Paper 123-27

Using SAS® Data To Drive Microsoft Office

Darren Key and David Shamlin, SAS Institute, Cary, NC

ABSTRACT

Microsoft Office provides tools for integrating its suite of programs with other vendors' data sources. There are a variety of ways to do this using various ActiveX components and VBA scripting techniques. While the number of choices available can add to the complexity of creating a customized solution, the basics of integrating with SAS data sources can be simplified with some basic code patterns for migrating data and reports between Microsoft Office and SAS.

We will outline basic patterns for importing SAS data into Microsoft Office documents as well as exporting data back to SAS. The paper gives implementations of these patterns in the form of Visual Basic code that can be customized for site-specific usage scenarios. We will demonstrate the application of these techniques to tasks involving the migration of data between SAS data sets and Microsoft Office and Outlook applications.

The content of the material presented is geared toward application developers with basic experience using Visual Basic for Applications, Microsoft Office and familiarity with SAS data concepts. Source code and data are available for all applications and utilities demonstrated.

INTRODUCTION

Microsoft Office provides a familiar interface and set of tools for managing information. More and more members of today's work force are proficient with Microsoft Office and its abilities. Allowing this set of individuals to use Microsoft Office to complete routine tasks optimizes their performance, but often the information needed for these tasks is not directly part of a Microsoft Office document or needs to be migrated from an Office document to another data store.

Migration of data between Microsoft Office and other data repositories can often be cumbersome and complicated depending on the transport mechanism. Standard integration tools provided by Microsoft can be useful; but since they are not customized for a specific analytic vendor like SAS, they can require a greater degree of expertise on behalf of the user.

This paper outlines SAS and Microsoft Office tools available to IT developers that can be leveraged to make Microsoft Office tasks easier when integrating SAS data. We provide basic coding patterns that together form a toolkit for building Office applications around SAS data. Through some specific case studies, we demonstrate how custom interfaces and wizards can be created to allow an Office user to interoperate with SAS data without advanced SAS expertise.

To begin building our toolkit, we start with an overview of how Microsoft Office accommodates custom extensions and the hooks that are available for access data stores.

MICROSOFT OFFICE AND OLE AUTOMATION

OLE Automation is a Windows service for extending the features of an application or connecting one application to the features of another. This is accomplished through an OLE protocol that allows an application developer to publish custom features that are not part of the host application's standard user interface. IT developers can leverage these OLE tools to extend the original application or integrate it with the features of another application.

Office applications expose groups of related features through automation interfaces. An interface is a collection of functions (also called methods) and attributes (also called properties) that can be called from other application code. In one of our case studies, we use a special automation interface that allows Excel and Access to call custom data import and export wizards implemented as COM add-ins. The interface is called IDTExtensibility2 and contains five methods:

- OnAddInsUpdate
- OnBeginShutdown
- OnConnection
- OnDisconnetion
- OnStartupComplete

Each of these methods is called by the Office application (also called the host application) running the add-in during different points of usage of the application; these methods coordinate the state of the application (start up, shut down, etc.) with the state of the add-in. We will explore these methods in more detail later in the paper.

Along with OLE Automation interfaces, we make use of a special set of OLE objects in our toolkit for accessing SAS data sources: ActiveX Data Objects (ADO). ADO is a set of easy-to-use objects created by Microsoft to give developers access to tabular, hierarchical and stream data. Two ADO objects encapsulate the core functions of data access: the Connection object and the Recordset object. The Connection object maintains the physical connection to a data store while the Recordset object contains the results of a tabular data request and provides methods to navigate the rows of data and manipulate column values.

These objects can be used in many situations with a handful of code patterns (connect, open, read, update, etc). After introducing the SAS components that support ADO, we will look at these code patterns in detail with respect to SAS data.

SAS OLE DB PROVIDERS

OLE DB is a low-level Microsoft specification for data access. A component that implements the OLE DB interface is called a provider; ADO is implemented to consume OLE DB providers. Essentially, ADO is a simplification of OLE DB. Typically, a data storage vendor implements an OLE DB provider to efficiently map the OLE DB API to its internal data representation. By associating an ADO Connection object with an OLE DB provider, you can easily access the associated data source from any OLE automation environment.

SAS Institute currently supports three OLE DB providers:

- The SAS Local Data Provider
- The SAS/SHARE® Data Provider
- The SAS IOM Data Provider

While each of these providers gives you access to SAS data, they do so in unique ways. Choosing the provider that is right for a specific usage case depends on where the data lives with respect to the machine running the application and how the data will be accessed.

The SAS Local Data Provider (LDP) gives you the ability to read data sets created by the SAS System for Windows. Using the LDP you can directly input the contents of a SAS data set located

in a Windows directory; the aid of a running SAS session is not required. The SAS/SHARE Data Provider (SDP) allows you to access SAS data sets and views from a SAS/SHARE server. In addition to reading data set, you can also update data sets using this provider. The SAS IOM Data Provider (IDP) also gives you update access to SAS data sets. Using the IDP you can manipulate SAS files through a SAS/Integration Technologies IOM server. While the latter two providers (SDP and IDP) require SAS server sessions be available to the client application, they also support SQL query processing. The table below summarizes the features of the three providers.

Provider	Type of data	Type of access
LDP	SAS data sets created by SAS for Windows	Read only
SDP	• SAS data sets and views • SQL queries • SAS/ACCESS files	Read and update
IDP	• SAS data sets and views • SQL queries • SAS/ACCESS files	Read and update

Choosing the right provider(s) for your application is a matter of understanding the environment housing your SAS data and the way you intend to access the data.

You can use the requirements for your Office customization to identify the SAS OLEDB provider (or providers) that are right for your application. Knowing where the data lives (in a native file format on a networked disk or on a foreign host within the domain of a server), how data requests are going to be expressed (through simple table access or using SQL queries), and what kind of concurrency is needed (exclusive or shared access) will help you clarify which provider(s) to use. Then you will be ready to write ADO code that implements the required data manipulation within the Office application context. Fortunately, most ADO data access tasks are accomplished with a few basic programming idioms:

- Connecting to a data source
- Opening a data set
- Creating a data set
- Navigating the rows of a data set
- Manipulating column values

In the next sections we will refine these idioms into a set of simple Visual Basic code patterns you can use with the SAS OLE DB providers to write your applications. Each of these patterns can be applied in a variety of ways, and there are subtleties involved in each related to the exact set of method parameters used and object properties set. Addressing the details of individual usage scenarios are outside the scope of this paper. Fortunately, the Institute distributes an ADO/OLE DB Cookbook that documents the specifics of these providers and how they relate to ADO objects. Throughout the remaining discussion we point out situations where you should be aware of nuances in usage but don't necessarily provide all the related detail; we advise you to consult the ADO/OLE DB Cookbook when you need more information to implement a particular task.

ADO CODE PATTERNS

The Connection object, the Recordset object and the Field object are the basic building blocks of most ADO applications. The Connection object defines the OLE DB provider to use and maintains the connection to a data source. A Recordset object uses a Connection object to reference a particular table in the underlying data source. As we normally thinking of a table as being made up of rows and columns, a recordset is made up of rows and fields. Each row in the recordset maps to a row in the associate table, and each field maps to a column in the table. A Recordset object open on a table maintains a current row position. This position identifies the row in the underlying table that is currently being accessed. When the recordset is positioned on a valid row, the recordset's collection of Field objects contains the values for the corresponding columns in that row. The relationship between Recordsets, rows and Fields is shown in more detail in the reading and updating code patterns.

CONNECTING

The first step in any ADO process is opening a connection to the desired data source. This is done with a Connection object. When initializing a Connection object, you specify the OLE DB provider to use and the data source connection properties. The exact set of properties needed depends on the provider and the data set used.

```
Dim obConnection as ADODB.Connection

Set obConnection = New ADODB.Connection

obConnection.Provider = "provider name"
obConnection.Properties("connection
property") = "value"

obConnection.Open
```

For the LDP, the symbolic provider name to use with the Connection object's Provider property is "sas.LocalProvider". "sas.ShareProvider" is used with the SDP, and "sas.IOMProvider" is used with the IDP. The exact set of connection properties to use with each provider is beyond the scope of this paper, but the details are given in the *ADO/OLE DB Cookbook for the SAS Data Provider*.

OPENING AN EXISTING DATA SET

Gaining access to the contents of a data set requires an open Connection object and a Recordset object. To open a data set you must specify its name, an ADO cursor type, and an ADO lock type. The latter two pieces of information control the access mode used to open the underlying data set. The Recordset Open method is used to open the physical data set:

```
Dim obRecordset as ADODB.Recordset
Set obRecordset = New ADODB.Recordset
obRecordset.Open "table name",
                 CursorType,
                 LockType,
                 adCmdTableDirect
```

The form of the first parameter is the same for the SDP and IDP: *libname.memname*. This is the standard SAS way to identify a data set. For the LDP, the syntax is slightly different. This parameter can either be a simple data set member name (if the connection was configured to point to a specific Windows directory as its data source) or a fully qualified Windows file specification that references a physical SAS data set file.

The cursor type parameter either limits access to forward-only movement through the data set (adOpenForwardOnly) or allows random access (adOpenDynamic). The lock type parameter controls the type of concurrency employed when opening the data set. Control can be limited to read-only access or optimistic locking. Be aware that the type of locking available in any given case can be constrained by the provider being used, the attributes of SAS data set being accessed and the way it is being accessed via the provider.

CREATING A DATA SET

In some instances you may wish to create a new data set instead of opening an existing one. The standard ADO way of doing this is through the Catalog and Table objects found in Microsoft's ActiveX Data Objects Extensions for Data Definition Language and Security (ADOX) library. Use the Table object to define basic data set attributes and variable structure:

```
Dim obTable as New ADOX.Table

obTable.Name = "data set name"
obTable.Columns.Append "variable_name",
                       adDouble
obTable.Columns.Append "variable_name",
                       adChar,
                       length
```

The form of the data set name depends on the associated provider as mentioned in the previous section. The variable name must conform to the rules for SAS data set variable names. The SAS providers map the two SAS variable types (character and numeric) to the ADO data types adDouble and adChar, respectively; when specifying the adChar type, you will also need to specify the size (in characters) you want the variable to be. Unfortunately, there is no way to specify formats, informats, data set or variable labels using ADOX. (With the SDP and IDP you can work around this problem with SQL CREATE TABLE statements.)

After you have initialized a Table object to describe the SAS data set you want to create, use the ADOX Catalog object to turn that table definition into a SAS data set:

```
Dim obCatalog as New ADOX.Catalog
Set obCatalog.ActiveConnection = obConnection
ObCatalog.Tables.Append obTable
```

By calling the Append method on a Catalog object's Tables collection, you are directing ADOX to add the table defined by the Table object to the underlying data source. When done using SAS OLE DB providers, this results in a data set being physically added to the associated data source.

Now that we've introduced the basics of creating new data sets and opening existing ones, the next obvious tasks to address involve manipulating data within a data set.

NAVIGATING OBSERVATIONS

Once a Recordset object has been used to open a data set, the row position properties and navigational methods can be used to access observations in the data set. The SAS OLE DB providers map data set observations to rows in an ADO Recordset. The Recordset object provides a set of methods to navigate through its set of rows, and there is a basic set of coding idioms that allow you to implement any kind of scheme needed to iterate through a data set's observations.

There are two Recordset properties for testing when the current position is at the beginning or end of the data: BOF and EOF, respectively. These are commonly used to control loops that iterate sequentially through a data set, or check for boundary conditions when the code skips a set of observations between reads. These properties can also be used to test for empty Recordsets; if the following expression evaluates to true when the Recordset encapsulates a SAS data set then the data set has zero observations:

```
ObRecordset.BOF and obRecordset.EOF
```

You can also easily test if a Recordset object allows random positioning within the underlying table; the following expression evaluates to true when the Recordset object supports random positioning:

```
obRecordset.Supports(adMovePrevious)
```

The most straightforward usage case regarding navigating a data set is sequentially iterating through all observations. This is done with a simple WHILE loop.

```
obRecordset.MoveFirst
Do While Not obRecordset.EOF
     ' Process the current row, then…
     obRecordset.MoveNext
Loop
```

The SAS OLE DB providers support the following standard ADO methods for navigating through the data: MoveFirst, MoveLast, MoveNext, MovePrevious and Move. These five methods along with ADO's bookmarking features give you a full compliment of navigational abilities. Since the implementation of this functionality is standard across all SAS providers, you can learn more details about them in Microsoft's *ADO Programmer's Guide* and *ADO Programmer's Reference*.

READING, WRITING AND UPDATING OBSERVATIONS

Once you have the code in place that gives you access to the desired observation or observations in a data set, the next obvious step is manipulating variable values within those observations. As mentioned previously, using the navigational Rowset methods sets the *current position* within the Recordset. The current position in turn references an observation in the underlying table. The values of variables for the observation are surfaced through the Recordset's *Fields* collection; the elements of the Fields collection are Field objects. Field objects have Value properties. A Value property contains the associated variable's value for the current position observation. There are a variety of Visual Basic expressions that evaluate to the variable value. For the sake of example, let's assuming we want to access the value of the fourth variable in an observation; the variable's name is "myvar". The following expressions can be used interchangeably to evaluate this hypothetical variable's value for the current position observation:

```
obrecordset.Fields("myvar").Value
obRecordset.Fields(3).Value
obRecordset.Fields("myvar")
oRecordset.Fields(3)
obRecordset("myvar").Value
obRecordset(3).Value
obRecordset("myvar")
obRecordset(3)
obRecordset!myvar
```

While myvar is the fourth variable in the observation, it is indexed as third element of the Fields collection since Visual Basic collection indices are zero-based. The various ways of expressing a Recordset Field's value are all Visual Basic short hand expressions of the first two given. Any of these values can be used to get or set a variable's value. (I.e, they can appear on the left or right hand side of an assignment expression.)

```
x = obRecordset!myvar + 7
obRecordset("myvar") = x * 2
```

After executing a statement that alters the value of a Field, the new field value is stored in a buffer internal to ADO. You must call one of ADO's update methods (Update or UpdateBatch) to commit the change to the data source. If you change the Recordset's current position (i.e., move to another observation) before calling one of these update methods, the new values are lost.

Adding an observation to a data set involves creating a new record, setting Field values for that record and then committing the new Field values to the data set:

```
If obRecordset.Supports(adAddNew) Then
   obRecordset.AddNew
   obRecordset!variable_name = value
   obRecordset.Update
   fBatchMode =
        obRecordset.Supports(adUpdateBatch)
   If fBatchMode Then obRecordset.UpdateBatch
End If
```

This code template is generic ADO; there is nothing SAS specific about the pattern. It shows the basic code structure for adding a row of data to a table through and ADO Recordset object.

USING SQL PROCESSING

The above usage patterns focus on data access idioms using atomic ADO methods. It is possible to accomplish many of the same operations using SQL statements when the underlying OLE DB provider and/or data source support SQL; this is the case with both the SAS/SHARE Data Provider and the SAS IOM Data Provider. In some situations, using SQL statements can be more efficient and allow for better exploitation of SAS features. For example, using the SQL CREATE TABLE statement to create data sets is faster than the ADO Catalog object; it also allows you to include special SAS features (formats, informats, integrity constraints, etc) in the data set creation process. This support is possible because SAS SQL includes extensions for these constructs. In general you can use the features supported by PROC SQL through the SAS/SHARE and SAS IOM Data Providers.

Patterns for executing SQL statements with ADO fall into two categories. One is for statements that return result sets (i.e., SELECT statements), and the other is for statements that don't (i.e., everything else.)

Statements that return result sets should be associated with ADO Recordset objects. When used in this way, the recordset encapsulates the rows in the SELECT statement's result set. Assume you have a string variable strMyQuery that is set to your SQL query string. It can include any semantics supported by the SAS server you are connected to through your ADO Connection object. Given an open Connection object obConnection you can evaluate your query using the following ADO statement

```
ObRecordset.Open strMyQuery, obConnection,
            AdOpenForwardOnly,
            adLockReadOnly,
            adCmdText
```

This method call interprets the first parameter as an SQL statement; the final parameter value (adCmdText) dictates such behavior. In this example, we have asked for a sequential (i.e., forward only), read-only Recordset. SQL result sets returned by SAS OLE DB providers will always be read-only, but they are not always restricted to forward only access. The SAS IOM Data Provider supports random access with SQL result sets.

The second class of SQL statements is all the other statements beside the SELECT statement that do not return result sets. These statements can be executed using a Connection object or a Command object. Given an open Connection object obConnection, a statement like

```
obConnection.Execute "CREATE TABLE MYTABLE …"
```

executes the given CREATE TABLE statement using the server associated with obConnection. Alternatively, you can associate a Command object with an open Connection object:

```
Dim obCommand as New ADODB.Command

Set obCommand.ActiveConnection = obConnection

ObCommand.Execute "CREATE VIEW MYVIEW…"
obCommand.Execute "DROP TABLE MYTABLE"
```

The secret to keep in mind when choosing between these two different approaches for SQL statement evaluation is that the former implicitly instantiates an ADO Command object behind the scenes; for every Execute method call made on a Connection object, a Command object is created by ADO under the covers. The allocation and initialization of these objects is expensive and can quickly degrade performance if more than a few Execute method calls are made. Therefore, it is recommended that you explicitly create a command object and reuse it if your application is going to execute a series of SQL statements.

So far we have covered the basics of ADO usage with respect to SAS OLE DB providers. Some of the scenarios shown are very specific to usage with SAS providers; some are generic ADO code patterns. Of course, there are other scenarios that require greater exploitation of SAS features (e.g., formats, informats, SAS/ACCESS engine, etc.) These are covered in the *ADO/OLE DB Cookbook for the SAS Data Providers*; turn to this reference if your application requires more advanced processing than discussed here. In the remainder of the paper, we review some real world usage of these patterns as we have applied them to tools developed at SAS Institute.

CASE STUDY: A DATA SET IMPORT/EXPORT WIZARD

There are a variety of alternatives to migrating data from SAS data sets to Microsoft Office applications like Excel and Access. While each option provides a valid choice and has its own benefits, they also come with limitations. Common among them are the fact that the operation is initiated from a SAS process and requires specific SAS knowledge. Even in the cases where the operation can be initiated from inside the Microsoft application, it can require an advanced understanding of the SAS System that is often beyond the knowledge of the typical Microsoft Office user. Therefore, we created some wizards that integrate tightly with the Office paradigm and hide the low level domain knowledge that must be provided to access a SAS data set. We tried to focus the user experience on pointing to the data to migrate and mapping columns to variables as necessary.

Our import/export wizard is implemented as a COM add-in for Microsoft Office 2000; it supports both Excel and Access. Much of the user interface and implementation are shared across both applications. Once installed, it adds custom import and export items to the applications' menus. Selecting one of the menu items initiates a set of wizard dialogs that guide the user through choosing a provider, data source and table. Once the user has provided the input needed to complete the import or export operation, the wizard generates a simple ADO script to migrate the data. The wizard gives the user the opportunity to save this script code for reference or future use. This gives the user the ability to create a more streamlined process for data that is migrated between environments routinely.

Dividing the code into two conceptual parts helps understand how the dialogs and data migration aspects are implemented apart from how the functionality is integrated into the Microsoft Office applications. As stated above, the wizard functionality is made available through menu items. The add-in will add the menu items when the Office application is invoked and see that control is given to the wizard implementation when one of these menu items is chosen.

The wizard add-in is implemented as a Visual Basic Add-In project called SASOfficeWizard. (The source material for this project is available on the Base SAS community web site: http://www.sas.com/rnd/base.) A series of Visual Basic files comprise the SASOfficeWizard project. Knowing the basic purpose of these files will help us understand the architecture of the project. In the remainder of this section we give an overview of the project internals to allow the reader to navigate through its various modules and classes and understand their roles.

A set of forms implements the wizard dialogs. Each of the .frm Visual Basic files encapsulates a single dialog used by the wizard.

The AccessAddin.dsr and ExcelAddin.dsr designer files couple the wizard to the Microsoft Office applications. This is accomplished via Windows registry keys and the IDTextensibility2 interface. Most of this code is generated by the Visual Basic project wizard which keeps us as Office developers from having to know too much of the related detail. (We took liberties with the code the Visual Basic project wizard generates with respect to the designer modules. The project wizard only generates one designer module per add-in project; we cloned the original designer module to support Excel and Access from the same add-in.) When the add-in is built (or the runtime DLL is copied to a new system and self-registered with the system utility regsvr32.exe) the appropriate keys are added to the system registry. (For specific details on add-in registration, see Microsoft's "Developing COM Add-Ins for Microsoft Office 2000" technical article.) After the DLL is property registered, the associated Office applications (also referred to as the hosts) connect to the add-in when the application is invoked. This is done through methods on IDTExtensibility2 that the add-in's designer components implement

For our import/export wizard, we only have to implement two of the five IDTExtensibility2 methods: OnConnection and OnDisconnection. These methods

- Add the custom menu items to the host's menu bar during a host connection event
- Remove the custom menu items from the host's menu bar during a host disconnection event
- Associate the menu item click events with the appropriate wizard import and export functions

The code that passes control from the add-in framework to the custom wizard is found in the GUIManager class. This class implements the Click event methods for m_oImportButton and m_oExportButton objects. The code for these methods is extremely simple; it merely calls the import or export prompt method on the project's Wizard class. The Wizard class implements the custom SAS dialogs and data migration operations that are the heart of our add-in.

There are five class modules. The simple AppInformation class manages the basic information needed to run the host application. The GUIManager class controls the user interface extensions the wizard makes to the host application interface. The OLEDBInformation and OLEDBProperty classes encapsulate the information the wizard collects from the user to connect to the SAS data source. Finally, the Wizard class ties all these pieces together.

Finally, there are three modules. Globals.bas declares global data storage and methods shared by the rest of the code. The remaining two modules (AccessMethods.bas and ExcelMethods.bas) manage host application specifics and use ADO objects to migrate data to and from SAS data stores. In each module there are a set of methods that access host application information needed to export data:

- To identify the set of rows and columns to export
  - AccessGetSelections
  - ExcelGetSelections
  - AccessSetSelection
  - ExcelSetSelection
- To get metadata about the columns to export
  - AccessGetSelectedColumnNames
  - ExcelGetSelectedColumnNames
  - AccessGetSelectedColumnTypes
  - ExcelGetSelectedColumnTypes
  - AccessGetSelectedColumnLengths
  - ExcelGetSelectedColumnLengths
- To determine the host application state
  - AccessGetFileName
  - AccessGetFileName
  - AccessHasOpenFile
  - ExcelHasOpenFile
- To copy data between the host application and SAS
  - AccessImportData
  - ExcelImportData
  - AccessExportData
  - ExcelImportData

The final set of methods is the most interesting with respect manipulating SAS data from an Office application. These methods apply the ADO code patterns discussed in previous sections to perform the work. Unfortunately, at the time of this writing the Visual Basic code that makes up the SASOfficeWizard is not frozen, so we are hesitant to discuss more specific detail here; by SUGI 2002 a complete copy of the Visual Basic project will be available to supplement this paper; with the source code we will provide additional documentation as needed.

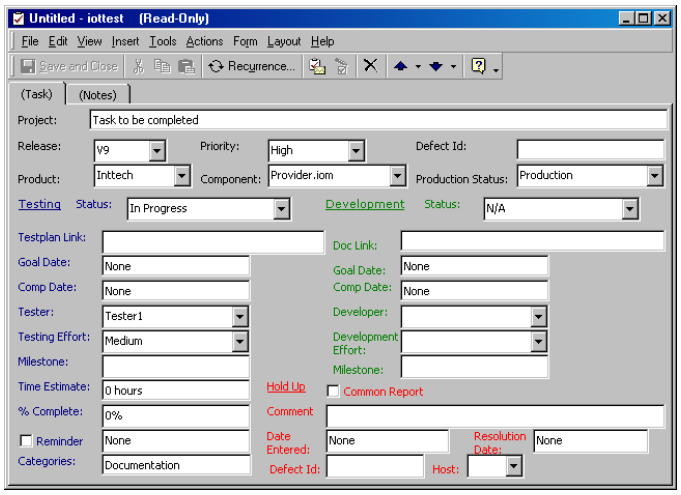
CASE STUDY: AN OUTLOOK TASK DATA EXTRACTOR

Project management is a task that every organization must deal with. It can often be difficult to find applications that provide the correct amount of detail so that each person knows exactly what their tasks and deadlines are, while at the same time providing good overviews and reports so that the progress of the project can be tracked efficiently. The Outlook public folders are an example of an application that handles the details of individual tasks very well. A set of public folders can be created that allow managers and employees to add and edit tasks related to a project. While this does a good job of letting each person know what their tasks are and when they are due, it is difficult for anyone to get a quick overview of how a project or set of projects is progressing. Using Visual Basic and the ADO code patterns discussed above it is possible to quickly develop an application that generates a SAS data set from the tasks in an Outlook public folder. This data set can then be used to generate the higher level reports that give perspective over the projects.

The following snippets cover the major coding tasks involved in creating an application that generates a SAS data set from a set of Outlook public folders containing task items. Unlike the Import/Export Wizard case study, the code will be distributed as code snippets in this document rather than as a ready-to-run source code. These snippets will illustrate the application of the ADO code patterns to this new problem.

The application is not an Add-in like the Import/Export Wizard. Instead, it is a standalone executable that connects to Outlook from the outside an Outlook session rather than being launched from a menu item. The first step is to set up the public folders in Outlook. Pick or create the folder that will contain all the sub folders and task items. This example will be using "/Public Folders/All Public Folders/SAS Institute/Dept/BSR/Testing/IOT/Projects". Once a public folder is

chosen as the root then a subfolder is created for each project to be tracked. In each project folder a "Tasks" subfolder is created. This allows other folders for ancillary project information to be created without interfering with the task list. The user-defined task form shown in the figure below is used as the default form for each "Tasks" folder created.



This form was created from the default Task form and named "IPM.Task.iottest". Values on this form are associated with Fields by

1. Right clicking on them in the form designer
2. Choosing "Properties" from the pop-up menu
3. Clicking on the "Values" tab
4. Choosing or creating a field name

These field names will be used later to retrieve the values on the form. The form designer can be accessed from the "Tools->Forms->Design A Form…" menu in Outlook.

Outlook is now ready to be used to track the tasks for each project. Managers and employees have a central place where tasks can be viewed, created, and modified as the need arises. The rest of this application is written in Visual Basic.

Start Visual Basic and open a Standard EXE project. Delete the Form and add a Module. From the "Project->Properties" menu on the "General" tab, choose "Sub Main" as the Startup Object. Note that in a full fledged application the Form should probably be used to provide progress information as well as configurations options. However, that is beyond the scope of this sample. From the "Project->References…" dialog add the following references to the ones already in the list:

- Microsoft ActiveX Data Objects 2.5 Library
- Microsoft CDO 1.21 Library

This Visual Basic project is now ready for code.

The first coding step is to set up some global configurations variables. These variables will be used to tell the application where to get the task information, which task information to ignore, and where to store the data set.

```
Option Explicit
Dim RootFolderPath As String
Dim OutputPath As String
Dim ProviderName As String
Dim Location As String
Dim Data_Source As String
Dim ExcludedFolders As Variant
Dim ExcludedCategories As Variant
```

RootFolderPath is the full path to the root folder chosen to contain all the task information. OutputPath is where the data set will be stored. The format of this path will depend on the OLE DB Provider used. This sample uses the SAS/SHARE Provider so the path is of the form "libname.member" where the libname is a libref already defined on the server. ProviderName, Location, and Data_Source all serve to identify where the data set is going to be stored. ProviderName for this sample is "SAS.SHAREProvider". Location is the IP address of the machine hosting the SAS/SHARE server and Data_Source is the server id of the SAS/SHARE server running on that machine. See the *ADO/OLE DB Cookbook for the SAS Data Providers* for more information on setting up a libname on a SAS/SHARE server and for connecting to other providers. In order to limit the amount of time spent in uninteresting folders there is also a list of folder names that will never be traversed as well as a list of categories that will never be added to the data set.

We begin initializing the application in the main subroutine by setting of the global configuration values.

```
RootFolderPath = "Public Folders/…"
OutputPath = "Taskdata.Taskinfo_" &_
   Format(Now(), "yyyy_mm_dd_hh_mm_ss")
ProviderName = "SAS.SHAREProvider"
Location = "localhost"
Data_Source = "shr1"
ExcludedFolders = Array("Template",_
                        "Completed",_
                        "Deferred")
ExcludedCategories = Array("Completed",_
                           "Deferred",_
                           "Cancelled")
```

A full-fledged application could store this information in the registry and provide a configuration dialog for the user to modify these values but, again, that is beyond the scope of this sample. The application must now find the task information in Outlook. This involves opening a MAPI session, logging on, and finding the root folder.

```
Dim obSes As New MAPI.Session
Dim obFolder As MAPI.Folder
obSes.Logon "My Profile Name"
Set obFolder = GetFolder(obSes, _
                         RootFolderPath)
If obFolder Is Nothing Then Exit Sub
```

GetFolder is a custom function that takes a logged on MAPI Session and a string representing a path to a folder. The path is parsed and the folder it represents is retrieved from the MAPI Session.

Once the folder has been found, the destination data set must be created. First the "Connecting" pattern is used to open a connection to the SAS/SHARE server.

```
Dim obCon As New ADODB.Connection
Dim obRec As New ADODB.Recordset

obCon.Provider = ProviderName
obCon.Properties("Location") = Location
obCon.Properties("Data Source")=Data_Source
obCon.Open
```

The "Using SQL Processing" pattern is then applied to pass an SQL CREATE TABLE statement to the server. The specific statement used to create the data set is

```
CREATE TABLE ?
     (Project char(255),
      Category char(255),
```

```
        Subject char(255),
        Release char(255),
        Defect char(255),
        Product char(255),
        Component char(255),
        Production_Status char(255),
        Testing_Status char(255),
        Testing_Effort char(255),
        Testplan_Link char(255),
        Tester_Goal_Date char(255),
        Tester_Comp_Date char(255),
        Tester char(255),
        Tester_Milestone char(255),
        Development_Status char(255),
        Development_Doc_Link char(255),
        Developer_Goal_Date char(255),
        Developer_Comp_Date char(255),
        Developer char(255),
        Development_Effort char(255),
        Developer_Milestone char(255),
        Time_Estimate char(255),
        Percent_Complete char(255),
        Reminder char(255),
        Reminder_Date char(255),
        Holdup_Common_Report char(255),
        Holdup_Comment char(255),
        Holdup_Host char(255),
        Holdup_Date char(255),
        Holdup_Defect char(255),
        Holdup_Resolution_Date char(255))
```

In this SQL statement, the question mark is substituted with the data set name represented by the OutputPath global variable. The fully formed CREATE TABLE statement, stored in a local variable called CreateTableStmt, is used with the ADO Execute method and the resulting data set is then opened as an ADO Recordset object:

```
obCon.Execute CreateTableStmt
obRec.Open OutputPath, obCon, _
           adOpenDynamic, _
           adLockOptimistic, _
           adCmdTableDirect
```

The only things left to do in the main subroutine are walk the folder tree adding a row to the data set for each task and cleaning up. The tree traversal is managed by the TraverseFolders subroutine. This routine takes two input parameters: obFolder (the MAPI folder opened when we logged into Outlook) and obRec (the Recordset we're populating with task data). The following code recursively traverses the folder tree and calls AddRowsForTasks to add the next project's tasks to the data set.

```
Dim obTmpFolder As MAPI.Folder
If SkipFolder(obFolder.Name) Then Exit Sub
If obFolder.Name = "Tasks" Then
    AddRowsForTasks obFolder, obRec
End If
For Each obTmpFolder In obFolder.Folders
    TraverseFolders obTmpFolder, obRec
Next
```

The TraversFolders Sub exits if the given folder's name is found in the global ExcludedFolders array. If the folder's name is "Tasks" then that folder and the open Recordset are passed to AddRowsForTasks. Each subfolder of the current folder is then passed to TraverseFolders causing each folder in the tree to be visited.

The AddRowsForTasks subroutine takes the same parameters as TraverseFolders. The responsibility of AddRowsForTasks is to iterate through all the messages of type "IPM.Task.iottest" (the name of the user-defined task form). For each message found whose category is not in the global ExcludeCategories array, the detailed task information stored in the message's Fields collection is copied into the corresponding Recordset's Fields' collection. (The Category, Project, and Subject properties are not found in the Fields collection and require the use of special syntax to get the values.) These updated fields are then committed to the SAS data set via the Recordset Update method.

```
Dim msg As MAPI.Message
Dim Categories() As Variant
For Each msg In TaskFolder.Messages
   If msg.Type = "IPM.Task.iottest" Then
      Categories = msg.Categories
      If SkipCategory(Categories) Then _
           Exit Sub
      obRec.AddNew
      On Error Resume Next
      obRec!Project = _
         TaskFolder.Parent.Parent.Name
      With msg
         obRec!Category = Categories()(0)
         obRec!Subject = Subject
         obRec!Release = Fields("Release")
         obRec!Defect = Fields("Defect")
         obRec!Product = Fields("Product")
         obRec!ComponenT = _
            Fields("Component")
         obRec!Production_Status = _
            Fields("Production Status")
         obRec!Testing_Status = _
            Fields("Testing Status")
         obRec!Testing_Effort = _
            Fields("Testing Effort")
         obRec!Testplan_Link = _
            Fields("Testplan Link")
         obRec!Tester_Goal_Date = _
            Fields("Tester Goal Date")
         obRec!Tester_Comp_Date = _
            Fields("Tester Comp Date")
         obRec!Tester = Fields("Tester")
         obRec!Tester_Milestone = _
            Fields("Tester Milestone")
         obRec!Development_Status = _
            Fields("Development Status")
         obRec!Development_Doc_Link = _
            Fields("Development Doc Link")
         obRec!Developer_Goal_Date = _
            Fields("Developer Goal Date")
         obRec!Developer_Comp_Date = _
            Fields("Developer Comp Date")
         obRec!Developer = _
            Fields("Developer")
         obRec!Development_Effort = _
            Fields("Development Effort")
         obRec!Developer_Milestone = _
            Fields("Developer Milestone")
         obRec!Time_Estimate = _
            Fields("Time Estimate")
         obRec!Percent_Complete = _
            Fields("PercentComplete")
         obRec!Reminder = Fields("Reminder1")
         obRec!Reminder_Date = _
            Fields("Reminder Date")
         obRec!Holdup_Common_Report = _
            Fields("Holdup Common Report")
         obRec!Holdup_Comment = _
            Fields("Status Comment")
         obRec!Holdup_Host = _
            Fields("Holdup Host")
         obRec!Holdup_Date = _
            Fields("Holdup Date")
```

```
            obRec!Holdup_Defect = _
               Fields("Holdup Defect")
            obRec!Holdup_Resolution_Date = _
               Fields("Holdup Resolution Date")
            obRec.Update
         End With
         On Error GoTo 0
      End If
```

The call to TraverseFolders (which in turn calls AddRowsForTask) is done in Main. It is the heart of this subroutine, and is followed by code the cleans up before the program ends:

```
TraverseFolders obFolder, obRec

obRec.Close
obCon.Close
obSes.Logoff
```

The simple functions SkipFolder and SkipCategory are implemented for convenience and simply look for the passed folder name or category array in the global exclusion arrays. If they are found then True is returned, otherwise False is returned.

The only step left to complete this project management application is to use the power of The SAS System to analyze the newly created data set. This is left as an exercise for the reader but the possibilities range from htmsql pages that query the live data set to batch reports that are run each night and email those people with overdue tasks.

CONCLUSION

We have shown the basic patterns for accessing SAS data using ADO and ways ADO allows you to integrate SAS data with Microsoft Office applications. The material is presented as a top-level introduction to these tools as opposed to a detailed "how to" type tutorial. The discussion on ADO usage patterns introduces you to basic ADO concepts and how they are applied with the SAS OLE DB providers. The case study discussions introduce ways the usage patterns can be applied. In addition to describing the applications and Office integration tricks done, we give the structure of each case studies' code to help you dissect it further if you chose to explore the code in detail. In doing so you will definitely gain a deeper understanding of using ADO to integrate Microsoft Office with SAS as well possibly finding some code fragments you can reuse in your own applications.

Before beginning your own coding project, consider supplementing this paper with other reference material. Mastery of Microsoft ADO basics is necessary for successful application building; the *ADO Programmer's Guide* is an excellent starting place for beginners. The *ADO/OLE DB Cookbook* applies the fundamentals of ADO programming to the SAS OLE DB providers and is the next logical reference to study. Together these publications give you the foundation for creating ADO applications using SAS data sources. There are several MSDN articles related to Microsoft Office programming which will give you further context for how to automate the integration of SAS data with Office documents. Consider this paper the basic road map for creating these kinds of applications; use the other resources listed below to complete your reference set.

The tools and samples presented above show the basic approach to developing such custom applications. We're confident that you can take greater advantage of SAS data with Microsoft Office by writing even the simplest kinds of extensions. Our experience has shown that once the basic ADO idioms are mastered creating useful applications is easy. With just a small investment learning the rich integration hooks available, you will soon be on your way to enhancing you Microsoft Office environment with the power of SAS data!

REFERENCES

Microsoft Corporation (1998-2001), *Microsoft ActiveX Data Objects (ADO) Programmer's Guide*, Redmond, WA: Microsoft Corporation.

Microsoft Corporation (1998-2001), *Microsoft ActiveX Data Objects (ADO) Programmer's Reference*, Redmond, WA: Microsoft Corporation.

SAS Institute Inc. (2001), *ADO/OLE DB Cookbook for the SAS Data Providers*, Cary, NC: SAS Institute Inc.

SAS Institute Inc. (1999), *SAS Procedures Guide*, Cary, NC: SAS Institute Inc.

Schultz, Ed (1999), "Developing COM Add-Ins for Microsoft Office 2000," *Microsoft Developer Network*, http://msdn.microsoft.com/library/.

CONTACT INFORMATION

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

Darren Key and David Shamlin
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Work Phone: 919-677-8000
Fax: 919-677-4444
Email: Darren.Key@sas.com
David.Shamlin@sas.com
Web: http://www.sas.com/