Controlling Desktop Applications from Micro Focus COBOL Using Automation

Abstract

This paper shows how the powerful support for the Microsoft Component Object Model (COM) in Micro Focus Net Express® enables you to integrate the functionality of common desktop applications into your own development. By reusing the existing functionality of applications that your users may already have on their desktops, you can reduce the amount of development effort needed to implement new functionality, as well as bring enhancements to your users faster.

Contents

Introduction

How many times have you developed an application and had to implement functionality similar to that provided by existing commonly available desktop applications? For example, you may have had to develop an application that generates an attractive looking report on Windows. You may have coded this yourself by making calls to various Windows API functions or COBOL run-time routines. Or you may have purchased a separate third-party package that provides calls you can make from your COBOL application. However, there is an alternative if you know your users will have Microsoft Word installed on their systems. You can simply make calls from your COBOL application to Word to create the report and print it for you.

The technology that makes this possible is called *Automation*. Automation is a technology based on Microsoft's Component Object Model (COM). Automation enables an application to expose its functionality so that it can be utilized by other applications. This means that parts of off-the-shelf packages can be used, in conjunction with custom software, to create new applications. All of the applications in the Microsoft Office suite expose their functionality via Automation, allowing parts of these applications to be reused by your applications to perform common functions. Other Microsoft technologies also support Automation.

When people think of using technologies such as Automati on, they usually think of using programming languages such as Microsoft Visual Basic or C++. It is rare to find someone who immediately thinks of COBOL. However, Micro Focus has supported the use of Automation from COBOL for many years and, as this paper will show, COBOL can be a very appropriate language for creating business applications and utilizing functionality that has been exposed through Automation. Micro Focus Net Express 3.1 provides all the support you need.

Automation Clients and Servers

When we talk about Automation, applications can be Automation *clients*, Automation *servers* or both.

An Automation server is an application that exposes its functionality so that it can be "controlled" by other applications. Examples of Automation servers are the Microsoft Office applications such as Word, Excel, PowerPoint, as well as applications such as Microsoft Visio and the application used for administration of Microsoft BackOffice servers.

An Automation client is an application or programming language that controls other Automation servers by accessing the functionality exposed by those servers. These include Microsoft Visual Basic, Microsoft Visual C++ and, of course, Micro Focus COBOL as supported by Net Express.

This paper will focus on COBOL as an Automation client. For information on creating Automation servers in COBOL, see the paper "Developing Mixed Visual Basic/COBOL Applications" at http://www.cobolportal.com/resources.

A Note on Terminology

Over the years, Microsoft has introduced many different names for their component-based technologies and you will often see different names referring to the same technology. Because Micro Focus Net Express has supported these technologies for many years, you may see areas where the older terms are used. This section introduces the different terminologies, but you do not a detailed understanding of these terms to read and understand this paper.

Underpinning it all is the Microsoft Component Object Model (COM). A COM component is simply an object that exposes specific interfaces that enable an application to query the capabilities of the component and use it. All of the other technologies are based on COM components.

There are two primary mechanisms used for accessing COM components – the *vtable* mechanism and the *Dispatch* mechanism.

- A Vtable is the lowest level form of interface in to a COM component. A vtable is basically a record structure containing a number of pointers to different functions in the component. The COM specification lays out the format of these record structures. Vtables are ideally suited for languages such as C++ where pointers to functions are extensively used, but are not so useful for higher level languages such as COBOL and Visual Basic. For these languages, the Dispatch mechanism was introduced.

- The Dispatch mechanism enables a programmer to call a function in the component by name, rather than finding a pointer to the function and calling that function.

All components that support the dispatch mechanism also support the vtable mechanism for accessing the component. However, the reverse is not true. Not all COM components support Dispatch interfaces. Increasingly though, writers of COM components are being encouraged to support both mechanisms to enable their components to be used from the widest possible range of programming languages. This is important for COBOL, since COBOL only supports the Dispatch mechanism for accessing a component. If the component can be accessed from Visual Basic, you can be reasonably confident that you will be able to use it from COBOL.

Applications that provide a Dispatch interface were initially called *OLE Automation* servers. OLE stands for Object Linking and Embedding. It was the original name used by Microsoft for their component technologies and you will see the term OLE Automation used in Net Express. Now such components are simply referred to as Automation Servers. You will also sometimes see them referred to as ActiveX servers. The term *Automation Server* will be used for the rest of this paper.

Why Use the Object Oriented Syntax?

When you look at the examples later in this paper, you will see that Micro Focus has made use of the new object oriented (OO) syntax that has been recently added to COBOL to implement support for Automation. Even if you are not already familiar with these new additions to COBOL, you'll see that it is very easy to use these powerful extensions to the language.

So, why do you need to use the OO syntax at all? The reason is that the functionality of an Automation Server is exposed through an object interface. Each function is accessed using a combination of a reference to the object being used and the name of the function or property being accessed. This would not be possible using the standard COBOL CALL syntax. Instead, the new COBOL verb *invoke* is used, together with the new data type, *object reference*.

The key point is that, although we are using the new syntax, most programs that use Automation are unlikely to be object oriented programs. As you will see from the examples in this paper, you can still use standard procedural programming techniques, even though you are using the OO syntax to access Automation Servers.

A Simple Example Using Microsoft Word

The best way to get started with Automation is to look at a simple example. This example will use Microsoft Word to create a new document, insert some text, make some of the inserted text bold and then save the document.

```
 $set ooctrl(+P)

  class-control.
      MicrosoftWord is class "$OLE$Word.Application".

  working-storage section.

  78 Automation-True      value 1.
  78 Automation-False     value 0.

  01 Word                 object reference.
  01 Documents            object reference.
  01 Document             object reference.
  01 TextRange            object reference.
  01 BoldRange            object reference.

  01 StartBoldPoint       pic 9(8) comp-5.
  01 EndBoldPoint         pic 9(8) comp-5.

  01 Example-Text         pic x(43) value
     z"This is an example document created using ".

  procedure division.
 *    Startup Microsoft Word
      invoke MicrosoftWord "new" returning Word
 *    Make Word visible so that we can see what is
 *    happening
      invoke Word "setVisible"
          using by value Automation-True
 *    Get the collection of documents
      invoke Word "getDocuments" returning Documents
 *    Add a new document
      invoke Documents "Add" returning Document
 *    Get a range object for the entire document
      invoke Document "Range" returning TextRange
 *    Insert some text. This will extend the range object
      invoke TextRange "InsertAfter" using Example-Text
 *    Store the end of the range
      invoke TextRange "getEnd" returning StartBoldPoint
 *    Insert some more text
      invoke TextRange "InsertAfter" using
          z"Micro Focus COBOL and Microsoft Word."
 *    Get the end point of the range
      invoke TextRange "getEnd" returning EndBoldPoint
*    Finish with this range object
      invoke TextRange "Finalize" returning TextRange
 *    Adjust the starting point because this would have
 *    included the end-of-paragraph marker
      subtract 1 from StartBoldPoint
 *    Create a new range object that just refers to the
 *    text "Micro Focus COBOL and Microsoft Word"
```

```
           invoke Document "Range"
               using by value StartBoldPoint
                     by value EndBoldPoint
               returning TextRange
       *       Make this text bold
           invoke TextRange "setBold"
               using by value Automation-True
       *       Now save the document
           invoke Document "SaveAs" using z"Example-Document"
       *       Finalize all of the objects
           invoke TextRange "Finalize" returning TextRange
           invoke Document "Finalize" returning Document
           invoke Documents "Finalize" returning Documents
       *       Close Microsoft Word and finalize the Word object
           invoke Word "Quit"
           invoke Word "Finalize" returning Word
           stop run.
```

Figure 1 shows this program being animated using Net Express.

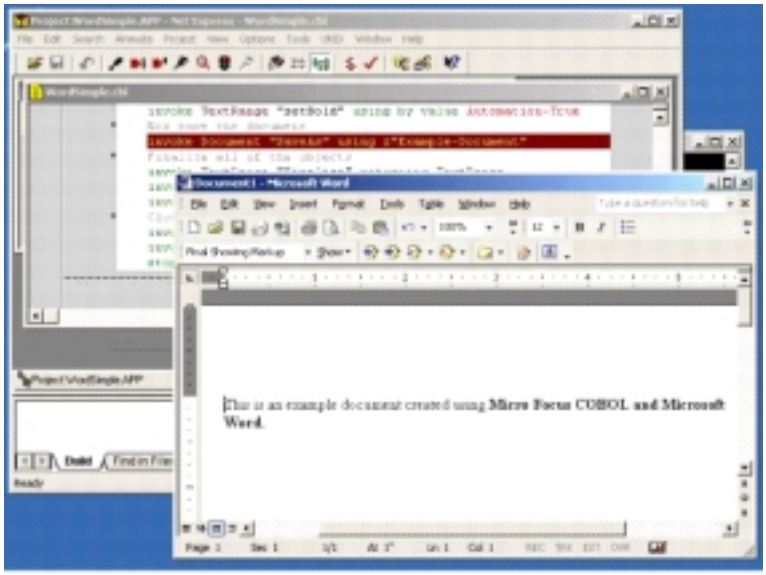

Figure 1 – Animating the Program using Net Express

Note. All of the examples in this paper have been tested using Microsoft Office XP. Most of them should work with Microsoft Office 2000 and Microsoft Office 97, but there have been some changes in the way the Office applications handle Automation over the years, so you should test with the versions of Office you will be expecting your users to use.

Basic Concepts

The previous example shows many of the concepts involved when using Automation. The following section will go through this program in detail.

The ooctrl(+P) directive

The first thing you will see in the program is the line:

```
$set ooctrl(+P)
```

This causes the **ooctrl(+P)** directive to be used when the program is compiled. This must be specified for all programs that use Automation. It ensures that the data type of each parameter in an invoke statement is available to the run-time. This is required to ensure that any conversion between data types is carried out correctly.

The CLASS-CONTROL Section

The next new feature in the program is shown in the lines:

```
class-control.
    MicrosoftWord is class "$OLE$Word.Application".
```

This is used to identify the Automation Servers that will be used by the application (if you were writing an OO program, this section would also identify the COBOL classes used by the application, but we don't need to worry about that in this program).

The Automation Server we are using is *Word.Application*. The name to use here will be specified in the documentation supplied with the Automation Server you are using.

The use of *OLE* in the name is used to notify the COBOL Run-Time System that the class being used is an Automation Server (Automation Servers used to be called OLE Automation Servers, hence the use of the word 'OLE'). If this is not specified, the COBOL Run-Time System will search for a COBOL class of the specified name, so OLE should always precede the name of the Automation Server you are using.

MicrosoftWord is the name that will be used for the class in the COBOL program. The use of this is seen later in the example.

The Object Reference

Before we can use any Automation Server, we will need one or more data items declared as type object reference. For example:

```
01 Word                  object reference.
```

This variable is used to hold a reference to an object in the Automation Server. There are two ways to get a reference to an object, either by using the "new" method or by it being returned by a method call to another object. Both types are seen in this example.

Creating a New Instance of an Automation Server

The first line in the Procedure Division is:

```
invoke MicrosoftWord "new" returning Word
```

This is used to create a new *instance* of Microsoft Word. This means that a new copy of Word is started and a reference to it is returned in the variable *Word*.

The *New* command (or *method*) is the only time you will use the name declared in the Class-Control section. From now on, you will use the reference returned in the variable **Word**.

Sending Commands to Word

Once you have a reference to an instance of the Automation Server you want to use, you can use that reference to send commands to it. The remainder of the program is a series of commands to Word to create a new document and insert some text. Many of these commands involve getting references to other objects. We will look at the techniques used in this section in more detail later.

Finalize the Objects

Finally, once you have completed using an object, you should ensure that any memory used by that object is released by using the Finalize method. In the example, this is done for the Word object using the following line:

```
invoke Word "Finalize" returning Word
```

To avoid memory leaks, you should ensure that you finalize every object you use.

Summary of the Basic Structure of an Automation Client

This example demonstrates the basic concepts of using COBOL as an Automation Client. The steps common to all programs are:

- Use the compiler directive ooctrl(+P).
- Define a Class-Control section that includes the Automation Servers you will be accessing.
- Declare at least one variable of type object reference to be used when sending commands to the Automation Server.
- Use the 'New' method to create a new instance of the Automation Server. This returns a reference to the object.
- Send commands to the Automation Server using the object reference returned by the 'New' method.
- Finalize all objects before terminating the application.

Object Models

To use most Automation Servers, including all of the Automation Servers in Microsoft Office, you must understand their *object models*. An object model is a representation of an application's functionality in terms of objects. An application has many different objects that are organized into various levels. These can be thought of as tiers in a hierarchy. The topmost tier is usually occupied by an object that represents the main application – the *Application* object. The second tier consists of a high-level categorization of objects. Lower tiers include additional objects used to access functionality that the second tier objects contain. Your application traverses the tiers to find the object you want to use.

Figure 2 shows the parts of the Word object model that are used by the previous example.

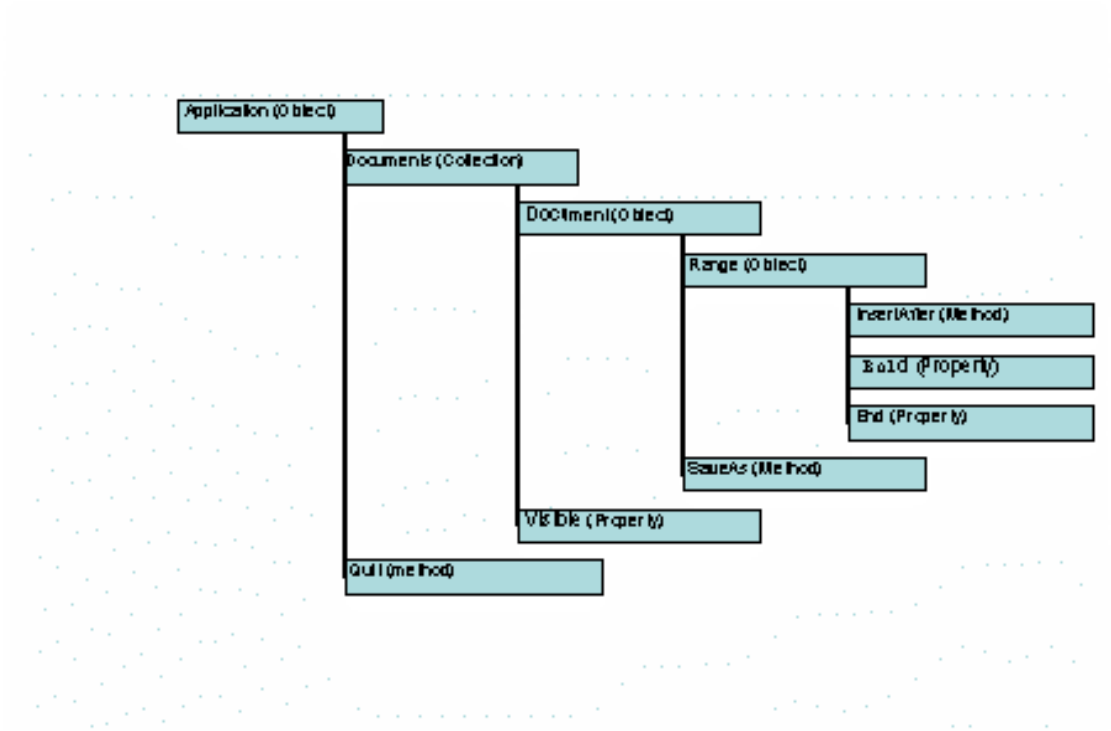

Figure 2 – The Parts of the Word Object Model used by this Example

The complete object model for Microsoft Word can be found at: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/off2000/html/wotocObjectModelApplication.asp)

Note. You may see some examples of using Microsoft Word via Automation that use a class called "Word.Basic". This is an earlier, simpler Automation model used by Word. Although Word.Basic is still supported by Word today, it is recommended that you use Word.Application for new applications.

You can see that different terms are used: Collections, Methods and Properties. These terms are described in the following sections.

Methods and Properties

Objects have *methods* and *properties*. Methods are actions that an object can perform. For example, in the previous program, we used the method **InsertAfter** to insert a piece of text in to the document. Properties are functions that access information about the state of something in the Automation server. For example, the property **Visible** on the application object indicates whether Word is visible or not.

In Micro Focus COBOL, both methods and properties are accessed using the COBOL *invoke* statement. To distinguish between them, to set a property, you prefix the property name with "set" and to retrieve the value of a property, you prefix the name with "get".

In the earlier example, you saw that the **Visible** property was set on the Application object, using the following line:

```
invoke Word "setVisible"
    using by value Automation-True
```

You might be wondering how you call a method if the method name begins with "set" or "get". To do this, you need to override the default behavior of the invoke verb. We will see how this is done in a later section.

Note. All of the examples in this paper set the Visible property for the application to true so that you can see what is happening. If you do not set the Visible property to true in Microsoft Office applications, they will run hidden.

Collections

In many applications, objects are grouped into collections. In the previous example, we retrieved the collection of documents currently loaded into Word using the following line:

```
invoke Word "getDocuments" returning Documents
```

Note that "Documents" is prefixed by the word "get". This is because collections are treated by most Automation Servers as properties of the object that contains the collection, so we have to prefix the name with "get" as described in the previous section.

We create a new document by using the **Add** method on the collection of documents:

```
invoke Documents "Add" returning Document
```

To access a particular element in a collection, use the **Item** property and specify the number of the element for which you want to obtain a reference. For example, to obtain a reference to the second document in a collection, you would use:

```
invoke Documents "getItem"
    using by value 2 returning Document
```

To determine the number of items in a collection, use the **Count** property. For example:

```
01 DocumentCount              pic 9(4) comp-5.

invoke Documents "getCount" returning DocumentCount
```

By using the **item** and **count** properties, you can loop through a collection. For example, to loop through the collection of documents, you could use:

```
01 DocumentCount              pic 9(4) comp-5.
01 CurrentDocument            pic 9(4) comp-5.
```

```
invoke Documents "getCount” returning DocumentCount
move 1 to CurrentDocument
perform until CurrentDocument > DocumentCount
    invoke Documents “getItem”
        using CurrentDocument
        returning Document

    *> Perform whatever functions are needed on the
    *> document and then finalize the reference
    *> to the document

    invoke Document “Finalize” returning Document
    add 1 to CurrentDocument
end-perform
```

Basic Data Types used with Automation Servers

When using Automation servers, you are limited in the types of data you can pass as parameters to the Automation server. Because Automation servers can be written in any programming language, you are restricted to data types that are common to all major programming languages.

Numeric Data Types

All integer variables that will be used in calls to Automation servers should be declared as a 2 or 4 byte COMP-5 item. For example:

```
* 2-byte integers
 01 Parameter1                pic xx comp-5.
 01 Parameter2                pic 9(4) comp-5.

* 4-byte integers
 01 Parameter3                pic x(4) comp-5.
 01 Parameter4                pic 9(9) comp-5.
```

If you want to use signed items, declare them as follows:

```
 01 Parameter3                pic s9(4) comp-5.
 01 Parameter4                pic s9(9) comp-5.
```

If the Automation Server is expecting a *Boolean* parameter, it should be declared as a one byte COMP-5 item as follows:

```
 01 Boolean                   pic x comp-5.
```

A value of 0 is used to indicate False. A value of 1 is used to indicate True.

Non-integer numeric data should be passed as either COMP-1 or COMP-2.

Strings

Strings require some care. In most cases, strings are handled internally by COM using a type called a *BSTR*. This is a null-terminated string that is prefixed by the number of characters in the string. There is no COBOL data type that corresponds to this, so the COBOL Run-Time System handles the conversion between BSTR and PIC X fields

It is recommended that any string that is going to be passed to an Automation server be zero-terminated. This means that the last character in the string must be a null character (X"00). To do this in COBOL, either append a null character to the string or use the 'z' prefix. For example:

```
01 Example-Text         pic x(43) value
   z"This is an example document created using ".

invoke TextRange "InsertAfter" using
       z"Micro Focus COBOL and Microsoft Word."
```

It is also important to remember that any string returned from an Automation server should be zero-terminated. This means that you should normally check for a null character and remove it before using the string in the COBOL program.

The following code segment shows how a string returned from an Automation Server can be converted into a standard COBOL space-terminated string. By moving low-values to the string before invoking Word to get the name of the application, we ensure that every character after the name will be null. This also ensures that no previous data will be left in the string since only the part of the string that is affected will be updated.

```
01 AppName              pic x(70).

move low-values to AppName
invoke Word "getName" returning AppName
inspect AppName replacing all X"00" by space
```

We will take a more detailed look at data types and how you choose which one to use later in this paper.

Example Using Excel

To show a more complex example of using an object model and different data types, the following example uses Microsoft Excel. This program reads a COBOL file containing fictional target and actual sales information for a company that does business in different European countries. An Excel worksheet is updated with this information and a chart is created to display the information graphically. This chart is then printed. You will see that the structure of this application is very similar to the previous one.

```
$set ooctrl(+P)
   file-control.
       select Sales-File assign "SALESDAT.DAT"
           organization indexed
           access sequential
           record key Country.

   class-control.
       MicrosoftExcel is class "$OLE$Excel.Application".

   data division.

   file section.
   fd  Sales-File.

   01 Sales-record.
    03 Country                  pic x(20).
    03 Sales-Target             pic 9(8).
    03 Sales-Actual             pic 9(8).
```

```
 working-storage section.

 01 Excel                object reference.
 01 WorkBooks            object reference.
 01 WorkBook             object reference.
 01 WorkSheets           object reference.
 01 WorkSheet            object reference.
 01 Cell                 object reference.
 01 CellRange            object reference.
 01 Charts               object reference.
 01 Chart                object reference.
 01 ColumnIndex          pic xx comp-5.
 01 FloatValue           comp-1.
 01 EndOfFile            pic 9 value 0.

 78 Automation-True      value 1.
 78 Automation-False     value 0.

 78 xl3DColumn           value -4100.

 procedure division.
*    Create a new instance of Microsoft Excel
     invoke MicrosoftExcel "new" returning Excel
*    Make Excel visible
     invoke Excel "setVisible"
         using by value Automation-True
*    Get the collection of WorkBooks
     invoke Excel "getWorkBooks" returning WorkBooks
*    Add a new WorkBook to the collection
     invoke WorkBooks "Add" returning WorkBook
     invoke WorkBook "getWorkSheets"
         returning WorkSheets
     invoke WorkSheets "getItem"
         using by value 1
         returning WorkSheet
*    Set the first column to the titles for the rows
     invoke WorkSheet "getCells" using by value 2
                                       by value 1
                                  returning Cell
     invoke Cell "setValue" using z"Target"
     invoke Cell "Finalize" returning Cell
     invoke WorkSheet "getCells" using by value 3
                                       by value 1
                                  returning Cell
     invoke Cell "setValue" using z"Actual"
     invoke Cell "Finalize" returning Cell
*    Now, read the sales file, filling in the columns in
*    the worksheet
     open input Sales-File
     move 2 to ColumnIndex
     perform until EndOfFile = 1
         read Sales-File
             at end move 1 to EndOfFile
         end-read
         if EndOfFile = 0
             invoke Excel "getCells"
                 using by value 1
```

```
                        by value ColumnIndex
                    returning Cell
                invoke Cell "setValue" using Country
                invoke Cell "finalize" returning Cell
                invoke Excel "getCells"
                    using by value 2
                             by value ColumnIndex
                    returning Cell
                move Sales-Target to FloatValue
                invoke Cell "setValue"
                    using by value FloatValue
                invoke Cell "finalize" returning Cell
invoke Excel "getCells"
                    using by value 3
                          by value ColumnIndex
                    returning Cell
                move Sales-Actual to FloatValue
                invoke Cell "setValue"
                    using by value FloatValue
                invoke Cell "finalize" returning Cell
                add 1 to ColumnIndex
            end-if
        end-perform
  *     Select the first 3 rows of cells
        invoke Excel "getRows" using z"1:3"
                               returning CellRange
        invoke CellRange "Select"
  *     Get the collection of charts
        invoke Excel "getCharts" returning Charts
  *     Add a new chart to the collection. This will create
  *     a chart using the 3 rows we selected above
        invoke Charts "Add" returning Chart
  *     Set the chart type to be 3D-Column
        invoke Chart "setType" using by value xl3DColumn
  *     Print the chart
        invoke Chart "PrintOut"
  *     Close the WorkBook, discarding the contents
        invoke WorkBook "Close" using by value 0
  *     Finalize all objects
        invoke Chart "finalize" returning Chart
        invoke Charts "finalize" returning Charts
        invoke CellRange "finalize" returning CellRange
        invoke WorkSheet "finalize" returning WorkSheet
        invoke WorkSheets "finalize" returning WorkSheets
        invoke WorkBook "finalize" returning WorkBook
        invoke WorkBooks "finalize" returning WorkBooks
  *     Exit Excel
        invoke Excel "Quit"
        invoke Excel "Finalize" returning Excel
        stop run.
```

Figure 3 shows the Excel display just before the chart is printed:

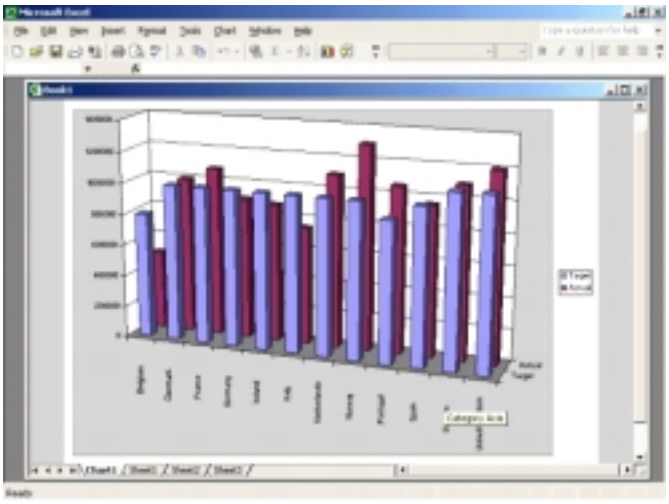

Figure 3 – The chart that will be printed by the example

Where Do I Find More Information on Automation Servers?

Hopefully, by now, you have seen that using Automation servers from COBOL is not difficult. However, how do you find out what functions are provided by the Automation server and the parameters they require? The answer to this question is not so easy for COBOL programmers since most documentation for Automation servers is written with Visual Basic programmers in mind. This section focuses on how to find the information you need and how to interpret it so that you can call the server from COBOL.

Documentation

For Microsoft applications, the best source of information is the Microsoft Developer Network (MSDN) Library CD available from Microsoft Corporation. This information can also be found on Microsoft's web site at http://msdn.microsoft.com/library. This provides information on the object models used in Microsoft Office and other Microsoft products.

For Microsoft Office, look in the Microsoft Office documentation for the Visual Basic Reference guide for the application you are interested in. All of the Microsoft Office applications use Visual Basic for Applications as their macro language. The language reference will list all of the objects, methods and properties for the application.

Translating from Visual Basic to COBOL

Once you have found a description of the method or property you want to use, how do you determine how to use it from COBOL? In order to do this, we need to look at how Automation servers are used from Visual Basic and then take a look at the COBOL equivalent. In the following sections, any reference to Visual Basic also includes Visual Basic for Applications.

Basic method calls

From Visual Basic, you would call a method using the following syntax if there was no return value:

```
Call object.method(parameters)
```

Or:

```
object.method(parameters)
```

From COBOL, you would use:

```
Invoke object "method" using parameters
```

If the method has a return value, you would use the following from Visual Basic:

```
RetVal = object.method(parameters)
```

From COBOL, you would use:

```
Invoke object "method" using parameters
                       returning RetVal
```

Pass numeric parameters using BY VALUE. Use BY REFERENCE to pass strings as parameters. For example:

```
invoke WorkBook "Close" using by value 0
invoke TextRange "InsertAfter"
    using by reference Example-Text
```

Note. BY REFERENCE is the default, so if neither BY VALUE or BY REFERENCE is specified, then BY REFERENCE is assumed.

So, the following two lines behave identically:

```
invoke TextRange "InsertAfter"
    using by reference Example-Text
invoke TextRange "InsertAfter"
    using Example-Text
```

Properties

In Visual Basic, you can move values to a property of an object and you can move the value of a property to a variable. For example:

```
object.Property = 5
avalue = object.Property
```

In COBOL, you make explicit method calls, prefixing the property name with “set” or “get”, for example:

```
invoke object “setProperty” using by value 5
invoke object “getProperty” returning avalue
```

Method Chaining

In Visual Basic, if a method returns an object, you can chain that object by directly calling another method on it. For instance:

```
MyRange.Find.Execute
```

The “Find” property method returns another object, and the method “Execute” is then called on that object, after which the temporary “Find” object is automatically released. This is equivalent to:

```
Dim findObject As Object
findObject = MyRange.Find
findObject.Execute
Set findObject = Nothing
```

In COBOL, this must be done explicitly, as follows:

```
01 findObject       object reference.
…
invoke MyRange “getFind” returning findObject
invoke findObject “Execute”
invoke findObject “finalize” returning findObject
```

In each case, both the Visual Basic Run-Time System and the COBOL Run-Time System are making the same number of calls to the Automation server.

Default Methods and Properties

For a particular collection of objects, one method or property may be declared as the “default”. Visual Basic may allow the programmer to assume the default method when chaining methods together. For example, in a Visual Basic program that uses Microsoft Word you might see:

```
Documents(1).Activate
```

This uses the default property, **Item**, for the Documents collection, which takes an integer parameter and returns the specific Document object from the collection. Once again, this object is chained by calling another method on it. If this was expanded out to show the code that is being executed, you would see:

```
Dim Document as Object

Document = Documents.Item(1)
Document.Activate
Set Document To Nothing
```

In COBOL, all of the methods assumed by Visual Basic should be invoked:

```
01 Document   object reference.
…
invoke Documents "getItem" using by value 1
                 returning Document
invoke Document "Activate"
invoke Document "finalize" returning Document
```

Getting Information from Type Libraries

One of the most useful sources of information about the methods supported by an Automation server and the parameters required for those methods is a type library. A type library is a binary file that contains specifications of the objects, methods and properties supported by an Automation server. Most Automation servers will provide a type library, either as a separate file (usually with a file extension of .TLB or .OLB) or built-in to the Automation server executable.

All of the Microsoft Office applications provide type libraries. They are installed in the same directory as the Microsoft Office executables. For Office XP, they have the following names:

MSACC.OLB	Type library for Microsoft Access
MSOUTL.OLB	Type library for Microsoft Outlook
MSPPT.OLB	Type library for Microsoft PowerPoint
MSWORD.OLB	Type library for Microsoft Word
XL5EN32.OLB	Type library for Microsoft Excel

Micro Focus Net Express 3.1 includes a useful utility to read a type library and generate a COBOL copy file containing equivalent COBOL definitions. This utility is called the *Type Library Assistant*.

Running the Type Library Assistant

The Type Library Assistant can be found on the Tools menu in Net Express. When you run it, you will see a window similar to that shown in Figure 4.

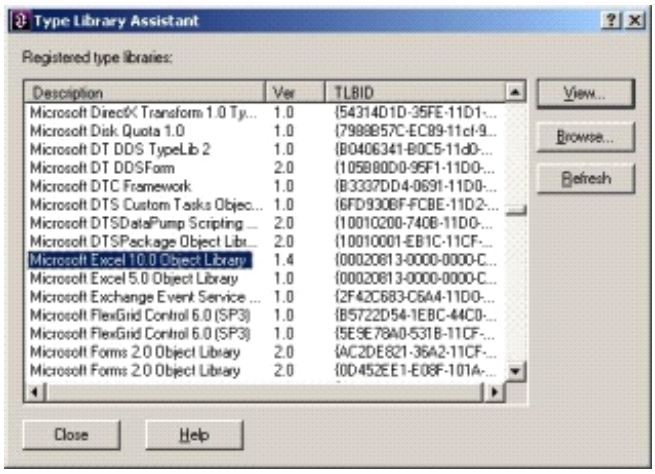

Figure 4 – The Type Library Assistant

This window shows all of the type libraries that are registered on your computer – there may be a lot of them. Type libraries are normally registered when the corresponding applications are installed. Microsoft Office is no exception. You can see that the type library for Microsoft Excel 10.0 has been selected.

If you want to use a type library that has not been registered, you can select **Browse** to search for the type library you want to use. If you press **View**, you will see the window shown in figure 5.

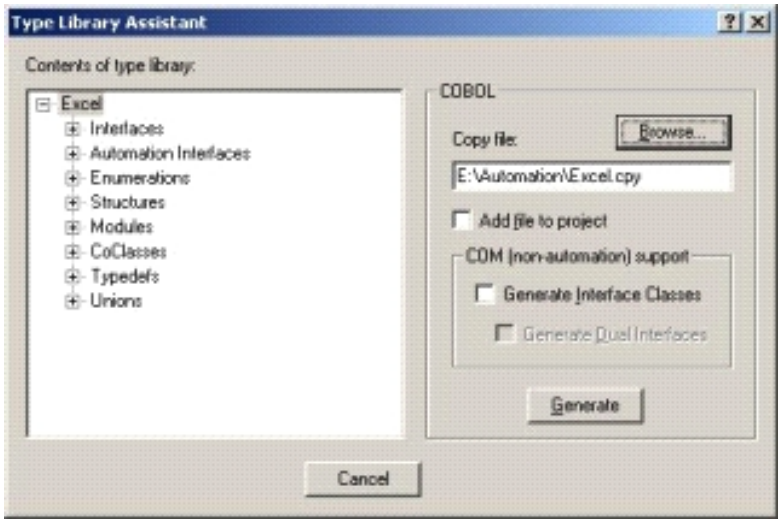

Figure 5 – Generating the copy file

Don't worry about the contents of the box labelled **Contents of type library**. This just allows you to limit the amount of information produced in the copy file. In most cases, you will just select the top level of the tree. Clicking **Generate** will generate the copy file you have requested.

Determining the Parameters to Use for a Method

Although the file generated is a valid COBOL copy file, it is unlikely that you will use every definition included in it unmodified. However, it is an excellent source of information for helping to determine what parameters to use when calling a particular function and which parameters are needed. For example, consider the Excel method **SaveAs** which can be used on a Worksheet. If we look in the Office XP documentation for Excel, we find the following description:

Saves changes to the chart or worksheet in a different file.

```
expression.SaveAs(FileName, FileFormat, Password,
WriteResPassword, ReadOnlyRecommended, CreateBackup, AddToMru,
TextCodepage, TextVisualLayout, Local)
```

expression Required. An expression that returns one of the above objects.

Filename Optional Variant. A string that indicates the name of the file to be saved. You can include a full path; if you don't, Microsoft Excel saves the file in the current folder.

FileFormat Optional Variant. The file format to use when you save the file. For a list of valid choices, see the **FileFormat** property. For an existing file, the default format is the last file format specified; for a new file, the default is the format of the version of Excel being used.

Password Optional Variant. A case-sensitive string (no more than 15 characters) that indicates the protection password to be given to the file.

WriteResPassword Optional Variant. A string that indicates the write-reservation password for this file. If a file is saved with the password and the password isn't supplied when the file is opened, the file is opened as read-only.

ReadOnlyRecommended Optional Variant. True to display a message when the file is opened, recommending that the file be opened as read-only.

CreateBackup Optional Variant. True to create a backup file.

AddToMru Optional Variant. True to add this workbook to the list of recently used files. The default value is False.

TextCodePage Optional Variant. Not used in U.S. English Microsoft Excel.

TextVisualLayout Optional Variant. Not used in U.S. English Microsoft Excel.

Local Optional Variant. True saves files against the language of Microsoft Excel (including control panel settings). False (default) saves files against the language of Visual Basic for Applications (VBA) (which is typically US English unless the VBA project where Workbooks.Open is run from is an old internationalized XL5/95 VBA project).

How do you convert this into the appropriate invoke statement from COBOL? The first thing to do is look at the definition for SaveAs that was produced in the copy file:

```
Method: "SaveAs".
01 Filename                    BSTR.
01 FileFormat                  VARIANT.
01 Password                    VARIANT.
```

```
01 WriteResPassword            VARIANT.
01 ReadOnlyRecommended         VARIANT.
01 CreateBackup                VARIANT.
01 AddToMru                    VARIANT.
01 TextCodepage                VARIANT.
01 TextVisualLayout            VARIANT.
01 Local                       VARIANT.
invoke using
   by value Filename             *> [ IN]
   by value FileFormat           *> [ IN][ OPTIONAL]
   by value Password             *> [ IN][ OPTIONAL]
   by value WriteResPassword     *> [ IN][ OPTIONAL]
   by value ReadOnlyRecommended  *> [ IN][ OPTIONAL]
   by value CreateBackup         *> [ IN][ OPTIONAL]
   by value AddToMru             *> [ IN][ OPTIONAL]
   by value TextCodepage         *> [ IN][ OPTIONAL]
   by value TextVisualLayout     *> [ IN][ OPTIONAL]
   by value Local                *> [ IN][ OPTIONAL] .
```

The first thing to notice is the comment "[OPTIONAL]" next to all of the parameters except the first one (**Filename**). This means that every parameter except the first one is optional and does not need to be specified. However, if you wish to specify one of the optional parameters, you must specify all previous parameters. For example, if you want to specify **Password**, you must also specify **Filename** and **FileFormat** in the list of parameters you pass to the method.

Next, notice that all of the parameters are specified as "[IN]". This means that they are all input parameters. If the method returns a value, you would see a **returning** clause on the invoke statement in the copy file.

Now we have to determine what types to use for the parameters. Any parameter specified as **BSTR** is a string. In COBOL, this means that a variable defined as pic x(n) or a string constant is used.

---

*Important Note. Even though the parameters in the definition are specified as BY VALUE, you **ALWAYS** pass strings to Automation servers BY REFERENCE. Strings cannot be passed BY VALUE. The COBOL Run-Time System will handle the string correctly when passing it to the Automation server.*

The rest of the parameters have a type of **VARIANT**. A variant is a COM data type that can contain different types. In COBOL terminology, it is similar in concept to a REDEFINES statement where a variable can be redefined as a different type. For example:

```
01 VarA                pic x(4).
01 VarB redefines VarA pic 9(9) comp-5.
```

VarA and VarB refer to the same area of memory, but that piece of memory can be used as a string or a 4-byte value depending on whether it is accessed using the name VarA or VarB. A variant type is similar, but it uses a type indicator to determine the type of the data stored in the variant.

Net Express includes a COBOL class to create and manipulate variants (look at the definition of the class **OLEVariant** in the Net Express documentation). However, you should very rarely need to use this class since the COBOL Run-Time System handles the conversion of COBOL data types to variants automatically when the method is called and converts any variant returned into the appropriate COBOL data type. So, if a variant is

specified, you should look at the description of the function to see the type of parameter that is being expected.

For example, for the FileFormat parameter, it refers to the FileFormat property. If we look up FileFormat in the Excel documentation, it references a type of **xlFileFormat.** Looking up xlFileFormat in the copy file generated by the Type Library Assistant, we find the definition:

```
01 XlFileFormat    pic s9(9) comp-5 typedef.
   88 xlAddIn                    VALUE 18.
   88 xlCSV                      VALUE 6.
   88 xlCSVMac                   VALUE 22.
   88 xlCSVMSDOS                 VALUE 24.
   88 xlCSVWindows               VALUE 23.
   88 xlDBF2                     VALUE 7.
   88 xlDBF3                     VALUE 8.
   88 xlDBF4                     VALUE 11.
   88 xlDIF                      VALUE 9.
   88 xlExcel2                   VALUE 16.
   88 xlExcel2FarEast            VALUE 27.
   88 xlExcel3                   VALUE 29.
   88 xlExcel4                   VALUE 33.
   88 xlExcel5                   VALUE 39.
   88 xlExcel7                   VALUE 39.
   88 xlExcel9795                VALUE 43.
   88 xlExcel4Workbook           VALUE 35.
   …
```

This tells us that the parameter to use for the FileFormat is a 4-byte signed integer (pic s9(9) comp-5) and it gives us the values of the constants used to specify different file formats.

If we look at the description of the Password parameter, we see that it says that it is a string. Therefore, we use a null-terminated pic x(n) field or string constant.

We can determine the types of the remaining parameters in a similar fashion.

So, this means that the following are all valid invokes of SaveAs:

```
01 Worksheet            object reference.
01 XlFileFormat         pic s9(9) comp-5.
 88 xlExcel9795         value 43.
78 xlExcel9795c         value 43.

invoke WorkSheet "SaveAs" using FileName

invoke WorkSheet "SaveAs" using
     by reference z"C:\Temp\data.xls"
     by value xlExcel9795c

set xlExcel9795 to true
invoke WorkSheet "SaveAs" using
     by reference z"C:\Temp\data.xls"
     by value xlFileFormat
     by reference z"Password"
```

It might take a bit of investigation to determine exactly the right parameters for a method call, but by using a combination of the documentation for the Automation server

and the copy file produced by the Type Library Assistant, you can find the information you need.

As another example, the following line was used in the first Excel example in this paper:

```
       invoke Chart "setType" using by value xl3DColumn
```

where xl3DColumn is defined as:

```
       78 xl3DColumn           value -4100.
```

How did we determine the value for this constant? If we look in the Excel documentation for chart types, we will find a long list of different types. Here is just a subset, showing the types of column chart available:

Description	**Constant**
Clustered Column	xlColumnClustered
3D Clustered Column	xl3DColumnClustered
Stacked Column	xlColumnStacked
3D Stacked Column	xl3DColumnStacked
100% Stacked Column	xlColumnStacked100
3D 100% Stacked Column	xl3DColumnStacked100
3D Column	xl3DColumn

You will notice that for each type, there is a named constant. If we search the copy file generated by the Type Library Assistant for the value **xl3DColumn**, we will find the following definition:

```
       01 XlChartType    pic s9(9) comp-5 typedef.
          88 xlColumnClustered          VALUE 51.
          88 xlColumnStacked            VALUE 52.
          88 xlColumnStacked100         VALUE 53.
          88 xl3DColumnClustered        VALUE 54.
          88 xl3DColumnStacked          VALUE 55.
          88 xl3DColumnStacked100       VALUE 56.
          88 xlBarClustered             VALUE 57.
          88 xlBarStacked               VALUE 58.
          88 xlBarStacked100            VALUE 59.
          88 xl3DBarClustered           VALUE 60.
          88 xl3DBarStacked             VALUE 61.
          88 xl3DBarStacked100          VALUE 62.
          88 xlLineStacked              VALUE 63.
          88 xlLineStacked100           VALUE 64.
          88 xlLineMarkers              VALUE 65.
          88 xlLineMarkersStacked       VALUE 66.
          88 xlLineMarkersStacked100    VALUE 67.
          88 xlPieOfPie                 VALUE 68.
          88 xlPieExploded              VALUE 69.
          88 xl3DPieExploded            VALUE 70.
```

```
88 xlBarOfPie                  VALUE 71.
88 xlXYScatterSmooth           VALUE 72.
88 xlXYScatterSmoothNoMarkers VALUE 73.
88 xlXYScatterLines            VALUE 74.
88 xlXYScatterLinesNoMarkers  VALUE 75.
88 xlAreaStacked               VALUE 76.
88 xlAreaStacked100            VALUE 77.
88 xl3DAreaStacked             VALUE 78.
88 xl3DAreaStacked100          VALUE 79.
88 xlDoughnutExploded          VALUE 80.
88 xlRadarMarkers              VALUE 81.
88 xlRadarFilled               VALUE 82.
88 xlSurface                   VALUE 83.
88 xlSurfaceWireframe          VALUE 84.
88 xlSurfaceTopView            VALUE 85.
88 xlSurfaceTopViewWireframe  VALUE 86.
88 xlBubble                    VALUE 15.
88 xlBubble3DEffect            VALUE 87.
88 xlStockHLC                  VALUE 88.
88 xlStockOHLC                 VALUE 89.
88 xlStockVHLC                 VALUE 90.
88 xlStockVOHLC                VALUE 91.
88 xlCylinderColClustered      VALUE 92.
88 xlCylinderColStacked        VALUE 93.
88 xlCylinderColStacked100     VALUE 94.
88 xlCylinderBarClustered      VALUE 95.
88 xlCylinderBarStacked        VALUE 96.
88 xlCylinderBarStacked100     VALUE 97.
88 xlCylinderCol               VALUE 98.
88 xlConeColClustered          VALUE 99.
88 xlConeColStacked            VALUE 100.
88 xlConeColStacked100         VALUE 101.
88 xlConeBarClustered          VALUE 102.
88 xlConeBarStacked            VALUE 103.
88 xlConeBarStacked100         VALUE 104.
88 xlConeCol                   VALUE 105.
88 xlPyramidColClustered       VALUE 106.
88 xlPyramidColStacked         VALUE 107.
88 xlPyramidColStacked100      VALUE 108.
88 xlPyramidBarClustered       VALUE 109.
88 xlPyramidBarStacked         VALUE 110.
88 xlPyramidBarStacked100      VALUE 111.
88 xlPyramidCol                VALUE 112.
88 xl3DColumn                  VALUE -4100.
88 xlLine                      VALUE 4.
88 xl3DLine                    VALUE -4101.
88 xl3DPie                     VALUE -4102.
88 xlPie                       VALUE 5.
88 xlXYScatter                 VALUE -4169.
88 xl3DArea                    VALUE -4098.
88 xlArea                      VALUE 1.
88 xlDoughnut                  VALUE -4120.
88 xlRadar                     VALUE -4151.
```

You will see that xl3DColumn is defined with a value of -4100. You can use this entire definition in your program and use the following code to set the chart type:

```
01 ChartType      xlChartType.
```

```
       set xl3DColumn of ChartType to true
       invoke Chart "setType" using by value ChartType
```

Alternatively, you could use the approach taken in the earlier program and define a constant (a level 78 item) of the appropriate value and use that constant directly. Either approach will work.

Using Methods with a Get or Set Prefix

Earlier, in the discussion on methods and properties, we said that you prefixed the name of a property with **get** to retrieve the value of a property or with **set** to set the value of a property. However, what happens if the Automation server provides a method name that begins with set or get? For example, Microsoft Excel provides a method **GetSaveAsFileName** that prompts the user for the filename to be used to save the file. Since this method begins with **Get**, the COBOL Run-Time System will try to retrieve the value of a property called **SaveAsFileName** and will fail.

To overcome this, we need to use a method in a COBOL support class, *olesup,* to override the default behavior. This method is called **setDispatchType**. The following program shows this in use:

```
   $set ooctrl(+P)

    class-control.
        MicrosoftExcel is class "$OLE$Excel.Application"
        AutomationSupport is class "olesup".

    data division.

    working-storage section.

    01 Excel                object reference.
    01 WorkBooks            object reference.
    01 WorkBook             object reference.
    01 WorkSheets           object reference.
    01 WorkSheet            object reference.
    01 Cell                 object reference.
    01 CellRange            object reference.

    78 Automation-True      value 1.
    78 Automation-False     value 0.

    01 FileFilter           pic x(46) value
       z"Excel Files (*.xls),*.xls,All Files (*.*),*.*".

    01 FileName             pic x(100).
procedure division.
   *    Create a new instance of Microsoft Excel
        invoke MicrosoftExcel "new" returning Excel
   *    Make Excel visible
        invoke Excel "setVisible"
            using by value Automation-True
   *    Get the collection of WorkBooks
        invoke Excel "getWorkBooks" returning WorkBooks
   *    Add a new WorkBook to the collection
        invoke WorkBooks "Add" returning WorkBook
        invoke WorkBook "getWorkSheets"
            returning WorkSheets
```

```
           invoke WorkSheets "getItem" using by value 1
                                       returning WorkSheet
      *    Set the first cell to "Example Sheet"
           invoke WorkSheet "getCells" using by value 1
                                             by value 1
                                       returning Cell
           invoke Cell "setValue" using z"Example Sheet"
           invoke Cell "Finalize" returning Cell
      *    Override the default behavior of a method prefixed
      *    by "get"
           invoke AutomationSupport "setDispatchType"
               using by value 0
           move low-values to FileName
      *    Get the filename to be used to save the file
           invoke Excel "GetSaveAsFileName"
               using by reference z"Sample"
                     by reference FileFilter
               returning FileName
      *    Use the name we just retrieved to save the file
invoke WorkSheet "SaveAs" using FileName
      *    Close the WorkBook, discarding the contents
           invoke WorkBook "Close" using by value 0
      *    Finalize all objects
           invoke WorkSheet "finalize" returning WorkSheet
           invoke WorkSheets "finalize" returning WorkSheets
           invoke WorkBook "finalize" returning WorkBook
           invoke WorkBooks "finalize" returning WorkBooks
      *    Exit Excel
           invoke Excel "Quit"
           invoke Excel "Finalize" returning Excel
           stop run.
```

A new class has been added to the Class Control section, as follows:

```
           AutomationSupport is class "olesup".
```

This class is a COBOL class provided with Net Express. It provides a number of methods that support the use of COM and Automation from COBOL. Note that this class is a standard COBOL class and so the name is not prefixed with "OLE". More information on this class can be found in the Net Express documentation. The method we need in this program is **setDispatchType**. We use this just before the invoke of **GetSaveAsFileName** as follows:

```
           invoke AutomationSupport "setDispatchType"
               using by value 0
```

This overrides the default behavior of the **Get** prefix for the next invoke of an Automation method. Therefore, **GetSaveAsFileName** will be handled as a method call, rather than a property get.

The same call would be used before a method that begins with the prefix **Set**.

Error Handling

You will notice that there has been no error handling in the examples shown in this paper so far. If an error occurs in any of the calls to the Automation servers, the COBOL Run-Time System will simply display an error on the screen and give you an opportunity to stop the program. Obviously, for real-life applications, we need to ensure that errors that

occur when using the Automation server are handled correctly. To do this, we need to install an *exception handler* to handle the error appropriately.

The following program installs an exception handler and then attempts to start up an Automation server that does not exist. Rather than stopping with a run time error, the error is trapped by the exception handler.

```
$set ooctrl(+P)

 class-control.
     AutomationServer is class "$OLE$Unknown"
     EntryPointCallback is class "entrycll"
     ExceptionManager is class "exptnmgr"
     AutomationExceptionManager is class "oleexpt".

 working-storage section.

 01 AutomationObject     object reference.
 01 NullReference        object reference value null.
 01 HandlerObject        object reference.

 01 ErrorOccurred        pic 9 value 0.

*  An empty local-storage section is needed to ensure
*  that the program is re-entrant.
 local-storage section.

* Parameters for Exception Callback
 linkage section.
 01 ErrorNumber      pic x(4) comp-5.
 01 ErrorObject      object reference.
 01 ErrorText        object reference.

 procedure division.
*    Register an exception handler
     invoke EntryPointCallback "new"
         using z"AutomationException"
         returning HandlerObject
     invoke ExceptionManager "register"
         using AutomationExceptionManager
               HandlerObject
*    Attempt to startup the Automation Server
     invoke AutomationServer "new"
         returning AutomationObject
     if ErrorOccurred = 1
         display "Unable to load the Automation Server"
         stop run
     end-if
     invoke AutomationObject "finalize"
         returning AutomationObject
     stop run.

 Callback section.
 entry "AutomationException"
     using by reference ErrorObject
           by reference ErrorNumber
           by reference ErrorText.
     move 1 to ErrorOccurred
```

```
display "Error number: " ErrorNumber
invoke ErrorText "display"
exit program returning NullReference.
```

The first thing you will notice is that three extra classes are defined in the class control section.

```
EntryPointCallback is class "entrycll"
ExceptionManager is class "exptnmgr"
AutomationExceptionManager is class "oleexpt".
```

These classes are COBOL classes that are used to handle exceptions. For more information on these classes, refer to the Net Express documentation.

Next we define an empty local storage section and a linkage section. The presence of a local storage section indicates to the COBOL Run-Time System that the program is allowed to be recursive, that is, calls can be made back to itself (by default, COBOL is not recursive). The linkage section defines the three parameters that will be passed to the exception handling routine when an exception occurs.

The first thing you see in the procedure division is the two lines of code that register the COBOL entry point, **AutomationException**, which will handle any exceptions:

```
invoke EntryPointCallback "new"
    using z"AutomationException"
    returning HandlerObject
invoke ExceptionManager "register"
    using AutomationExceptionManager
          HandlerObject
```

This means that, from that point on in the program, if any error occurs in an Automation server, the entry point **AutomationException** will be called. This entry point will be passed three parameters

- A number identifying the error that occurred
- The object that caused the error
- A reference to an ordered collection that contains a description of the error. You can use methods on the ordered collection object to get the different lines of the description. In this example, we simply invoke the method to display the text.

Note. The "Distributed Computing" book in the Net Express 3.1 documentation contains errors in its description of exception handlers. It only mentions the first two parameters. However, a third parameter containing the error text is always passed through to the exception handler routine.

In this program, we simply set a flag (**ErrorOccured**) to indicate that an error occurred. When the exception handler finishes, control is returned to the statement immediately after the statement that caused the problem. In this case, it is the line:

```
if ErrorOccurred = 1
```

that tests the flag and stops execution if an error occurred.

In the example above, we mentioned that the error text is returned as an ordered collection containing multiple lines. In most cases, you will only want the first line. The

following code shows an alternative exception handling routine that you could use that would set the first line of the error message in the COBOL string **ExceptionText** and the length of the text in **ExceptionTextLen**

```
  01 ExceptionText        pic x(160) value spaces.
  01 ExceptionTextLen     pic 9(9) comp-5 value 0.
  01 i                    pic 9(9) comp-5.
  01 ErrorTextObj         object reference.
  …

 entry "AutomationException" using
         by reference ErrorObject
         by reference ErrorNumber
         by reference ErrorText.
     move 1 to ErrorOccured
     if ErrorText not = null
         move 1 to i
         invoke ErrorText "at" using i
             returning ErrorTextObj
         move 160 to i
         invoke ErrorTextObj "getValueWithSize" using i
             returning ExceptionText
*        Remove anything after any x"0D0A" (CR/LF)
*        sequence
         move 1 to i
         perform until i > 160 or
                       ExceptionText(i:1) = x"0D"
             add 1 to i
         end-perform
         if i <= 160
             move spaces to ExceptionText(i:)
             move i to ExceptionTextLen
         else
             move 160 to ExceptionTextLen
         end-if
*        Remove any trailing spaces
         perform until
             ExceptionTextLen = 0 or
             ExceptionText(ExceptionTextLen:1) not = space
             subtract 1 from ExceptionTextLen
         end-perform
         display ExceptionText(1:ExceptionTextLen)
     else
         move spaces to ExceptionText
         move 0 to ExceptionTextLen
     end-if
     exit program returning NullReference.
```

More on Data Types

Using Arrays

Some functions exposed by Automation Servers require that you pass an array to the function. For example, if you want to populate a range of cells in an Excel spreadsheet in one single Invoke, you would need to use an array. These arrays are called *SafeArrays*.

Net Express includes an Object COBOL class that provides you with the functionality to:

- Create a Safe Array
- Add data to a Safe Array
- Retrieve data from a Safe Array
- Destroy a Safe Array

The following example shows how Safe Arrays can be used in Excel to populate ranges of cells:

```
  $set ooctrl(+P)
   class-control.
       MicrosoftExcel is class "$OLE$Excel.Application"
       OleSafeArray is class "olesafea".

   working-storage section.
   copy "olesafea.cpy".

   01 Excel                object reference.
   01 WorkBooks            object reference.
   01 WorkBook             object reference.
   01 WorkSheets           object reference.
   01 WorkSheet            object reference.
   01 CellRange            object reference.
   01 Charts               object reference.
   01 Chart                object reference.

   01 LoopCount            pic xx comp-5.

   01 saBound              SAFEARRAYBOUND.
   01 bstrSafeArray        object reference.
   01 intSafeArray         object reference.
   01 saIndex              pic x(4) comp-5.
   01 hResult              pic x(4) comp-5.
   01 iValue               pic x(2) comp-5.
   01 dPointer             pointer.
78 Automation-True      value 1.
   78 Automation-False     value 0.

   78 xl3DBar              value -4099.

   procedure division.
  *    Create a new instance of Microsoft Excel
       invoke MicrosoftExcel "new" returning Excel
  *    Make Excel visible
```

```
      invoke Excel "setVisible"
          using by value Automation-True
*     Get the collection of WorkBooks
      invoke Excel "getWorkBooks" returning WorkBooks
*     Add a new WorkBook to the collection
      invoke WorkBooks "Add" returning WorkBook
*     Get a reference to the first WorkSheet
      invoke WorkBook "getWorkSheets"
          returning WorkSheets
      invoke WorkSheets "getItem" using by value 1
                                  returning WorkSheet
*     Select the range of cells to populate
      invoke WorkSheet "getRange" using z"A1:C1"
                                  returning CellRange
*     Create a 3-element safearray containing strings
      move 3 to cElements of saBound
      move 0 to llBound of saBound
      invoke OleSafeArray "new"
          using by value VT-BSTR size 2
                by value 1 size 4
                by reference saBound
          returning bstrSafeArray
      move 0 to saIndex
      invoke bstrSafeArray "putString"
          using by reference saIndex
                by value 4 size 4
                by reference "Dogs"
          returning hResult
      move 1 to saIndex
      invoke bstrSafeArray "putString"
          using by reference saIndex
                by value 4 size 4
                by reference "Cats"
          returning hResult
      move 2 to saIndex
      invoke bstrSafeArray "putString"
          using by reference saIndex
                by value 6 size 4
                by reference "Horses"
          returning hResult
*     Populate the range of cells from the safearray
      invoke CellRange "setValue"
          using by value bstrSafeArray
      invoke bstrSafeArray "Finalize"
          returning bstrSafeArray
      invoke CellRange "Finalize" returning CellRange
*     Get a new range to populate
      invoke WorkSheet "getRange"
          using z"A2:C2"
          returning CellRange
*     Create a 3-element safearray containing 2-byte
*     integers
      move 3 to cElements of saBound
      move 0 to llBound of saBound
      invoke OleSafeArray "new"
          using by value VT-I2 size 2
                by value 1 size 4
                by reference saBound
```

```
            returning intSafeArray
        set dPointer to address of iValue
        move 34 to iValue
        move 0 to saIndex
        invoke intSafeArray "putElement"
            using by reference saIndex
                  by value dPointer
            returning hResult
        move 53 to iValue
        move 1 to saIndex
        invoke intSafeArray "putElement"
            using by reference saIndex
                  by value dPointer
            returning hResult
        move 12 to iValue
        move 2 to saIndex
        invoke intSafeArray "putElement"
            using by reference saIndex
                  by value dPointer
            returning hResult
*       Populate the range from the safearray
        invoke CellRange "setValue"
            using by value intSafeArray
        invoke intSafeArray "Finalize"
            returning intSafeArray
        invoke CellRange "Finalize" returning CellRange
*       Get range to select and select it
        invoke WorkSheet "getRange"
            using z"A1:C2"
            returning CellRange
        invoke CellRange "Select"
*       Get the collection of charts
        invoke Excel "getCharts" returning Charts
*       Add a new chart to the collection
        invoke Charts "Add" returning Chart
*       Set the chart type to be 3D-Bar
        invoke Chart "setType" using by value xl3DBar
*       Remove the legend
        invoke Chart "setHasLegend"
            using by value Automation-False
*       Print the chart
        invoke Chart "PrintOut"
*       Close the WorkBook, discarding the contents
        invoke WorkBook "Close"
            using by value Automation-False
*       Finalize all objects
        invoke Chart "finalize" returning Chart
        invoke Charts "finalize" returning Charts
        invoke CellRange "finalize" returning CellRange
        invoke WorkSheet "finalize" returning WorkSheet
        invoke WorkSheets "finalize" returning WorkSheets
        invoke WorkBook "finalize" returning WorkBook
        invoke WorkBooks "finalize" returning WorkBooks
*       Exit Excel
        invoke Excel "Quit"
        invoke Excel "Finalize" returning Excel
        stop run.
```

The following sections describe what you must do to use Safe Arrays:

Use the Class OleSafeArray

Your class control section should include a definition of the *OleSafeArray* class as follows:

```
OleSafeArray is class "olesafea"
```

This class is fully documented in the online help and in the "Distributed Computing" book supplied with Net Express. You should refer to the "Distributed Computing" book for more information on using safe arrays.

Include OleSafeA.cpy

Add the following line to your working storage section:

```
copy "olesafea.cpy".
```

This file includes all of the type definitions and constants needed when using Safe Arrays.

Define a Safe Array of the Appropriate Type and Size

When you need to use a Safe Array, you create a new array by sending the "New" method to the class OleSafeArray. The following information needs to be provided as parameters to the call:

- The type of the element stored in the array. This is specified by using the constants defined in OleSafeA.cpy that have the prefix *VT-*. In the example above, two arrays are created. One contains strings (VT-BSTR) and the other array contains 2-byte integers (VT-I2).
- The number of dimensions in the array.
- The lower and upper bounds for each dimension. These are specified in a SAFEARRAYBOUND structure.

The New method returns an object reference to the array.

Populate the Safe Array

Methods are provided to put data into the elements in a Safe Array. In this program, "PutString" is used to place a string into an array that contains strings and "PutElement" is used to place numbers into the array that holds integers. For example:

```
move 1 to saIndex
invoke bstrSafeArray "putString"
    using by reference saIndex
          by value 4 size 4
          by reference "Cats"
    returning hResult
```

and:

```
set dPointer to address of iValue
move 34 to iValue
move 0 to saIndex
invoke intSafeArray "putElement"
    using by reference saIndex
          by value dPointer
    returning hResult
```

Destroy the Safe Array

When you have finished using the Safe Array, you should destroy it by sending the Finalize method to the array. For example:

```
invoke intSafeArray "Finalize"
    returning intSafeArray
```

More Examples

Here are more examples showing what is possible using Automation.

Using Microsoft Visio

This program shows a different way of representing the sales data used in the first Excel example using Microsoft Visio. Visio is a diagramming tool that can be used to help visualize information. This program uses the data to colorize a map of Europe with data from the different countries, showing graphically whether they are performing above or below target.

```
$set ooctrl(+P)
 file-control.
     select Sales-File assign "SALESDAT.DAT"
         organization indexed
         access dynamic
         record key Country
         status File-Status.

 class-control.
     MicrosoftVisio is class "$OLE$Visio.Application".

 data division.

 file section.
 fd  Sales-File.

 01 sales-record.
  03 Country                    pic x(20).
  03 Sales-Target               pic 9(8).
  03 Sales-Actual               pic 9(8).

 working-storage section.

 01 VSDFile               pic x(35) value
    z"e:\Automation\VisioDemo\Europe.VSD".
```

```
 01 Visio                 object reference.
 01 Docs                  object reference.
 01 Doc                   object reference.
 01 PagesObj              object reference.
 01 PageObj               object reference.
 01 Shapes                object reference.
 01 Shape                 object reference.
 01 ShapeCount            pic 9(9) comp-5.
 01 ShapeIndex            pic 9(9) comp-5.
 01 ShapeName             pic x(20).

 01 file-status.
  03 file-status-1        pic x.
  03 file-status-2        pic x.

 01 Percent-Difference    pic s9(8).

 procedure division.
 Main section.
*    Open a new instance of Visio
     invoke MicrosoftVisio "new" returning Visio
*    Open the map drawing
     invoke Visio "Documents" returning Docs
     invoke Docs "Open"
         using VSDFile
         returning Doc
*    Get the first page in the drawing
     invoke Doc "Pages" returning PagesObj
     invoke PagesObj "getItem" using by value 1
                               returning PageObj
*    Get the collection of shapes
     invoke PageObj "GetShapes" returning Shapes

*    For each of the country shapes, find its record in
*    the data file and determine which color to fill it
*    with based on the sales performance
     open input Sales-File
     perform varying ShapeIndex from 2 by 1
             until ShapeIndex > 13
         invoke Shapes "GetItem"
             using by value ShapeIndex
             returning Shape
         move spaces to Country
         invoke Shape "GetName" returning Country
         inspect Country replacing all X"00" by space
         read Sales-File
         compute Percent-Difference =
                (Sales-Actual - Sales-Target) /
                 Sales-Target * 100
```

```
            evaluate true
                when Percent-Difference <= -20
                    invoke Shape "SetFillStyle"
                        using z"Red fill"

                when Percent-Difference <= -5 and
                     Percent-Difference > -20
                    invoke Shape "SetFillStyle"
                        using z"Magenta fill"

                when Percent-Difference > -5 and
                     Percent-Difference < 5
                    invoke Shape "SetFillStyle"
                        using z"Blue fill"

                when Percent-Difference >= 5 and
                     Percent-Difference < 20
                    invoke Shape "SetFillStyle"
                        using z"Yellow fill"

                when Percent-Difference >= 20
                    invoke Shape "SetFillStyle"
                        using z"Green fill"

            end-evaluate
            invoke Shape "Finalize" returning Shape
        end-perform
        invoke Shapes "Finalize" returning Shapes
        invoke PageObj "Finalize" returning PageObj
        invoke PagesObj "Finalize" returning PagesObj
        close Sales-File
*       Print the document
        invoke Doc "print"
*       Set the 'saved' flag so that we can close the
*       document without being prompted for a save
        invoke Doc "SetSaved" using by value 1
*       Close the document
        invoke Doc "Close"
*       Finalize the remaining objects
        invoke Doc "Finalize" returning Doc
        invoke Docs "Finalize" returning Docs
*       Close Visio
        invoke Visio "Quit"
        invoke Visio "Finalize" returning Visio
        stop run.
```

Figure 6 shows the Visio window just before the drawing is printed:

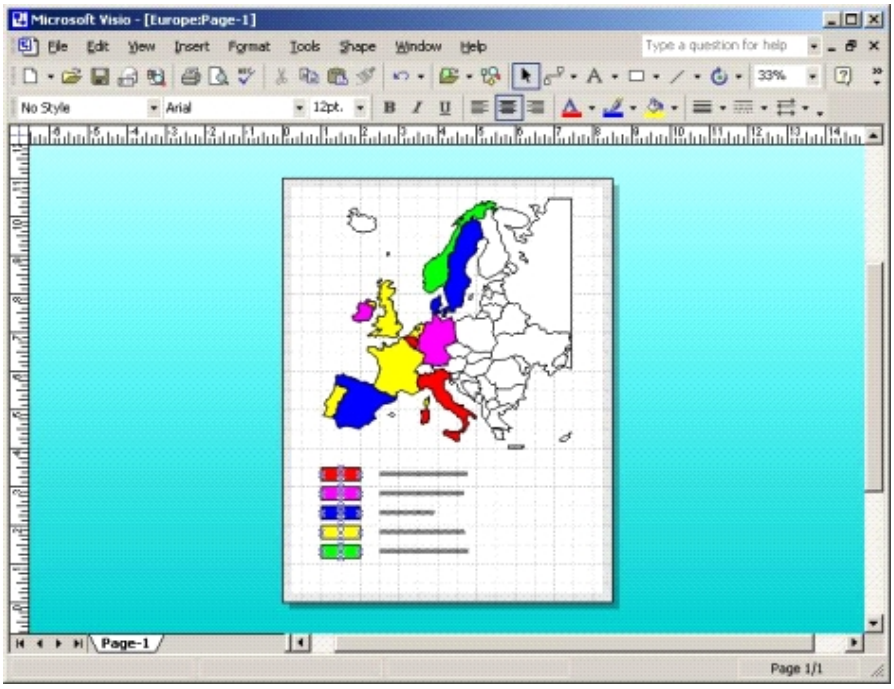

Figure 6 – The Visio Drawing created by the program

Launching Internet Explorer

The following code could be used by your application to launch a copy of Microsoft Internet Explorer and navigate to a particular URL:

```
$set ooctrl(+P)
 class-control.
     IE is class "$OLE$InternetExplorer.Application".

 working-storage section.
 01 IEObject     object reference.

 procedure division.
*    Create a new instance of Internet Explorer
     invoke IE "new" returning IEObject
*    Make Internet Explorer visible
     invoke IEObject "setVisible" using by value 1
*    Navigate to the reguired URL
     invoke IEObject "Navigate"
         using by reference z"http://www.microfocus.com"
*    Cleanup. This will leave Internet Explorer running
     invoke IEObject "Finalize" returning IEObject
     stop run.
```

Conclusion

Hopefully, you have now seen the opportunities made possible to you by the use of Automation from your programs. Automation makes an incredible range of functionality available to you that can be easily exploited from your applications. If you need to provide similar functionality in your application and you know your users have the relevant Automation Server on their workstations, consider using Automation as an alternative to writing the code yourself.

About the author: Wayne Rippin is a self-employed consultant. Previously, he worked for Micro Focus for 16 years, first as a systems programmer and later as a product manager. His most recent role there was director of product management, leading a team of product managers responsible for Net Express, Mainframe Express and UNIX compiler products.

Micro Focus

Choosing the right partner is as critical as choosing the right technology. As you move forward to meet these demands and the demands of your customers, Micro Focus continues to move forward with you as your strategic ally for legacy change. Unlike other e-business vendors, our approach starts with your enterprise legacy system and is designed to leverage, integrate and build upon your legacy assets. We have no computers or applications to sell. Our focus is to build the best tools to make your legacy system better. For more information on this approach or any of the supporting Micro Focus technologies, please contact your Micro Focus representative, or use the contact information listed.