection property to .NULL.; the RecordSet must use batch optimistic locking for obvious reasons. While it's disconnected, you can make changes to records, add and delete records, sort, filter, search, and anything else you might do with a connected recordset except, of course, write the changes back to the data source or requery the recordset. To reconnect the RecordSet, simply establish a connection and set the RecordSet's ActiveConnection property to the Connection object.

If you want a disconnected recordset to persist beyond the current session of the form or application it was created in, use the Save method to write the recordset to disk. Pass this method the name of the file to create. The default format is a proprietary one called ADTG, but you can also save it in XML or HTML by passing the appropriate value for the second parameter; see the table below for the choices.

File Type	Constant	Value
ADTG	adPersistADTG	0
XML	adPersistXML	1
HTML	adPersistHTML	2

To load a saved recordset, pass the name of the file as the first parameter to the Open method. For example:

```
oRS.Open('customer.rs', , adOpenStatic, adLockBatchOptimistic)
```

The ADO Test Bench form has an optiongroup of these file types and a Save button so you can try this out. If you have IE 5.0 or later, you can view an XML-persisted recordset as a formatted document; otherwise, you can use Notepad to view the file.

Command Object

The Command object is mainly used to execute commands that either don't return a recordset (such as a SQL DELETE or UPDATE statement or a stored procedure) or require parameters (such as a parameterized query). The ProgID for the Command object is ADODB.Command.

The ActiveConnection property contains a reference to a Connection object. The CommandText property is the command to execute, such as a SQL statement or the name of a stored procedure. The CommandType property defines what type of command CommandText contains; the choices are the same as those for the RecordSet CommandType property we looked at earlier.

Once these properties have been set, call the Execute method to execute the command. Here's an example that deletes certain records (it assumes a connection has already been made through the oConn object):

```
oCommand = createobject('ADODB.Command')
oCommand.CommandText = "delete * from customers where country <> 'US'"
oCommand.CommandType = adCmdText
oCommand.ActiveConnection = oConn
oCommand.Execute()
```

Here's an example that calls a stored procedure:

```
oCommand.CommandText = 'PurgeRecords'
oCommand.CommandType = adCmdStoredProc
oCommand.ActiveConnection = oConn
oCommand.Execute()
```

Like VFP, ADO supports parameterized queries (and some DBMS, like SQL Server, support parameterized stored procedures). To define parameters for a command, add items to the Parameters collection of the Command object and set the appropriate value for each Parameter item. To create a Parameter object, use the Command object's CreateParameter method. You can either pass it the name of the parameter, the data type, the parameter "direction" (input, output, both input and output, or return value), the size, and the value, or set the appropriate properties of the Parameter object after you create it. See ADOVFP.H for the constants used for the data type and parameter direction. After creating the Parameter object, add it to the Parameters collection using the Append method. Here's an example:

```
oParameter = oCommand.CreateParameter('Country', adChar, adParamInput, 15)
oCommand.Parameters.Append(oParameter)
* get the value for the country into lcCountry
oParameter.Value = lcCountry
oCommand.CommandText = 'select * from customers where country = ?'
oRS = oCommand.Execute()
```

Notice the slight difference in syntax between ADO and VFP: the ? here isn't followed by a variable name as it would be in a VFP parameterized query. Also notice that unlike the previous examples, this time Execute returns a recordset.

One problem with this approach: the recordset is read-only, server-side, and forward-only. If you want to control the attributes of the recordset, create a RecordSet object, set its properties as desired, then pass the Command object to the RecordSet's Open method.

```
oRS = createobject('ADODB.RecordSet')
oRS.CursorLocation = adUseClient
oRS.CursorType = adOpenStatic
oRS.Open(oCommand)
```

You can create multiple parameter objects. The order in which they're added to the Parameters collection is the order in which they're applied to ? in the command.

Fabricated RecordSets

In addition to opening a recordset from a data source, you can create a recordset on the fly. Use the Append method to add fields to the Fields collection of the RecordSet object, passing it the name of the field, the data type, and, if necessary, the size. As I mentioned earlier, data types are numeric values, so use the constants defined in ADOVFP.H. Here's an example that creates a recordset with three fields:

```
loRS = createobject('ADODB.RecordSet')
loRS.Fields.Append('Part',     adChar, 30)
loRS.Fields.Append('Quantity', adUnsignedInt)
loRS.Fields.Append('Level',    adUnsignedTinyInt)
loRS.Open()
```

You can then add records to the recordset and pass the recordset to other objects as desired. This is often used in VFP to return a result set from some operation to something expecting an ADO RecordSet.

Hierarchical RecordSets

ADO has an incredibly cool capability: it can create hierarchical recordsets. A hierarchical recordset has records from both parent and child (and grandchild and grandgrandchild, etc.) tables. The way it works is that one of the fields in the recordset actually contains another recordset. Besides having all the records in a single hierarchical object, hierarchical recordsets have another advantage: they only require one round-

trip to the server; getting individual recordsets would each require one round-trip. Also, it avoids sending redundant data (if you do a join, fields from the parent table are repeated in every record).

To create a hierarchical recordset, use the MSDataShape provider. Here's an example of how to open a datasource using this provider:

```
oConn.Open('Provider = MSDataShape;' + ;
    'Data Source=D:\Program Files\Microsoft Visual Studio\VB98\Nwind.mdb;' + ;
    'Data Provider=Microsoft.Jet.OLEDB.4.0')
```

You then use the Shape language (similar to SQL) to open the recordset. Shape is beyond the scope of this document, but here's an example that opens a recordset showing all fields from the Customers table, and within each customer record, all fields from the Orders table for orders for the customer, and within each order, all fields from Order Details table for details for the order.

```
oRS.Open('shape {select * from Customers} as Customers ' + ;
    'append (( shape {select * from Orders} as Orders ' + ;
    'append ({select * from `Order Details`} as OrderDetails ' + ;
    'relate OrderID to OrderID) as OrderDetails) as Orders ' + ;
    'relate CustomerID to CustomerID) as Orders', oConn)
```

Like the Microsoft FlexGrid control I discussed earlier, the Hierarchical FlexGrid can be bound to an ADO recordset. The difference between these controls is that the Hierarchical FlexGrid can properly displayed a hierarchical recordset, using TreeView-like plus signs to drill down into a child recordset. Run the HIERFLEXGRID sample form to see how this works.

RDS

ADO allows access to data on the same machine or network as the application. Remote Data Services (RDS) allows access to data over the Internet via HTTP. RDS has a different object model than ADO, but supports similar capabilities.

The main client-side object is the RDS DataControl (ProgID RDS.DataControl). You need to set three properties of the DataControl object to connect to a remote data source:

- Server: the address for the server
- Connect: the connection string
- SQL: the command to execute

After setting these properties, call the Refresh method to connect to the server and create a recordset in the RecordSet property. Behind the scenes, the following happens:

- The DataControl object instantiates an RDS.DataSpace object (the RDS equivalent of an ADO Connection object).
- The DataSpace object by default instantiates an RDSServer.DataFactory object on the server (other COM objects can also be used).
- The DataFactory object on the server creates an ADO recordset and converts it to a text stream which is returned by the server to the client.
- The DataSpace object converts the text stream to a recordset object and returns it to the DataControl object.

Here's an example:

```
oDC = createobject('RDS.DataControl')
oDC.Server = 'http://www.myserver.com'
oDS.Connect = 'dsn=Northwind'
```

```
oDS.SQL = 'select * from customers'
oDS.Refresh()
? oDS.RecordSet.Fields('CompanyName').Value
```

You can work with the recordset as you would any other ADO client-side recordset. To update the data source with changes made to the recordset, call the SubmitChanges method of the DataControl object.

Instead of working with the DataControl object, you can use the ADO Connection object by specifying MS Remote as the provider and the server address in the connection string. Here's an example:

```
oConn = createobject('ADODB.Connection')
oConn.Provider = 'MS Remote.1'
oConn.ConnectionString = 'Remote server=http://www.myserver.com;' + ;
    'dsn=Northwind'
oConn.Open()
oRS = createobject('ADODB.RecordSet')
oRS.Open('select * from customers', oConn)
```

VFPCOM.DLL

VFP 6 Service Pack 3 introduced a new tool for VFP developers: VFPCOM.DLL. This COM object has several methods useful with ADO, including:

- RSToCursor: converts an ADO RecordSet to a VFP cursor, usually so it can be bound to a grid. Pass the RecordSet object as the first parameter and the name of the cursor as the second. The Browse button in the ADO Test Bench form shows an example of this.

- CursorToRS: converts a VFP cursor to an ADO RecordSet. The RecordSet object must already be instantiated; pass a reference to it as the first parameter and the name of the cursor as the second. Currently, CursorToRS has a bug in that it doesn't respect filters, deleted records, or SET KEY settings (it simply copies all records from the cursor to the RecordSet), so you'll need to handle that if it's an issue.

- BindEvents: allows a VFP object to respond to events of COM objects. ADO fires events in lots of places, such as while a recordset is being fetched and when it's done. Unfortunately, by itself, VFP has no mechanism to receive notification about such events because, unlike Visual Basic, it has no WITH EVENTS command. VFPCOM's BindEvents method allows you to bind a VFP object to a COM object so when the COM object fires an event, a method of the VFP object with the same name as the event will be called. BindEvents is beyond the scope of this document; see the documentation that accompanies VFPCOM for more information.

The ProgID for VFPCOM is VFPCOM.ComUtil. Here's an example of the RSToCursor method:

```
oCom = createobject('VFPCOM.ComUtil')
oCom.RSToCursor(oRS, 'tempcursor')
browse
```

Summary

Microsoft's UDA strategy, including OLE DB, ADO, and RDS, is the data mechanism of choice for Windows DNA applications. It's easy to use in VFP, both for getting data from other platforms and for sending data from VFP components. This document only scratched the surface of what's in ADO today, so be sure to look at the references listed below to learn more.

References

The "bible" of ADO information for VFP developers is John Petersen's excellent "ADO Jumpstart for Microsoft Visual FoxPro Developers", available at msdn.microsoft.com/library/techart/adojump.htm.

Microsoft Data Access SDK documentation is available at www.microsoft.com/data.

"Microsoft Visual InterDev 6.0 Web Technologies Reference" from Microsoft Press has a large ADO reference section.

A lot of books are now available on ADO. One good one is "ADO 2.1 Programmer's Reference" from Wrox Press, but there are a lot of others available.

Acknowledgements

I would like to acknowledge the following people who directly or indirectly helped with the information in this document: John Petersen, Miriam Liskin, and John Koziol.

Doug Hennig
Partner
Stonefield Systems Group Inc.
1112 Winnipeg Street, Suite 200
Regina, SK Canada S4R 1J6
Phone: (306) 586-3341
Fax: (306) 586-5080
Email: dhennig@stonefield.com
World Wide Web: www.stonefield.com

Biography

Doug Hennig is a partner with Stonefield Systems Group Inc. in Regina, Saskatchewan, Canada. He is the author or co-author of Stonefield's add-on tools for FoxPro developers, including Stonefield Database Toolkit, Stonefield Query, and Stonefield Reports. He is also the author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's "The Pros Talk Visual FoxPro" series. Doug has spoken at the 1997, 1998, and 1999 Microsoft FoxPro Developers Conferences (DevCon) as well as user groups and regional conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Microsoft Certified Professional (MCP).