Paper 12-26

Migrating Your SAS ODBC Application To ADO

David Shamlin, SAS Institute, Cary, NC

ABSTRACT

Do you use the SAS ODBC driver to incorporate SAS data into your applications? You can simplify your applications by using Microsoft's ActiveX Data Objects (ADO) and additional features supported through the SAS/SHARE Data Provider. This paper describes how to convert your ODBC data sources to ADO Connection objects, how to use these objects to access SAS data and how to leverage SAS/SHARE servers using ADO.

INTRODUCTION

With release of Version 8 of the SAS System, SAS introduced support for Microsoft's OLE DB specification and ActiveX Data Objects (ADO). OLE DB is Microsoft's successor to ODBC. While ODBC is a procedural interface, OLE DB defines an object-oriented model for accessing data. ODBC is also inherently SQL-centric; OLE DB allows a DBMS to expose any kind of command processing language, and OLE DB providers make it possible to directly access tabular data directly without the overhead of any query processing.

SAS is committed to supporting OLE DB and ADO. Three beta quality OLE DB providers shipped with 8.0 and became production in 8.0M1:

- The SAS Local Data Provider which ships with SAS/BASE Software for Windows software
- The SAS IOM Data Provider which ships with SAS/Integration Technologies Software
- The SAS/SHARE Data Provider which ships with SAS/SHARE Software

The SAS Local Data Provider gives you the ability to read SAS data sets outside of a SAS session; it is similar to the Universal ODBC driver in this regard. The SAS IOM Data Provider supports the ability to query and update data sets via a SAS/Integration Technology IOM server. The SAS/SHARE Data Provider implements the same set of features against SAS/SHARE servers.

This paper focuses on converting SAS ODBC driver applications to SAS/SHARE Data Provider applications using ADO, but the principles and techniques outlined here are equally suitable for use with the other two providers.

INSTALLING THE SAS/SHARE DATA PROVIDER

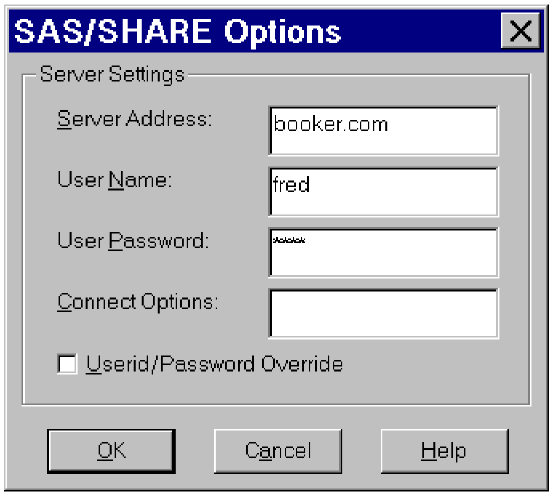
The SAS/SHARE Data Provider is shipped with SAS/SHARE software. While the SAS/SHARE Data Provider is a Windows specific component, it is distributed with SAS/SHARE for all operating systems. The SAS/SHARE Data Provider install is found in the sasmisc directory of the SAS/SHARE product install hierarchy (i.e., $SASROOT/Share/sasmisc). The self-extracting setup program is called shroledb.exe. You are welcome to download and run this setup application on any Windows PC in your organization.

The setup will copy and register the needed executables onto your machine. It will also copy the usage documentation; the documentation is distributed as compiled HTM help files. You can find these files in C:\Program Files\SAS Institute\Shared Files\SAS OLE DB Data Providers. The SAS/SHARE Data Provider is available on the SAS/SHARE software download page of the SAS web site. The copy of the provider available on the web will have the latest maintenance fixes and documentation updates.

MAPPING AN ODBC DATA SOURCE DEFINITION TO AN ADO CONNECTION

Most ODBC savvy users are familiar with the ODBC Data Source Administrator and how this control panel is used to specify data source connection information. When you define connection information using this administrative tool, the ODBC driver manager stores the information in the ODBC.INI file or the Windows system registry. To open an ADO Connection, you specify the same basic information required to connect to an ODBC data source. While it is the normal practice to persist ODBC data source connection information separately from the ODBC application, ADO Connection information is often encoded in the application itself or entered by the user at the time the application is run.

The first step required to specify a SAS ODBC Driver DSN is to give the server options. Figure 1 shows the SAS ODBC driver configuration dialog that defines the server connection information for a hypothetical server. The server is running on a machined identified as booker.com.


SAS/SHARE Options

Server Settings

Server Address: booker.com

User Name: fred

User Password: ****

Connect Options:

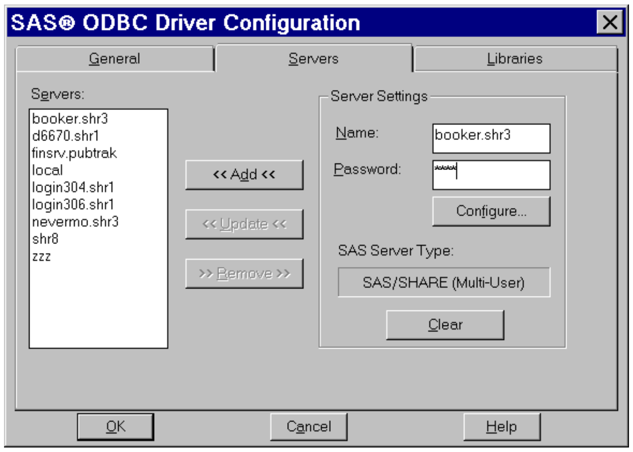
☐ Userid/Password Override

Figure 1

The ID of the SAS/SHARE server is shr3; this is reflected in the server settings seen in Figure 2. The server is associated with an ODBC data source on the General tab of the ODBC driver configuration dialog. Figure 3 shows a data source named "SUGI 26 Example" that uses our hypothetical server.

The same information needed to open an ODBC connection is needed to open an ADO Connection. There are a number of different ways to build and pass connection information to ADO. Here I will outline a simple programmatic solution using VisualBasic. (You could also use VBScript, Jscript, C, C++, or any programming language that supports COM bindings; the syntax will differ but the semantics are the same.) The first step is to declare an ADO Connection object and associate it with the OLE DB provider you want to use (in this case the SAS/SHARE Data Provider).

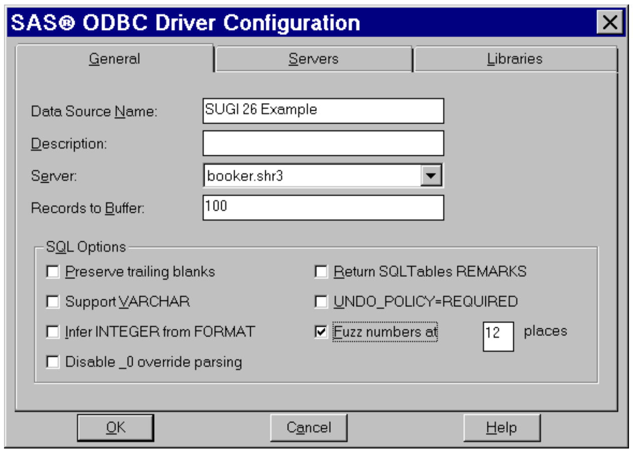
```
Dim obConnection As New ADODB.Connection
obConnection.Provider = "sas.ShareProvider.1"
```



Servers:
booker.shr3
d6670.shr1
finsrv.pubtrak
local
login304.shr1
login306.shr1
nevermo.shr3
shr8
zzz

Server Settings
Name: booker.shr3
Password: ****
SAS Server Type: SAS/SHARE (Multi-User)

Figure 2



Data Source Name: SUGI 26 Example
Description:
Server: booker.shr3
Records to Buffer: 100

SQL Options
☐ Preserve trailing blanks
☐ Support VARCHAR
☐ Infer INTEGER from FORMAT
☐ Disable _0 override parsing
☐ Return SQLTables REMARKS
☐ UNDO_POLICY=REQUIRED
☑ Fuzz numbers at 12 places

Figure 3

This step is similar to adding a new DSN with the ODBC Data Source Administrator and associating it with the SAS ODBC Driver. You must also specify the SAS/SHARE server ID and the name of the node running the server:

```
obConnection.Properties("Data Source") =
"shr3"
obConnection.Properties("Location") =
"booker.com"
```

You can specify user login authentication and a user access password through Connection properties as well:

```
obConnection.Properties("User ID") = "fred"
obConnection.Properties("Password") = "pwd1"
obConnection.Properties("SAS Server Access
Password") = "pwd2"
```

We have now encoded the connection information found in our ODBC DSN as ADO Connection properties. If you wanted to write the application so that it can be run by more than one user, or if you don't want to hard code passwords directly into your code, you can write your application so that you prompt the user for these values when the application is run.

The next step is to call the Connection object's Open method to begin accessing the server:

```
obConnection.Open
```

This method authenticates the user and establishes a session with the SAS/SHARE server. You can now use the Connection object to manipulate data sets.

ACCESSING YOUR DATA

In general, ODBC allows you to manipulate data using SQL; in particular, the SAS ODBC Driver allows you to pass SQL statements to a SAS/SHARE server and exploit the power of SAS/SHARE and SAS/ACCESS software. The SAS/SHARE Data Provider supports the same SQL processing abilities as the SAS ODBC Driver. In addition, the SAS/SHARE Data Provider implements the necessary interfaces that allow you to open a data set directly. By using these features in appropriate scenarios, you can reduce the amount of overhead processing (thereby improving performance) and take advantage of more flexible locking models.

First, I will review how you can migrate ODBC calls that make execute a SQL query into an ADO application. Then I will apply some of these ADO constructs to manipulate data sets directly.

EXECUTING A SQL STATEMENT

In terms of the ODBC function calls needed to process a SQL SELECT statement, querying an ODBC driver for a result set is a more involved task than inserting, updating, or deleting rows of data. This section shows the relationship between the basic ODBC and ADO calls involved in sequentially reading the results of a SELECT statement. It is fairly straightforward to extrapolate the basic ADO algorithm we will use here to SQL statements that alter data.

The basic algorithm for processing a SELECT statement with ODBC is

- Prepare the SELECT statement (using the SQLPrepare function.) This step is optional and only required if the SELECT is going to be executed multiple times.
- Set any parameter values (using the SQLSetParam function.)
- Execute the SELECT statement (using the SQLExecDirect or SQLExecute function.)
- Determine the characteristics of the result set (using the SQLNumResultCols and SQLDescribeCol functions.)
- Assign storage for the result set column values you want to fetch (using the SQLBindCol function.)
- Fetch individual rows of data (using SQLFetch) until there are no more rows to process.

ADO simplifies this process by automatically handling several of these tasks. The first ADO step involved is declaring a Command object and coupling it with an open Connection object:

```
Dim obCommand as New ADODB.Command
Set obCommand.ActiveConnection = obConnection
```

Next we can simply set the SQL statement text and execute the query:

```
obCommand.SetText = "SELECT * FROM …"
obCommand.CommandType = adCmdText
Dim obRecordset as ADODB.Recordset
Set obRecordset = ObCommand.Execute
```

The Command object's CommandType property dictates the flavor of the text string you want to process. Setting the value of this property to adCmdText tells ADO you want the command text to be treated like a SQL statement. I will introduce two other supported CommandType settings later.

The Command object's execute method returns an ADO Recordset object if the given command text produces a result set. This is only true for SELECT statements. For other types of SQL statements (INSERT, UPDATE, etc) no Recordset is returned. In these cases, you should just call the Excute method as a single VisualBasic statement and not as part of a Set statement expression.

In general, ADO uses Recordset objects to encapsulate rectangular data. In our example, the Recordset will contain the result set of our SELECT statement. The result set’s column information and first row of data (if there is one) are immediately available to us in the Recordset. Column information and row data are bound to the Recordset’s Fields collection; each element in this collection is a Field. The column’s basic metadata (data type, length, name, etc.) and value for the current row are found in the properties of a Field object.

You can proceed to processing the data returned in the Recordset without having to explicitly bind the columns you want to access; ADO implicitly binds all columns. This does not necessarily cost you performance when you are only using a small subset of the total number of columns. ADO gives the SAS/SHARE Data Provider enough cues so that the provider can transfer data to and from the server in optimized buffers.

To continue with our example, we will sequentially read through the result set and print out each column (or Field) name and value for each row returned. We will process records until we reach the end of the result set:

```
While Not obRecordset.EOF
```

If the SELECT statement executed results in an empty set of records, the Recordsets EOF property will immediately be true and we will never enter our main loop. If records are returned, the Recordset will initially be positioned at the first row returned so we can immediately begin processing the fields in that row:

```
Dim obField as ADODB.Field
For Each obField in obRecordset.Fields
```

As the For loop iterates through the elements of the Recordset’s Fields collection, we can access each field’s metadata and value for the current row. For example, the following will write a field’s name and value to VisualBasic’s Immediate window:

```
Debug.Print obField.Name obField.Value
```

Closing the For loop with a Next statement causes obField to be assigned the value of the next Field in the Recordset’s collection until all fields have been iterated. After the For loop exists, we want to move to the next row in the Recordset:

```
obRecordset.MoveNext
```

The Recordset’s EOF property will be set to True when the Recordset is positioned on the last row in the result set and MoveNext is called. So we want to terminate our While loop with a Wend statement after the MoveNext call. Here is the VisualBasic code pattern outlined above abstracted slightly for general processing:

```
Dim obField as ADODB.Field
While Not obRecordset.EOF
   For Each obField in obRecordset.Fields
      ‘ TODO: process column
   Next obField
   ObRecordset.MoveNext
Wend
```

Since SQL result sets returned by the SAS/SHARE server are inherently sequential and read-only, the above code outlines the most common pattern for processing SELECT statements with the SAS/SHARE Data Provider. The next section discusses opening a data set directly for random and update access.

OPENING A DATA SET

You can create an ADO Recordset object and use it to open a data set directly without an ADO Command object. Calling the Open method on this kind of Recordset object with the adCmdTableDirect command type constant results in directly opening a data set without any SQL processing being involved. The following example opens the data set CLASS in the SASHELP library for random read-only access:

```
Dim obRecordset as New ADODB.Recordset
ObRecordset.Open “SASHELP.CLASS”, _
   obConnection, _
   adOpenDynamic, adLockReadOnly, _
   adCmdTableDirect
```

The first parameter to the Open method specifies the data set to open. For the SAS/SHARE Data Provider the form of this string is always *libname.memname*. The second parameter identifies an ADO Connection; this couples the Recordset to a server session.

The third and fourth parameters dictate the ADO cursor type and lock type values. The SAS/SHARE Data Provider translates this information into data set open and access modes; they dictate whether the data set is opened for read versus update mode and sequential or random access. The values given in the example cause the file to be opened for random, read-only access. (I will discuss other possible values for these parameters later in the paper.)

The final parameter on the Open method is the most interesting parameter in the example. It is related to the CommandType property introduced in the previous section. The value given for this parameter (officially called the Options parameter) indicates the semantic form of the first, or Source, parameter. The adCmdTableDirect value tells ADO to treat the Source parameter value as an OLE DB table identifier; ADO in turn calls the provider in such a way that the data set is opened directly. (I will refer to Recordsets opened in this manner as “direct open” Recordsets.)

Another command type value of interest is adCmdTable. This value tells ADO to build a simple SQL SELECT statement to return all the columns in the table identified by the Source parameter. If you change the Options parameter in the example above from adCmdTableDirect to adCmdTable, ADO will ask the SAS/SHARE Data Provider to execute the query SELECT * FROM SASHELP.CLASS. So the following two Open method calls return the same set of columns and rows.

```
obRecordsetA.Open “SASHELP.CLASS”, _
   obConnection, , , adCmdTableDirect
obRecordsetB.Open “SASHELP.CLASS”, _
   obConnection, , , adCmdTable
```

However, there are differences in how the data is retrieved and how ADO will allow you to manipulate it:

- Rows in obRecordsetA can be accessed randomly and bookmarks are available for “note” and “point” type operations; obRecordsetB can only be processed sequentially.
- obRecordsetA can be updated (assuming a lock type that allows update); obRecordsetB is read-only.
- Since obRecordsetB encapsulates a SQL result set, it incurs the overhead of the SQL processor while obRecordsetB returns the same set of rows without the additional overhead.

As you can see from this example, knowing what features the provider supports and understanding the subtleties of ADO can have a significant impact on the effectiveness and efficiency of ADO applications.

FILTERING DATA

In many instances, a large data set needs to be filtered and only rows that meet a given criterion need to be processed. SQL supports filtering data with the WHERE clause. The following example returns the set of female employees who earn six digit salaries:

```
SELECT * FROM HR.EMPLOYEES
   WHERE SEX = `F` AND SALARY > 100000
```

The SAS/SHARE Data Provider can handle this kind of SQL filtering via the ADO Command object. Alternatively, there are two ways you can filter data returned by a direct open Recordset. The first method allows ADO to filter the data returned from the provider. You can set the Recordset Filter property to a criteria string similar to the WHERE clause above:

```
ObRecordset.Open "HR.EMPLOYEES", _
   obConnection, , , adCmdTableDirect
obRecordset.Filter = _
   "SEX = `F` AND SALARY > 100000"
```

You can reset or clear the filter (with the special adFilterNone property value) at any time. For a single Recordset open this allows you to dynamically change the filtered view of the data. (For more information on the Filter property see the *Microsoft ADO Programmer's Reference*.)

The final method of filtering a direct open Recordset uses the SAS data set WHERE= option. This method causes the filtering process to occur on the server so only rows meeting the selection criteria are returned. The SAS/SHARE Data Provider implements a custom Recordset property that takes as its value a SAS WHERE expression. This option must be set on the Recordset before it is opened since the WHERE expression must be sent to the server with the data set open request. Here is our example scenario rewritten to filter using the data set WHERE= option.

```
Set obRecordset.ActiveConnection = _
   obConnection
obRecordset.Properties("SAS Where") = _
   "SEX = `F` AND SALARY > 100000"
obRecordset.Open "HR.EMPLOYEES", , , , _
   adCmdTableDirect
```

(For more information on the semantics and syntax of SAS WHERE expressions see the *SAS Language Reference: Dictionary*.) The order of the statements here is important. Since the property being set is a custom property, the Recordset needs to "know" what provider implements the property. An open Connection is already associated with a provider; setting the Recordset's ActiveConnection property to the Connection you want to use gives the Recordset access to the underlying provider's custom properties.

Each of these three methods of filtering data has unique benefits and costs. Using SQL SELECT statements with WHERE clauses may be very simple to code if you are converting an ODBC application to ADO. However, you will be limited to sequential, read-only processing. If you choose the Recordset Filter property, you can randomly access rows of data, update them and have the flexibility of changing the filter criteria multiple times on the same open Recordset, but you pay the cost of always fetching all the data set's rows from the server. You can sacrifice this degree of flexibility in favor of having the server filter the data using WHERE= data set option processing. In this latter case you retain the options of random and update access.

UPDATING ROWS OF DATA

The SAS/SHARE Data Provider supports updating rows of data through SQL statements and direct open Recordsets. When converting an ODBC application to ADO it may be quicker to reuse existing SQL UPDATE or INSERT statements than to write ADO code to perform updates on direct open Recordsets. For example if you already have code that builds a valid UPDATE statement, you can reuse that code and simply pass the resulting SQL expression to an ADO command object.

```
Dim strUpdate as String

' Code that builds the UPDATE statement
' goes here

obCommand.Execute strUpdate
```

This Execute method call looks slightly different than the one presented previously. Unlike the Execute method call with a SELECT statement, this call does not return a Recordset because the UPDATE statement does not return a result set.

Simplicity of ADO command objects aside, there are potential benefits to updating data sets using direct open Recordsets:

- Direct open Recordsets may provide a better record locking strategy for your application.
- Using ADO methods and properties to code update operations may result in code that is simpler than code written to construct complex SQL statements.
- For your specific usage scenario, update operations may be more efficient with direct open Recordsets than SQL statements.

The first step involved in updating data with a direct open Recordset is to call the Open method with a lock type that supports update. ADO defines three lock types that support update:

- adLockOptimistic
- adLockBatchOptimistic
- adLockPessimistic

The SAS/SHARE Data Provider supports the first two. If you specify adLockPessimistic as the lock type, ADO and the provider will convert that to adLockBatchOptimistic.

Once you have opened a Recordset object with one of these lock types, you can go about updating rows of data. First you need to position the Recordset on the row you wish to update. The best way to perform such positioning is fairly specific to the application. (Reviewing all the positioning features ADO supports is outside the scope of this paper; see the *Microsoft ADO Programmer's Reference* for more details.) When the Recordset is positioned on the desired row, updating column values is simply a matter of changing the corresponding Field's value property. As an example, here is a code fragment that increases the salary column's amount by 5%:

```
Dim dblBaseSalary As Double
dblBaseSalary = obRecordset.Fields("SALARY")
obRecordset.Fields("SALARY") = _
   dblBaseSalary + (dblBaseSalary * 0.05)
```

This code changes the SALARY column value that is buffered by ADO. To transmit this change to the provider's data cache, call the Recordset's Update method:

```
obRecordset.Update
```

When the Recordset's lock type is adLockOptimistic, the row's changes will be transmitted to the server at the time ADO transmits them to the provider (i.e., when the Update method is called.) If the Recordset is open with the adLockBatchOptimistic lock type, you must also call the UpdateBatch method at some point to transmit changes from the provider's cache to the server. This lock type allows you to buffer changes for more than one row and then transmit the set of changed rows simultaneously. Using the Supports method, you can test the Recordset to see if you need to call UpdateBatch:

```
if obRecordset.Supports(adUpdateBatch) Then _
   ObRecordset.UpdateBatch
```

Exactly how often and at what point in the code you perform this step is up to you and the requirements of your application. To help tie all these concepts together, below is a revision of our salary raise example where everyone in the company is given a 5% raise:

```
obRecordset.Open "HR.EMPLOYEES", _
   obConnection, _
   adOpenForwardOnly, adLockOptimistic, _
   adCmdTableDirect
While Not obRecordset.EOF
   dblBaseSalary = obRecordset!SALARY
   obRecordset!SALARY = _
      dblBaseSalary + (dblBaseSalary * 0.05)
   obRecordset.Update
   obRecordset.MoveNext
Wend
obRecordset.Close
```

Since I explicitly opened the Recordset with a lock type of adLockOptimistic and I know that the SAS/SHARE Data Provider will honor that, I don't have to code the UpdateBatch condition discussed above. Also, I used a shorthand notation for referencing a column value. This gives an indication of one of ADO's fundamental design principles: there are many number of ways to express the same concept.

DATA SET VARIABLE METADATA

To this point I have focused on accessing data sets, navigating rows and manipulating a row's column values. It is also common to need metadata about data set variables (name, type, size, label, etc.) The provider maps data set variables to OLE DB columns, and ADO maps a provider's columns to Fields on the Recordset. The ADO Field object defines properties for rudimentary column metadata (name, type, defined size, etc.) The SAS/SHARE Data Provider maps basic data set variable information to these Field properties in a fairly straightforward fashion as shown in Table 1.

The other Field properties do not map directly to any interesting data set variable metadata, although there is other interesting variable metadata (like label, format and informat information) that can be useful in writing an application but does not map into any of the Field object's properties. Fortunately, OLE DB defines an extensible construct, the schema rowset, which providers can use to surface custom metadata. ADO naturally maps these OLE DB rowsets to Recordsets.

Table 1

	Numeric variable	Character variable
Name	This property is set to the variable name	
Type	adDouble	adChar
DefinedSize	8	The length of the variable

ADO's OpenSchema method returns schema rowsets; the COLUMNS schema rowset contains the same variable metadata found in the Fields collection plus SAS specific metadata. By default, the provider returns information for every column in each table in the OLE DB data source; when you only want column information for a specific table, you can apply a restriction filter. The following example echoes the name of each variable and any stored format name in the hypothetical EMPLOYEES data set to VisualBasic's Immediate window.

```
Dim obColumnsSchema as ADODB.Recordset

Set obColumnsSchema = _
   ObConnection.OpenSchema(adSchemaColumns, _
      Array(Empty, Empty, "HR.EMPLOYEES"))

While Not obColumnsSchema.EOF
   Debug.Print obColumnsSchema!COLUMN_NAME _
      & " " & obColumnsSchema!FORMAT_NAME
   obColumnsSchema.MoveNext
Wend

obColumnsSchema.Close
```

Omitting the second parameter to the OpenSchema method returns each column in every data set known to the server--a fairly time intensive operation. (For more information on the COLUMNS schema rowset see *The OLE DB Programmer's Reference*; for the complete list of custom columns the SAS/SHARE Data Provider includes on the COLUMNS schema rowset see the *ADO/OLE DB Cookbook.*)

BELLS AND WHISTLES

The previous sections show the basic elements of converting a SAS ODBC Driver application to an ADO application using the SAS/SHARE Data Provider. The paper also introduces how you can accomplish data access tasks common to most applications. For the most part this discussion has entailed fairly straightforward ADO constructs and mappings between ADO/OLE DB and SAS. There are a few SAS features the SAS/SHARE Data Provider supports that do not map so naturally to OLE DB and ADO. Some of these features are popular aspects of SAS you might require in your application, but their integration with ADO is not always intuitive. I mention them briefly here so that you can be aware of the peculiarities

FORMATS AND INFORMATS

The SAS/SHARE Data Provider supports standard SAS formats and informats. It is easy to ask the provider to return each row of data using each variable's stored or default formats. This is done through the custom SAS Formats property. The property accepts a special keyword value, "_ALL_", which tells the SAS/SHARE Data Provider to format all values using the format information stored in the data set if it exists for a variable. If there is no stored format information for the variable, the default SAS format for the variable's data type is used.

```
Set obRecordset.ActiveConnection = _
   ObConnection
obRecordset.Properties("SAS Formats") = _
   "_ALL_"
obRecordset.Open "HR.EMPLOYEES", _
   adOpenDynamic, adLockReadOnly, _
   adCmdTableDirect
```

Similarly, you can update data using informats by setting the custom SAS Informat Recordset property. Because of limitations in how ADO integrates with OLE DB, if a Recordset is opened to use informats, it also implicitly uses formats. The rules involving this limitation, how formats are coupled with informats in such cases and the syntax for overriding the stored/default format and informat information are outlined in the *ADO/OLE DB Cookbook for OLE DB Providers.*

MISSING VALUES

The SAS/SHARE Data Provider handles missing numeric values uniquely. While SAS supports special missing values for numeric data (so that it is possible for numeric missing values to be given different meanings), the SAS/SHARE Data Provider collapses all special missing values to a single NULL value. In other words, data sets containing special missing values may loose some of their semantics when read with the SAS/SHARE Data Provider.

The SAS/SHARE Data Provider surfaces numeric missing values in such a way that ADO assigns them the special VisualBasic Null value. If your code does not anticipate this possibility, certain operations generate exceptions. Null is not a valid expression in an arithmetic statement. For example, if you have a data set with numeric variables A and B, and either of these variables contains missing values, then the following code will fail:

```
Z = obRecordset!A + obRecordset!B
```

Fortunately, you can programmatically test for and trap missing values in your code to avoid such errors. Exactly how you handle the occurrence of a missing value depends on the requirements of your application. To refine our example, we'll assume missing values should be interpreted as zeros.

```
If IsNull(obRecordset!A) Then
   A = 0
Else
   A = obRecordset!A
End If
If IsNull(obRecordset!B) Then
   B = 0
Else
   B = obRecordset!B
End If

Z = A + B
```

To write a missing value out to a data set, simply set the corresponding Field's value to Null.

Missing character values are treated similarly to how SAS treats them. SAS defines missing character values as strings containing nothing but blanks. If an ADO character Field's value is a zero length string or contains only blanks, it is considered to be missing. (The *ADO/OLE DB Cookbook for OLE DB Providers* gives complete details of missing value processing with ADO.)

REMOTE SQL PASS THROUGH (RSPT)

SAS allows SQL queries to be passed to a third party DBMS (via a SAS/ACCESS engine) for evaluation. This feature is sometimes desirable for performance reasons and has been integrated into the SAS/SHARE Data Provider. Two custom Connection properties control Remote SQL Pass Through (RSPT) behavior:

- SAS SQL Engine
- SAS SQL Engine Options

Both of these custom properties are added to the Connection object's Properties collection after the Connection has been opened. Set the SAS SQL Engine property value to the name of the SAS/ACCESS engine you want to perform the RSPT services (i.e., the engine that will pass the queries to the third party DBMS). The SAS SQL Engine Options property value should be set to the specific SAS/ACCESS engine options needed to connect to the third party DBMS; the exact options and values you place in this string depend on the SAS/ACCESS engine used and the third party DBMS configuration.

Here are example property values that result in the server passing SQL statements to an Oracle DBMS called "sales".

```
ObConnection.Properties("SAS SQL Engine") = _
   "ORACLE"
obConnection.Properties("SAS SQL Engine
Options") = _
   "path=sales user=scott password=tiger"
```

DATA ENCRYPTION

If you have sensitive data that you want to protect when accessed by the SAS/SHARE Data Provider, consider adding SAS/SECURE Software to your server. This will give you various data encryption schemes for use when transmitting data to and from the server. If SAS/SECURE data encryption is enabled in the server, the server and the SAS/SHARE Data Provider will encrypt your data before each transmission. This happens automatically from the perspective of the ADO application; no Connection properties need to be set in order for encryption to happen.

SAS/SECURE Software integration is unique to the SAS/SHARE Data Provider; the SAS ODBC Driver does not implement this level of security support.

CAVEATS AND WARNINGS

To help you avoid a few headaches and disappointments here are a few items that can be useful:

1. **Avoid defaulting to optional parameter values.**

   Don't assume that the provider will honor the default values for optional method parameters. ADO does not consistently pass default information to the provider; it sometimes assumes the provider mirrors its default behavior. Also, there have been a few cases where we have had to change the default provider behavior from one release to the next when we discovered a shortcoming in our conformance to the OLE DB specification.

   If you explicitly give a value for all method parameters, you will avoid a number of seemingly inexplicable problems.

2. **Get ADO 2.5.**

   The syntax used here for setting custom Recordset properties is not possible in releases prior to 2.5. In early versions of ADO, you have to use a Command object in a round about way to set custom Recordset properties. If you must use one of these early releases, see the *ADO/OLE DB Cookbook for OLE DB Providers* regarding this issue.

3. **There is no SAS/SHARE Data Provider equivalent of PROC ODBCSERV.**

   The SAS/SHARE server must always be started before a SAS/SHARE Data Provider application can use it. The SAS/SHARE Data Provider does not have the ability to automatically start a local server like the SAS ODBC driver can.

4. **Accessing V6 SAS/SHARE servers is experimental.**

   The ability to run the SAS/SHARE Data Provider against V6 SAS/SHARE servers is documented but is not fully tested. Hot fixes are available from Tech Support to fix a few known fundamental problems. At a minimum you will need those fixes to open a Connection to a V6 server; at present the reliability of the SAS/SHARE Data Provider running against a V6 server is not guaranteed, but please report any problems you encounter.

CONCLUSION

The SAS/SHARE Data Provider occupies the same basic feature space as the SAS ODBC driver. Just as OLE DB is Microsoft's successor to ODBC, the SAS/SHARE Data Provider is SAS's successor to the SAS ODBC Driver. I encourage you to convert your SAS ODBC applications to SAS/SHARE Data Provider applications using ADO at your earliest convenience.

ADO is a very flexible set of objects. You can accomplish the same type of task many different ways. ADO gives you the same SQL processing abilities you are familiar with in the SAS ODBC driver. In addition, you have the ability to directly manipulate data sets and views when appropriate. This paper only focuses on some of the ways you can perform the most common data set operations.

In terms of the lifetimes of ODBC and OLE DB at SAS Intitute, we will continue to maintain the SAS ODBC driver until the marketplace completes the transition from ODBC to OLE DB and ADO. There are no plans to add new features to the SAS ODBC driver or enhance the level of ODBC support currently provided. On the other hand, the SAS/SHARE Data Provider will continue to be enhanced and refined. New SAS features will be integrated with the provider as quickly as possible.

REFERENCES

Microsoft Corporation (2000), *Microsoft ADO Programmer's Reference*, Redmond, WA: Microsoft Corporation

Microsoft Corporation (2000), *The OLE DB Programmer's Reference*, Redmond, WA: Microsoft Corporation

SAS Institute Inc. (2000), *ADO/OLE DB Cookbook for OLE DB Providers*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999), *SAS Language Reference: Dictionary*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999), *SAS ODBC User's Guide and Programmer's Reference*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999), *SAS/SHARE User's Guide*, Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

David Shamlin
SAS Institute Inc
Cary, NC 27513
Work Phone: 919-531-7755
Fax: 919-677-4444
Email: david.shamlin@sas.com
Web: www.sas.com