

Automation Part 1

Program examples compiled using Visual C++ 6.0 compiler on Windows XP Pro machine with Service Pack 2. The Excel version is Excel 2003/Office 11. Topics and sub topics for this tutorial are listed below. Don't forget to read Tenouk's small [disclaimer](#). The supplementary notes for this tutorial are [Demo.xls](#), [mymfc29A.xls](#), [mymfc29B.xls](#), [automation.](#), [variant](#) and [COlevariant class](#).

Index:

Intro

Connecting C++ with Visual Basic for Applications (VBA)

Automation Clients and Automation Components

Microsoft Excel: A Better Visual Basic than Visual Basic

Properties, Methods, and Collections

The Problem That Automation Solves

The IDispatch Interface

Automation Programming Choices

The MFC IDispatch Implementation

An MFC Automation Component

An MFC Automation Client Program

An Automation Client Program Using the Compiler's #import Directive

The VARIANT Type

The COleVariant Class

Parameter and Return Type Conversions for Invoke()

Automation Examples

The MYMFC29A Automation Component - EXE Example: No User Interface

MYMFC29A Story

Debugging an EXE Component Program

The MYMFC29B Automation Component DLL Example

Parameters Passed by Reference

The MYMFC29B steps From Scratch

The Story

Debugging a DLL Component

IDispatch Interface Info

IDispatchEx Interface

Intro

After reading [Module 9](#), you should know what an interface is; you've already seen two standard COM interfaces, IUnknown and IClassFactory. Now you're ready for "applied" COM, or at least one aspect of it, Automation (formerly known as OLE Automation). You'll learn about the COM IDispatch interface, which enables C++ programs to communicate with Microsoft Visual Basic for Applications (VBA) programs and with programs written in other scripting languages. In addition, IDispatch is the key to getting your COM object onto a Web page. You'll use the MFC library implementation of IDispatch to write C++ Automation component and client programs. Both out-of-process components and in-process components are explored. But before jumping into C++ Automation programming, you need to know how the rest of the world writes programs. In this module, you'll get some exposure to VBA as it is implemented in Microsoft Excel. You'll run your C++ components from Excel, and you'll run Excel from a C++ client program.

Connecting C++ with Visual Basic for Applications (VBA)

Not all programmers for Microsoft Windows-based applications are going to be C++ programmers, especially if they have to learn the intricacies of COM theory. If you've been paying attention over the last few years, you've probably noticed a trend in which C++ programmers produce reusable modules. Programmers using higher-level languages (Visual Basic, VBA, and Web scripting languages, for example) consume those modules by integrating them into applications. You can participate in this programming model by learning how to make your software

Script-friendly. Automation is one tool available now that is supported by the Microsoft Foundation Class library. **ActiveX Controls** are another tool for C++/VBA integration and are very much a superset of Automation because both tools use the `IDispatch` interface. Using ActiveX Controls, however, might be overkill in many situations. Many applications, including Microsoft Excel 97, can support both Automation components and ActiveX controls. You'll be able to apply all that you learn about Automation when you write and use ActiveX controls.

Two factors are responsible for Automation's success. **First**, VBA (or VB Script) is now the programming standard in most Microsoft applications, including Microsoft Word, Microsoft Access, and Excel, not to mention Microsoft Visual Basic itself. All these applications support Automation, which means they can be linked to other Automation-compatible components, including those written in C++ and VBA. For example, you can write a C++ program that uses the text-processing capability of Word, or you can write a C++ matrix inversion component that can be called from a VBA macro in an Excel worksheet.

The **second** factor connected to Automation's success is that dozens of software companies provide Automation programming interfaces for their applications, mostly for the benefit of VBA programmers. With a little effort, you can run these applications from C++. You can, for example, write an MFC program that controls **Visio** drawing program.

Automation isn't just for C++ and VBA programmers. Software-tool companies are already announcing Automation-compatible, Basic-like languages that you can license for your own programmable applications. One version of Smalltalk even supports Automation.

Automation Clients and Automation Components

A clearly defined "**master-slave**" relationship is always present in an Automation communication dialog. The **master** is the **Automation client** and the **slave** is the **Automation component** (server). The client initiates the interaction by constructing a component object (it might have to load the component program) or by attaching to an existing object in a component program that is already running. The client then calls **interface functions** in the component and releases those interfaces when it's finished. Here are some interaction scenarios:

- A C++ Automation client uses a Microsoft or third-party application as a component. The interaction could trigger the execution of VBA code in the component application.
- A C++ Automation component is used from inside a Microsoft application (or a Visual Basic application), which acts as the Automation client. Thus, VBA code can construct and use C++ objects.
- A C++ Automation client uses a C++ Automation component.
- A Visual Basic program uses an Automation-aware application such as Excel. In this case, Visual Basic is the client and Excel is the component.

Microsoft Excel: A Better Visual Basic than Visual Basic

At the time that the first three editions of this book were written, Visual Basic worked as an Automation client, but you couldn't use it to create an Automation component. Since version 5.0, Visual Basic lets you write components too, even ActiveX controls. We originally used Excel instead of VB because Excel was the first Microsoft application to support VBA syntax and it could serve as both a client and a component. We decided to stick with Excel because C++ programmers who look down their noses at Visual Basic might be inclined to buy Excel (if only to track their software royalties).

We strongly recommend that you get a copy of Excel 97 (or a later version - 2000, 2003). This is a true 32-bit application and a part of the Microsoft Office suite. With this version of Excel, you can write VBA code in a separate location that accesses worksheet cells in an object-oriented manner. Adding visual programming elements, such as pushbuttons, is easy. Forget all you ever knew about the old spreadsheet programs that forced you to wedge macro code inside cells.

This module isn't meant to be an Excel tutorial, but we've included a simple Excel workbook. (A workbook is a file that can contain multiple worksheets plus separate VBA code.) This workbook demonstrates a VBA macro that executes from a pushbutton. You can use Excel to load [Demo.xls](#) or you can key in the example from scratch. Figure 1 shows the actual spreadsheet with the button and sample data.

In this spreadsheet, you highlight cells A4 through A9 and click the **Process Col** button. A VBA program iterates down the column and draws a hatched pattern on cells with numeric values greater than 10.

Figure 2 shows the macro code itself, which is "behind" the worksheet. In Excel 97/2000/2003 (this Tutorial uses Excel 2003), choose **Macro** from the **Tools** menu, and then choose **Visual Basic Editor**. **Alt-F11** is the shortcut. As you can see, you're working in the standard VBA 5.0 environment at this point.

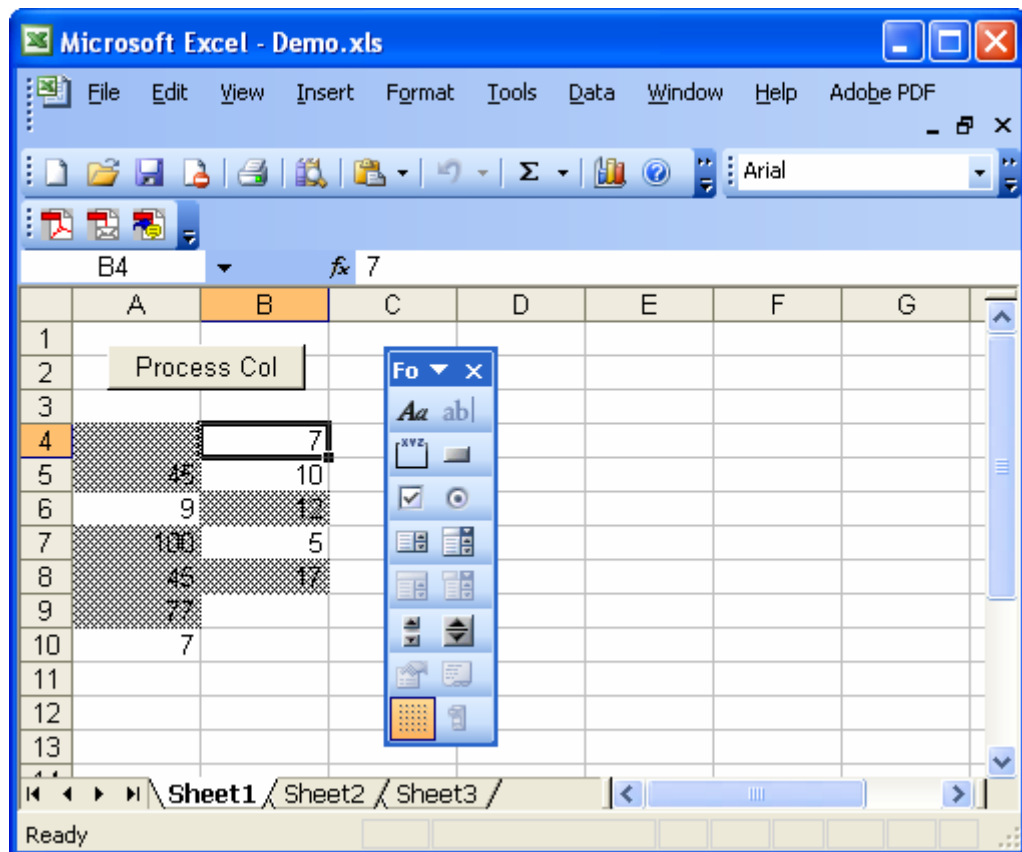


Figure 1: An Excel spreadsheet that uses VBA code.

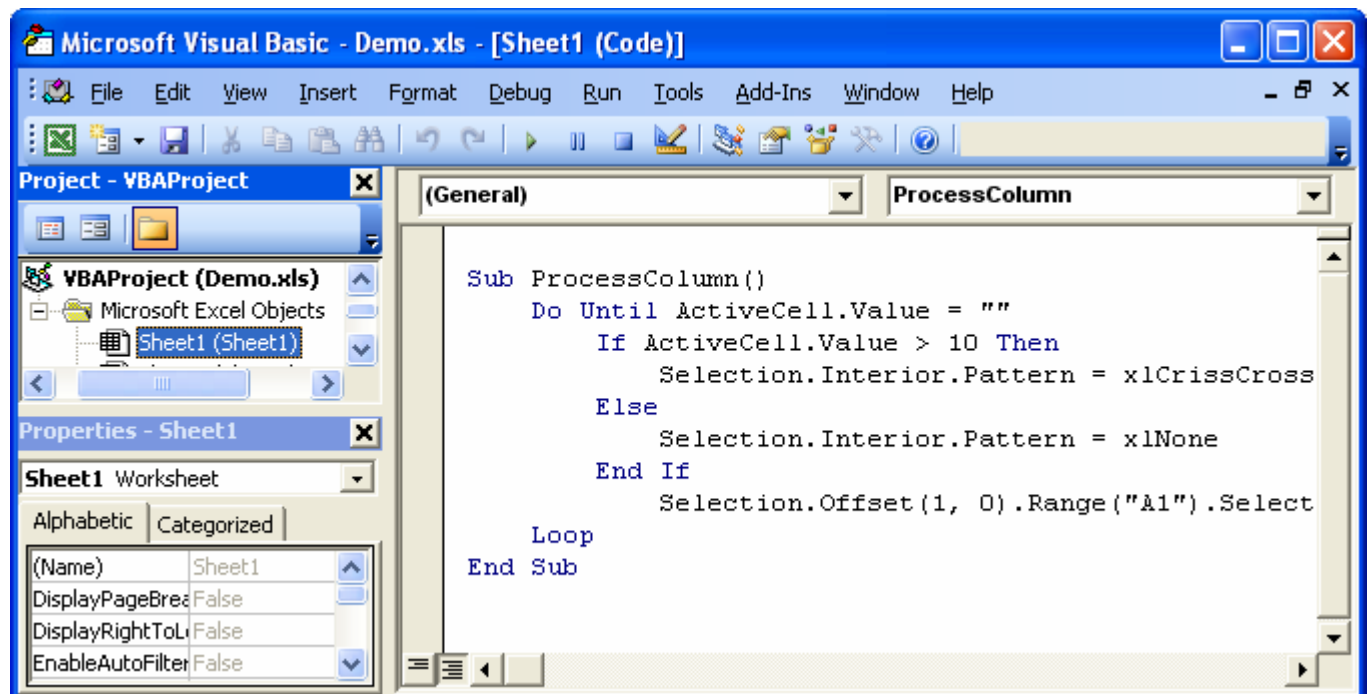


Figure 2: The VBA code for the Excel spreadsheet.

If you want to create the example yourself, follow these steps:

1. Start Excel with a new workbook, press Alt-F11 to invoke the Visual Basic Editor.

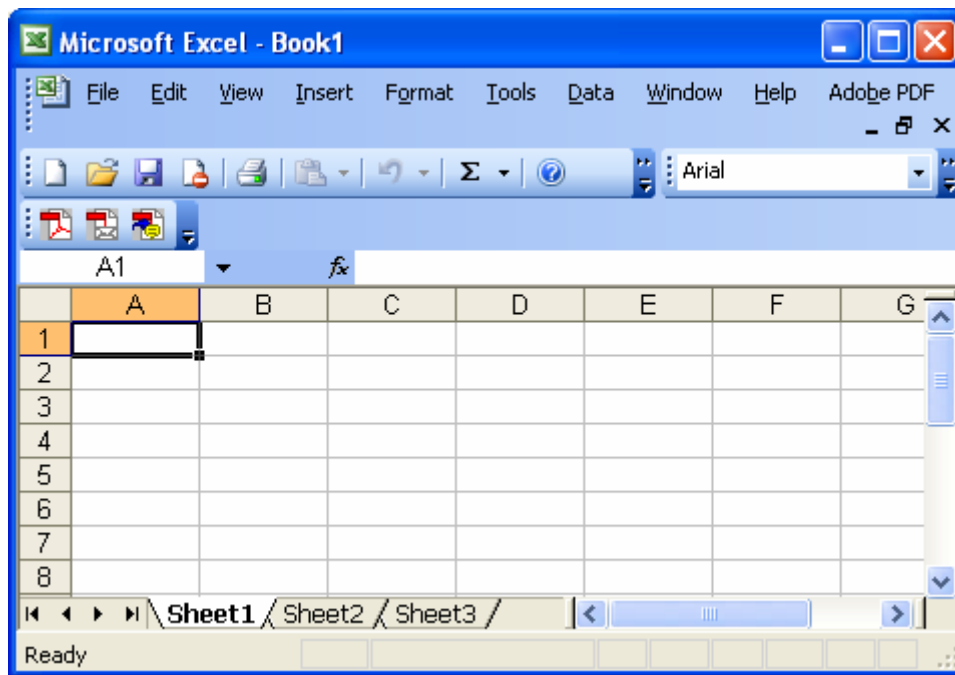


Figure 3: Excel workbook.

2. Then double-click **Sheet1** in the top left window.

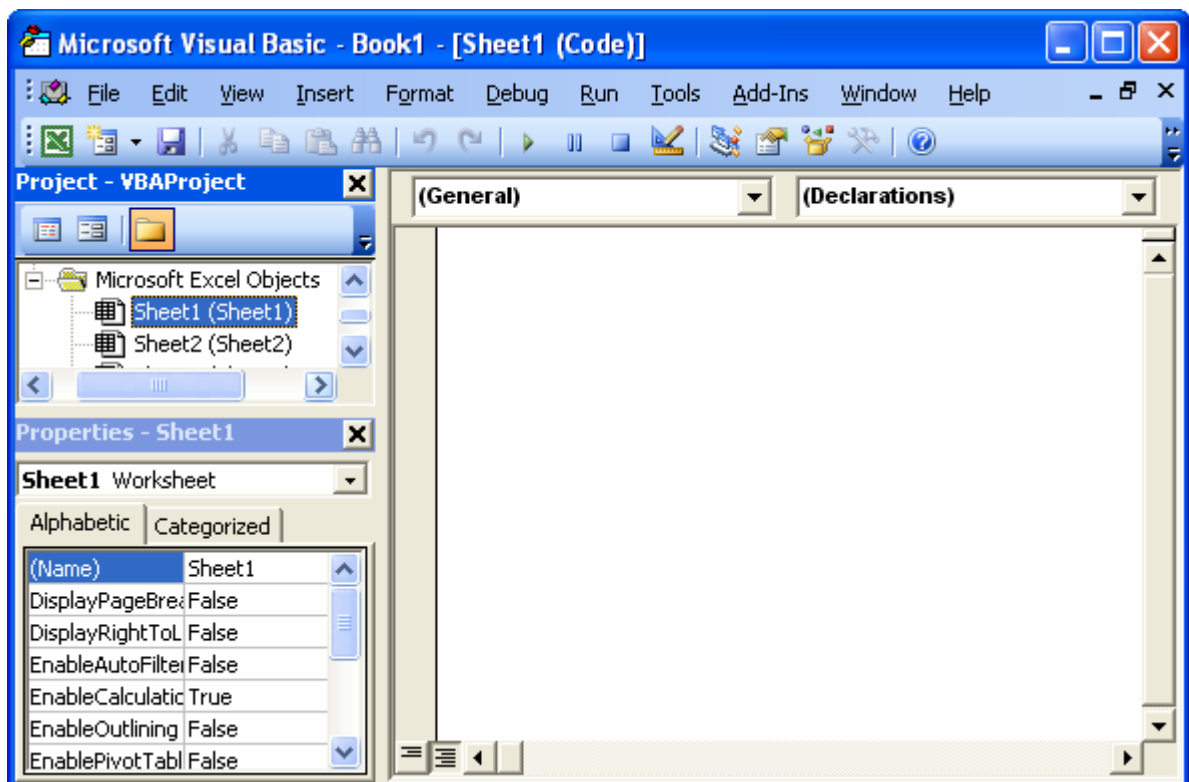


Figure 4: visual Basic editor.

3. Type in the macro code shown below.

```
Sub ProcessColumn()
    Do Until ActiveCell.Value = ""
        If ActiveCell.Value > 10 Then
            Selection.Interior.Pattern = xlCrissCross
        End If
        ActiveCell.Offset(1, 0).Select
    Loop
End Sub
```

```

Else
    Selection.Interior.Pattern = xlNone
End If
Selection.Offset(1, 0).Range("A1").Select
Loop
End Sub

```

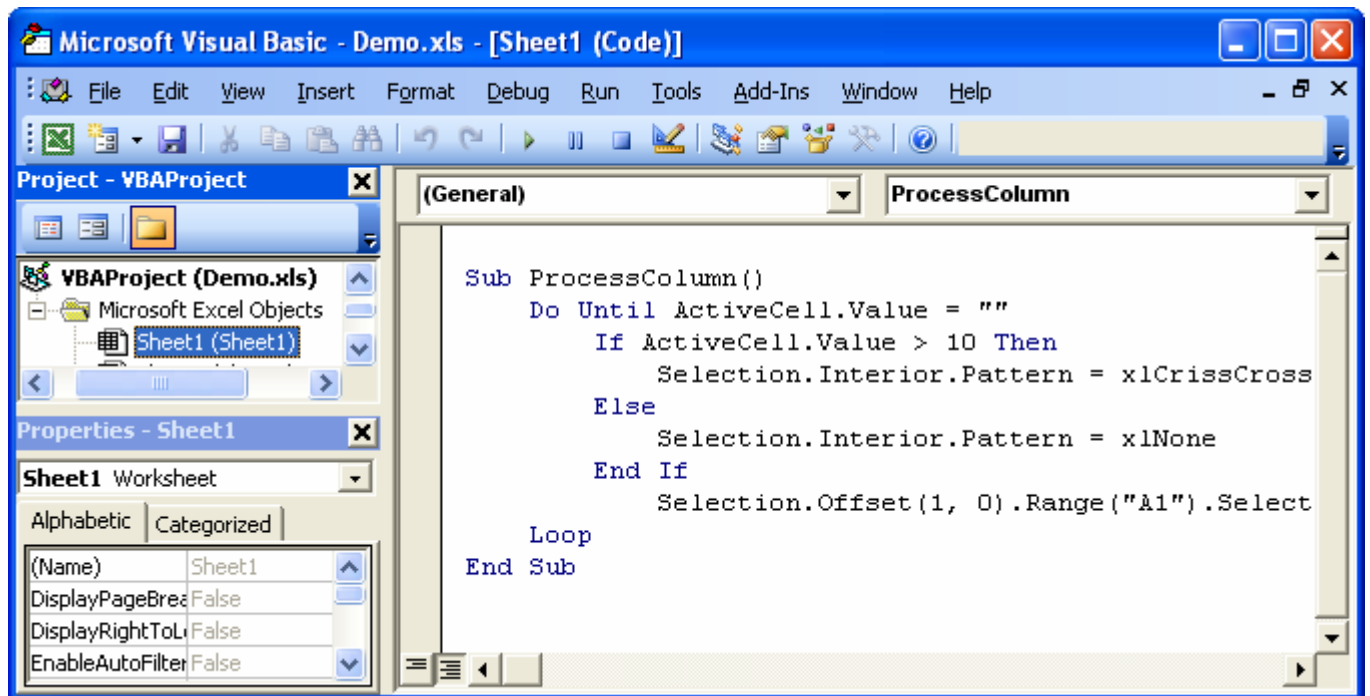


Figure 5: Typing the macro code.

- Return to the Excel window by choosing **Close And Return To Microsoft Excel** from the **File** menu as shown in Figure 6. Choose **Toolbars** from the **View** menu. Check **Forms** to display the **Forms** toolbar as shown in Figure 7. You can also access the list of toolbars by right-clicking on any existing toolbar.

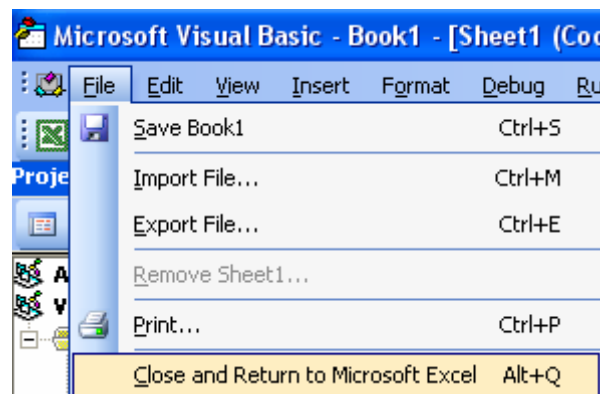


Figure 6: Closing the Visual Basic Editor and back to Excel.



Figure 7: Excel forms toolbar.

- Click the **Button** control, and then create the pushbutton by dragging the mouse in the upper-left corner of the worksheet. Assign the button to the **Sheet1.ProcessColumn** macro.

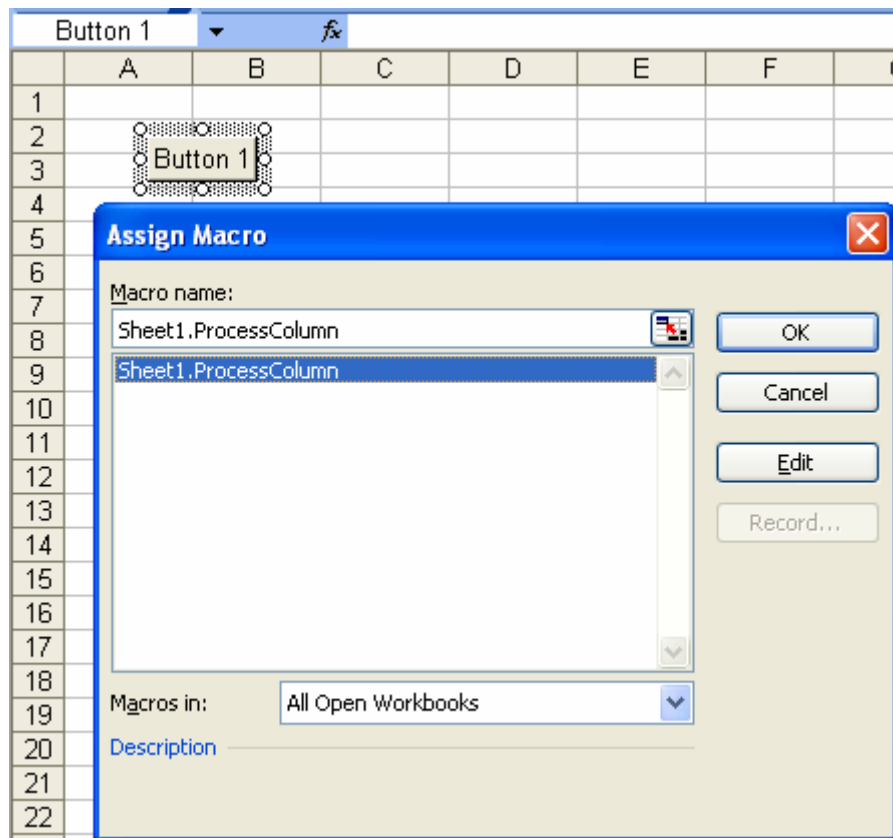


Figure 8: Attaching macro to button in Excel.

6. Size the pushbutton as needed, and type the caption **Process Col**, as shown in Figure 9.

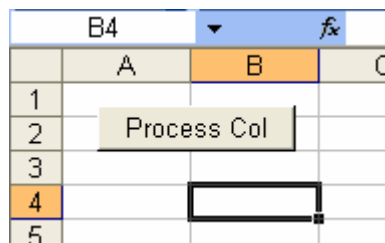


Figure 9: Button in Excel.

7. Type some numbers in the column starting at cell A4. Select the cells containing these numbers, and then click the **Process Col** button to test the program.

	A	B	C
1			
2		Process Col	
3			
4	30		
5	45		
6	9		
7	100		
8	45		
9	77		
10	7		
11			
12			

Figure 10: VBA in action.

Pretty easy, isn't it? Let's analyze an Excel VBA statement from the macro above:

```
Selection.Offset(1, 0).Range("A1").Select
```

The first element, `Selection`, is a property of an implied object, the Excel application. The `Selection` property in this case is assumed to be a `Range` object that represents a rectangular array of cells. The second element, `Offset`, is a property of the `Range` object that returns another `Range` object based on the two parameters. In this case, the returned `Range` object is the one-cell range that begins one row down from the original range. The third element, `Range`, is a property of the `Range` object that returns yet another range. This time it's the upper-left cell in the second range. Finally, the `Select` method causes Excel to highlight the selected cell and makes it the new `Selection` property of the application.

As the program iterates through the loop, the preceding statement moves the selected cell down the worksheet one row at a time. This style of programming takes some getting used to, but you can't afford to ignore it. The real value here is that you now have all the capabilities of the Excel spreadsheet and graphics engine available to you in a seamless programming environment.

Properties, Methods, and Collections

The distinction between a **property** and a **method** is somewhat artificial. Basically, a property is a value that can be both set and retrieved. You can, for example, set and get the `Selection` property for an Excel application.

Another example is Excel's `Width` property, which applies to many object types. Some Excel properties are read-only; most are read/write.

Properties don't officially have parameters, but some properties are indexed. The property index acts a lot like a parameter. It doesn't have to be an integer, and it can have more than one element (row and column, for example). You'll find many indexed properties in Excel's object model, and Excel VBA can handle indexed properties in Automation components.

Methods are more flexible than properties. They can have zero or many parameters, and they can either set or retrieve object data. Most frequently they perform some action, such as showing a window. Excel's `Select` method is an example of an action method.

The Excel object model supports collection objects. If you use the **Worksheets** property of the Application object, you get back a `Sheets` collection object, which represents all the worksheets in the active workbook. You can use the `Item` property (with an integer index) to get a specific `Worksheet` object from a `Sheets` collection or you can use an integer index directly on the collection.

The Problem That Automation Solves

You've already learned that a COM interface is the ideal way for Windows programs to communicate with one another, but you've also learned that designing your own COM interfaces is mostly impractical. Automation's general-purpose interface, `IDispatch`, serves the needs of both C++ and VBA programmers. As you might guess from your glimpse of Excel VBA, this interface involves **objects, methods, and properties**.

You can write COM interfaces that include functions with any parameter types and return values you specify. `IMotion` and `IVisual`, created in [Module 23](#), are some examples. If you're going to let VBA programmers in, however, you can't be fast and loose anymore. You can **solve the communication problem** with one interface that has a member function smart enough to accommodate methods and properties as defined by VBA. Needless to say, `IDispatch` has such a function: `Invoke()`. You use `IDispatch::Invoke` for COM objects that can be constructed and used in either C++ or VBA programs.

Now you're beginning to see what Automation does. It funnels all inter module communication through the `IDispatch::Invoke` function. How does a client first connect to its component? Because `IDispatch` is merely another COM interface, all the registration logic supported by COM comes into play. Automation components can be DLLs or EXEs, and they can be accessed over a network using **distributed COM** (DCOM).

The IDispatch Interface

`IDispatch` is the heart of Automation. It's fully supported by COM marshaling (that is, Microsoft has already marshaled it for you), as are all the other standard COM interfaces, and it's supported well by the MFC library. At the component end, you need a COM class with an `IDispatch` interface (plus the prerequisite class factory, of

course). At the client end, you use standard COM techniques to obtain an `IDispatch` pointer. As you'll see, the MFC library and the wizards take care of a lot of these details for you.

Remember that `Invoke()` is the principal member function of `IDispatch`. If you looked up `IDispatch::Invoke` in the Visual C++ online documentation, you'd see a really ugly set of parameters. Don't worry about those now. The MFC library steps in on both sides of the `Invoke()` call, using a data-driven scheme to call component functions based on dispatch map parameters that you define with macros.

`Invoke()` isn't the only `IDispatch` member function. Another function your controller might call is `GetIDsOfNames()`. From the VBA programmer's point of view, properties and methods have symbolic names, but C++ programmers prefer more efficient integer indexes. `Invoke` uses integers to specify properties and methods, so `GetIDsOfNames()` is useful at the start of a program for converting each name to a number if you don't know the index numbers at compile time. You've already seen that `IDispatch` supports symbolic names for methods. In addition, the interface supports symbolic names for a method's parameters. The `GetIDsOfNames()` function returns those parameter names along with the method name. Unfortunately, the MFC `IDispatch` implementation doesn't support named parameters.

Automation Programming Choices

Suppose you're writing an Automation component in C++. You've got some choices to make. Do you want an in-process component or an out-of-process component? What kind of user interface do you want? Does the component need a user interface at all? Can users run your EXE component as a stand-alone application? If the component is an EXE, will it be SDI or MDI? Can the user shut down the component program directly?

If your component is a DLL, COM linkage will be more efficient than it would be with an EXE component because no marshaling is required. Most of the time, your in-process Automation components won't have their own user interfaces, except for modal dialog boxes. If you need a component that manages its own child window, you should use an ActiveX control, and if you want to use a main frame window, use an out-of-process component. As with any 32-bit DLL, an Automation DLL is mapped into the client's process memory. If two client programs happen to request the same DLL, Windows loads and links the DLL twice. Each client is unaware that the other is using the same component.

With an EXE component, however, you must be careful to distinguish between a **component program** and a **component object**. When a client calls `IClassFactory::CreateInstance` to construct a component object, the component's class factory constructs the object, but COM might or might not need to start the component program.

Here are some scenarios:

1. The component's COM-creatable class is programmed to require a new process for each object constructed. In this case, COM starts a new process in response to the second and subsequent `CreateInstance()` calls, each of which returns an `IDispatch` pointer.
2. Here's a special case of scenario 1 above, specific to MFC applications. The component class is an MFC document class in an SDI application. Each time a client calls `CreateInstance()`, a new component process starts, complete with a document object, a view object, and an SDI main frame window.
3. The component class is programmed to allow multiple objects in a single process. Each time a client calls `CreateInstance()`, a new component object is constructed. There is only one component process, however.
4. Here's a special case of scenario 3 above, specific to MFC applications. The component class is an MFC document class in an MDI application. There is a single component process with one MDI main frame window. Each time a client calls `CreateInstance()`, a new document object is constructed, along with a view object and an MDI child frame window.

There's one more interesting case. Suppose a component EXE is running before the client needs it, and then the client decides to access a component object that already exists. You'll see this case with Excel. The user might have Excel running but minimized on the desktop, and the client needs access to Excel's one and only `Application` object. Here the client calls the COM function `GetActiveObject()`, which provides an interface pointer for an existing component object. If the call fails, the client can create the object with `CoCreateInstance()`.

For component object deletion, normal COM rules apply. Automation objects have reference counts, and they delete themselves when the client calls `Release()` and the reference count goes to 0. In an MDI component, if the Automation object is an MFC document, its destruction causes the corresponding MDI child window to close. In an SDI component, the destruction of the document object causes the component process to exit. The client is responsible for calling `Release()` for each `IDispatch` interface before the client exits. For EXE components,

COM will intervene if the client exits without releasing an interface, thus allowing the component process to exit. You can't always depend on this intervention, however, so be sure that your client cleans up its interfaces! With generic COM, a client application often obtains multiple interface pointers for a single component object. Look back at the **spaceship** example in [Module 23](#), in which the simulated COM component class had both an `IMotion` pointer and an `IVisual` pointer. With Automation, however, there's usually only a single (`IDispatch`) pointer per object. As in all COM programming, you must be careful to release all your interface pointers. In Excel, for example, many properties return an `IDispatch` pointer to new or existing objects. If you fail to release a pointer to an in-process COM component, the Debug version of the MFC library alerts you with a memory-leak dump when the client program exits.

The MFC `IDispatch` Implementation

The component program can implement its `IDispatch` interface in several ways. The most common of these pass off much of the work to the Windows COM DLLs by calling the COM function `CreateStdDispatch()` or by delegating the `Invoke()` call to the `ITypeInfo` interface, which involves the component's type library. A type library is a table, locatable through the Registry, which allows a client to query the component for the symbolic names of objects, methods, and properties. A client could, for example, contain a browser that allows the user to explore the component's capabilities.

The MFC library supports type libraries, but it doesn't use them in its implementation of `IDispatch`, which is instead driven by a dispatch map. MFC programs don't call `CreateStdDispatch()` at all, nor do they use a type library to implement `IDispatch::GetIDsOfNames`. This means that you can't use the MFC library if you implement a multilingual Automation component, one that supports English and German property and method names, for example. `CreateStdDispatch()` doesn't support multilingual components either.

Later in this module you'll learn how a client can use a type library, and you'll see how `AppWizard` and `ClassWizard` create and maintain type libraries for you. Once your component has a type library, a client can use it for browsing, independent of the `IDispatch` implementation.

An MFC Automation Component

Let's look at what happens in an MFC Automation component, in this case, a simplified version of the `MYMFC29C` alarm clock program that is discussed later in this module. In the MFC library, the `IDispatch` implementation is part of the `CCmdTarget` base class, so you don't need `INTERFACE_MAP` macros. You write an Automation component class, `CClock`, for example, derived from `CCmdTarget`. This class's CPP file contains `DISPATCH_MAP` macros:

```
BEGIN_DISPATCH_MAP(CClock, CCmdTarget)
    DISP_PROPERTY(CClock, "Time", m_time, VT_DATE)
    DISP_PROPERTY_PARAM(CClock, "Figure", GetFigure, SetFigure,
VT_VARIANT, VTS_I2)
    DISP_FUNCTION(CClock, "RefreshWin", Refresh, VT_EMPTY, VTS_NONE)
    DISP_FUNCTION(CClock, "ShowWin", ShowWin, VT_BOOL, VTS_I2)
END_DISPATCH_MAP()
```

Looks a little like an MFC message map, doesn't it? The `CClock` class header file contains related code, shown here:

```
public:
    DATE m_time;
    afx_msg VARIANT GetFigure(short n);
    afx_msg void SetFigure(short n, const VARIANT& vaNew);
    afx_msg void Refresh();
    afx_msg BOOL ShowWin(short n);
    DECLARE_DISPATCH_MAP()
```

What does all this stuff mean? It means that the `CClock` class has the following properties and methods.

Name	Type	Description
Time	Property	Linked directly to class data member <code>m_time</code> .
Figure	Property	Indexed property, accessed through member functions <code>GetFigure()</code> and

		SetFigure(): first parameter is the index; second (for SetFigure()) is the string value. The figures are the "XII," "III," "VI," and "IX" that appear on the clock face.
RefreshWin	Method	Linked to class member function Refresh() - no parameters or return value.
ShowWin	Method	Linked to class member function ShowWin() - short integer parameter, Boolean return value.

Table 1.

How does the MFC dispatch map relate to IDispatch and the Invoke() member function? The dispatch-map macros generate static data tables that the MFC library's Invoke() implementation can read. A controller gets an IDispatch pointer for CClock (connected through the CCmdTarget base class), and it calls Invoke() with an array of pointers as a parameter. The MFC library's implementation of Invoke(), buried somewhere inside CCmdTarget, uses the CClock dispatch map to decode the supplied pointers and either calls one of your member functions or accesses m_time directly.

As you'll see in the examples, ClassWizard can generate the Automation component class for you and help you code the dispatch map.

An MFC Automation Client Program

Let's move on to the client's end of the Automation conversation. How does an MFC Automation client program call Invoke()? The MFC library provides a base class COleDispatchDriver for this purpose. This class has a data member, m_lpDispatch, which contains the corresponding component's IDispatch pointer. To shield you from the complexities of the Invoke() parameter sequence, COleDispatchDriver has several member functions, including InvokeHelper(), GetProperty() and SetProperty(). These three functions call Invoke() for an IDispatch pointer that links to the component. The COleDispatchDriver() object incorporates the IDispatch pointer.

Let's suppose our client program has a class CClockDriver, derived from COleDispatchDriver(), that drives CClock objects in an Automation component. The functions that get and set the Time property are shown here.

```
DATE CClockDriver::GetTime()
{
    DATE result;
    GetProperty(1, VT_DATE, (void*)&result);
    return result;
}

void CClockDriver::SetTime(DATE propVal)
{
    SetProperty(1, VT_DATE, propVal);
}
```

Here are the functions for the indexed **Figure** property:

```
VARIANT CClockDriver::GetFigure(short i)
{
    VARIANT result;
    static BYTE parms[] = VTS_I2;
    InvokeHelper(2, DISPATCH_PROPERTYGET, VT_VARIANT, (void*)&result,
parms, i);
    return result;
}

void CClockDriver::SetFigure(short i, const VARIANT& propVal)
{
    static BYTE parms[] = VTS_I2 VTS_VARIANT;
    InvokeHelper(2, DISPATCH_PROPERTYPUT, VT_EMPTY, NULL, parms, i,
&propVal);
}
```

```
}
```

And finally, here are the functions that access the component's methods:

```
void CClockDriver::RefreshWin()
{
    InvokeHelper(3, DISPATCH_METHOD, VT_EMPTY, NULL, NULL);
}

BOOL CClockDriver::ShowWin(short i)
{
    BOOL result;
    static BYTE parms[] = VTS_I2;
    InvokeHelper(4, DISPATCH_METHOD, VT_BOOL, (void*)&result, parms,
i);
    return result;
}
```

The function parameters identify the **property** or **method**, its **return value**, and its **parameters**. You'll learn about dispatch function parameters later, but for now take special note of the first parameter for the `InvokeHelper()`, `GetProperty()` and `SetProperty()` functions. This is the unique integer index, or dispatch ID (DISPID), for the property or method. Because you're using compiled C++, you can establish these IDs at compile time. If you're using an MFC Automation component with a dispatch map, the indexes are determined by the map sequence, beginning with 1. If you don't know a component's dispatch indexes, you can call the `IDispatch` member function `GetIDsOfNames()` to convert the symbolic property or method names to integers. The following illustration shows the interactions between the client (or controller) and the component.

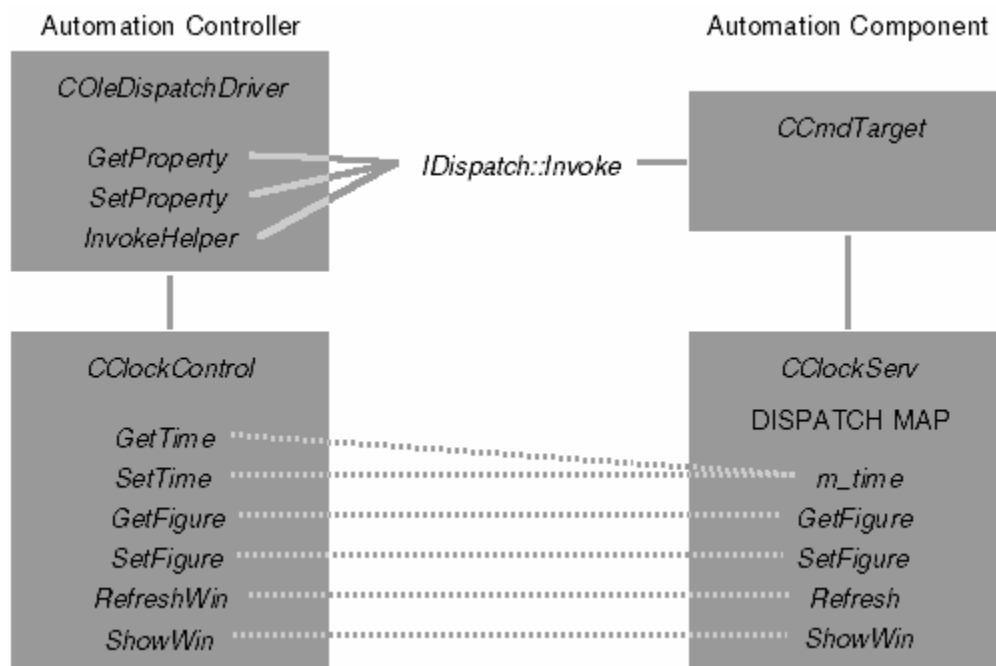


Figure 11: Interaction between client and component.

The solid lines show the actual connections through the MFC base classes and the `Invoke()` function. The dotted lines represent the resulting logical connections between client class members and component class members. Most Automation components have a **binary type library file** with a **TLB** extension. ClassWizard can access this type library file to generate a class derived from `COleDispatchDriver`. This generated controller class contains member functions for all the component's methods and properties with hard-coded dispatch IDs. Sometimes you need to do some surgery on this generated code, but that's better than writing the functions from scratch. After you have generated your driver class, you embed an object of this class in your client application's view class (or in another class) like this:

```
CClockDriver m_clock;
```

Then you ask COM to create a clock component object with this statement:

```
m_clock.CreateDispatch("Mymfc29C.Document");
```

Now you're ready to call the dispatch driver functions:

```
m_clock.SetTime(COleDateTime::GetCurrentTime());
m_clock.RefreshWin();
```

When the `m_clock` object goes out of scope, its destructor releases the `IDispatch` pointer.

An Automation Client Program Using the Compiler's `#import` Directive

Now there's an entirely new way of writing Automation client programs. Instead of using ClassWizard to generate a class derived from `COleDispatchDriver`, you use the compiler to generate header and implementation files directly from a component's type library. For the clock component, your client program contains the following statement:

```
#import "..\mymfc29C\debug\mymfc29C.tlb" rename_namespace("ClockDrv")
using namespace ClockDrv;
```

The compiler then generates (and processes) two files, **mymfc29C.tlh** and **mymfc29C.tli**, in the project's **Debug** or **Release** subdirectory. The **TLH** file contains the `IMymfc29C` clock driver class declaration plus this smart pointer declaration:

```
_COM_SMARTPTR_TYPEDEF(IMymfc29C, __uuidof(IDispatch));
```

The `_COM_SMARTPTR_TYPEDEF` macro generates the `IMymfc29CPtr` pointer type, which encapsulates the component's `IDispatch` pointer. The **Tli** file contains inline implementations of member functions, some of which are shown in the following code:

```
inline HRESULT IMymfc29C::RefreshWin()
{
    return _com_dispatch_method(this, 0x4, DISPATCH_METHOD, VT_EMPTY,
    NULL, NULL);
}

inline DATE IMymfc29C::GetTime()
{
    DATE _result;
    _com_dispatch_propget(this, 0x1, VT_DATE, (void*)&_result);
    return _result;
}

inline void IMymfc29C::PutTime(DATE _val)
{
    _com_dispatch_propput(this, 0x1, VT_DATE, _val);
}
```

Note the similarity between these functions and the `COleDispatchDriver()` member functions you've already seen. The functions `_com_dispatch_method()`, `_com_dispatch_propget()` and `_com_dispatch_propput()` are in the runtime library. In your Automation client program, you declare an embedded smart pointer member in your view class (or in another class) like this:

```
IMymfc29CPtr m_clock;
```

Then you create a clock component object with this statement:

```
m_clock.CreateInstance(__uuidof(Document));
```

Now you're ready to use the `IMymfc29CPtr` class's overloaded `→` operator to call the member functions defined in the TLI file:

```
m_clock->PutTime(ColeDateTime::GetCurrentTime());
m_clock->RefreshWin();
```

When the `m_clock` smart pointer object goes out of scope, its destructor calls the `COM Release()` function. The `#import` directive is the future of COM programming. With each new version of Visual C++, you'll see COM features moving into the compiler, along with the document-view architecture itself.

The VARIANT Type

No doubt you've noticed the `VARIANT` type used in both Automation **client** and **component** functions in the previous example. `VARIANT` is an all-purpose data type that `IDispatch::Invoke` uses to transmit parameters and return values. The `VARIANT` type is the natural type to use when exchanging data with VBA. Let's look at a **simplified version** of the `VARIANT` definition in the Windows header files.

```
struct tagVARIANT {
    VARTYPE vt; // unsigned short integer type code
    WORD wReserved1, wReserved2, wReserved3;
    union {
        short      iVal;           // VT_I2  short integer
        long       lVal;           // VT_I4  long integer
        float     fltVal;         // VT_R4  4-byte float
        double     dblVal;         // VT_R8  8-byte IEEE float
        DATE       date;           // VT_DATE stored as dbl
                                   // date.time
        CY         vtCY;           // VT_CY 64-bit integer
        BSTR       bstrVal;        // VT_BSTR
        IUnknown*  punkVal;        // VT_UNKNOWN
        IDispatch* pdispVal;       // VT_DISPATCH
        short*     piVal;          // VT_BYREF | VT_I2
        long*      plVal;          // VT_BYREF | VT_I4
        float*     pfltVal;        // VT_BYREF | VT_R4
        double*    pdblVal;        // VT_BYREF | VT_R8
        DATE*      pdate;         // VT_BYREF | VT_DATE
        CY*        pvtCY;         // VT_BYREF | VT_CY
        BSTR*       pbstrVal;      // VT_BYREF | VT_BSTR
    }
};

typedef struct tagVARIANT VARIANT;
```

As you can see, the `VARIANT` type is a [C structure](#) that contains a type code `vt`, some reserved bytes, and a big union of types that you already know about. If `vt` is `VT_I2`, for example, you would read the `VARIANT`'s value from `iVal`, which contains a 2-byte integer. If `vt` is `VT_R8`, you would read this value from `dblVal`, which contains an 8-byte real value.

A `VARIANT` object can contain actual data or a pointer to data. If `vt` has the `VT_BYREF` bit set, you must access a pointer in `piVal`, `plVal`, and so on. Note that a `VARIANT` object can contain an `IUnknown` pointer or an `IDispatch` pointer. This means that you can pass a complete COM object using an Automation call, but if you want VBA to process that object, its class should have an `IDispatch` interface.

Strings are special. The **BSTR** type is yet another way to represent character strings. A `BSTR` variable is a pointer to a zero-terminated character array with a character count in front. A `BSTR` variable could, therefore, contain binary characters, including zeros. If you had a `VARIANT` object with `vt = VT_BSTR`, memory would look like this.

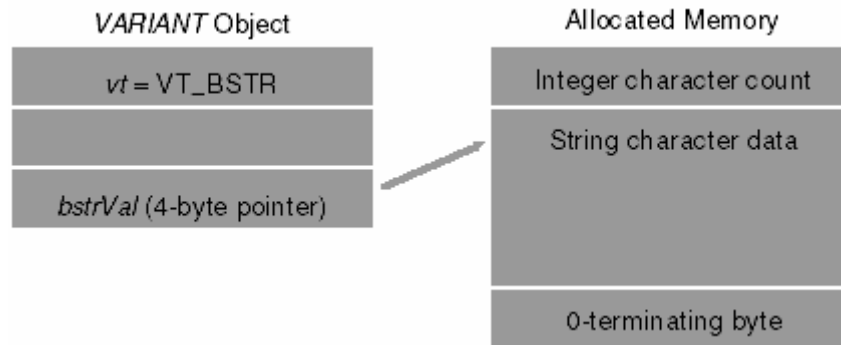


Figure 12: VARIANT data type and memory arrangement.

Because the string has a terminating 0, you can use `bstrVal` as though it was an ordinary char pointer, but you have to be very, very careful about memory cleanup. You can't simply delete the string pointer, because the allocated memory begins with the character count. Windows provides the `SysAllocString()` and `SysFreeString()` functions for allocating and deleting BSTR objects.

`SysAllocString()` is another COM function that takes a wide string pointer as a parameter. This means that all BSTRs contain wide characters, even if you haven't defined `_UNICODE`. Be careful.

Windows supplies some useful functions for `VARIANT`s, including those shown in the following table. If a `VARIANT` contains a BSTR, these functions ensure that memory is allocated and cleared properly. The `VariantInit()` and `VariantClear()` functions set `vt` to `VT_EMPTY`. All the variant functions are global functions and take a `VARIANT*` parameter.

Function	Description
<code>VariantInit()</code>	Initializes a <code>VARIANT</code> .
<code>VariantClear()</code>	Clears a <code>VARIANT</code> .
<code>VariantCopy()</code>	Frees memory associated with the destination <code>VARIANT</code> and copies the source <code>VARIANT</code> .
<code>VariantCopyInd()</code>	Frees the destination <code>VARIANT</code> and performs any indirection necessary to copy the source <code>VARIANT</code> .
<code>VariantChangeType()</code>	Changes the type of the <code>VARIANT</code> .

Table 2.

The COleVariant Class

Writing a C++ class to wrap the `VARIANT` structure makes a lot of sense. Constructors can call `VariantInit()`, and the destructor can call `VariantClear()`. The class can have a constructor for each standard type, and it can have copy constructors and assignment operators that call `VariantCopy()`. When a variant object goes out of scope, its destructor is called and memory is cleaned up automatically.

Well, the MFC team created just such a class, mostly for use in the Data Access Objects (DAO) subsystem. It works well in Automation clients and components, however. A simplified declaration is shown here.

```

class COleVariant : public tagVARIANT
{
// Constructors
public:
    COleVariant();

    COleVariant(const VARIANT& varSrc);
    COleVariant(const COleVariant& varSrc);

    COleVariant(LPCTSTR lpszSrc);
    COleVariant(CString& strSrc);

    COleVariant(BYTE nSrc);
  
```

```

    ColeVariant(short nSrc, VARTYPE vtSrc = VT_I2);
    ColeVariant(long lSrc, VARTYPE vtSrc = VT_I4);

    ColeVariant(float fltSrc);
    ColeVariant(double dblSrc);
    ColeVariant(const COleDateTime& dateSrc);
// Destructor
    ~ColeVariant(); // deallocates BSTR
// Operations
public:
    void Clear(); // deallocates BSTR
    VARIANT Detach(); // more later
    void ChangeType(VARTYPE vartype, LPVARIANT pSrc = NULL);
};

```

In addition, the `CArchive` and `CDumpContext` classes have comparison operators, assignment operators, conversion operators, and friend insertion/extraction operators.

Now let's see how the `ColeVariant` class helps us write the component's `GetFigure()` function that you previously saw referenced in the sample dispatch map. Assume that the component stores strings for four figures in a class data member:

```

private:
    CString m_strFigure[4];

```

Here's what we'd have to do if we used the `VARIANT` structure directly:

```

VARIANT CClock::GetFigure(short n)
{
    VARIANT vaResult;
    ::VariantInit(&vaResult);
    vaResult.vt = VT_BSTR;
    // CString::AllocSysString creates a BSTR
    vaResult.bstrVal =
m_strFigure[n].AllocSysString();
    return vaResult; // Copies vaResult without copying BSTR
                    // BSTR still must be freed later
}

```

Here's the equivalent, with a `ColeVariant` return value:

```

VARIANT CClock::GetFigure(short n)
{
    return ColeVariant(m_strFigure[n]).Detach();
}

```

Calling the `ColeVariant::Detach` function is critical here. The `GetFigure()` function is constructing a temporary object that contains a pointer to a BSTR. That object gets bitwise-copied to the return value. If you didn't call `Detach()`, the `ColeVariant` destructor would free the BSTR memory and the calling program would get a `VARIANT` that contained a pointer to nothing.

A component's variant dispatch function parameters are declared as `const VARIANT&`. You can always cast a `VARIANT` pointer to a `ColeVariant` pointer inside the function. Here's the `SetFigure()` function:

```

void CClock::SetFigure(short n, const VARIANT& vaNew)
{
    ColeVariant vaTemp;
    vaTemp.ChangeType(VT_BSTR, (COleVariant*) &vaNew);
    m_strFigure[n] = vaTemp.bstrVal;
}

```


Remember that all BSTRs contain wide characters. The CString class has a constructor and an assignment operator for the LPCWSTR (wide-character pointer) type. Thus, the m_strFigure string will contain single-byte characters, even though bstrVal points to a wide-character array.

Client dispatch function variant parameters are also typed as const VARIANT&. You can call those functions with either a VARIANT or a COleVariant object. Here's an example of a call to the CClockDriver::SetFigure function:

```
pClockDriver->SetFigure(0, COleVariant("XII"));
```

Visual C++ 5.0 added two new classes for BSTRs and VARIANTS. These classes are independent of the MFC library: _bstr_t and _variant_t. The _bstr_t class encapsulates the BSTR data type; the _variant_t class encapsulates the VARIANT type. Both classes manage resource allocation and de-allocation. For more information on these classes, see the online documentation.

Parameter and Return Type Conversions for Invoke ()

All IDispatch::Invoke parameters and return values are processed internally as VARIANTS. Remember that! The MFC library implementation of Invoke () is smart enough to convert between a VARIANT and whatever type you supply (where possible), so you have some flexibility in declaring parameter and return types. Suppose, for example, that your controller's GetFigure () function specifies the return type BSTR. If a component returns an int or a long, all is well: COM and the MFC library convert the number to a string. Suppose your component declares a long parameter and the controller supplies an int. Again, no problem.

An MFC library Automation client specifies the expected return type as a VT_ parameter to the COleDispatchDriver functions GetProperty (), SetProperty (), and InvokeHelper (). An MFC library Automation component specifies the expected parameter types as VTS_ parameters in the DISP_PROPERTY and DISP_FUNCTION macros.

Unlike C++, **VBA is not a strongly typed language**. VBA variables are often stored internally as VARIANTS. Take an Excel spreadsheet cell value, for example. A spreadsheet user can type a text string, an integer, a floating-point number, or a date/time in a cell. VBA treats the cell value as a VARIANT and returns a VARIANT object to an Automation client. If your client function declares a VARIANT return value, it can test vt and process the data accordingly.

VBA uses a date/time format that is distinct from the MFC library CTime class. Variables of type DATE hold both the date and the time in one double value. The fractional part represents time (.25 is 6:00 AM), and the whole part represents the date (number of days since December 30, 1899). The MFC library provides a COleDateTime class that makes dates easy to deal with. You could construct a date this way:

```
COleDateTime date(2005, 10, 1, 18, 0, 0);
```

The above declaration initializes the date to October 1, 2005, at 6:00 PM. The COleVariant class has an assignment operator for COleDateTime, and the COleDateTime class has member functions for extracting date/time components. Here's how you print the time:

```
TRACE("time = %d:%d:%d\n", date.GetHour(), date.GetMinute(),  
date.GetSecond());
```

If you have a variant that contains a DATE, you use the COleVariant::ChangeType function to convert a date to a string, as shown here:

```
COleVariant vaTimeDate = date;  
COleVariant vaTemp;  
vaTemp.ChangeType(VT_BSTR, &vaTimeDate);  
CString str = vaTemp.bstrVal;  
TRACE("date = %s\n", str);
```

One last item concerning Invoke () parameters: a dispatch function can have optional parameters. If the component declares trailing parameters as VARIANTS, the client doesn't have to supply them. If the client calls the function without supplying an optional parameter, the VARIANT object's vt value on the component end is VT_ERROR.

Automation Examples

The remainder of this module presents five sample programs. The first three programs are Automation components, an EXE component with no user interface, a DLL component, and a multi-instance SDI EXE component. Each of these component programs comes with a Microsoft Excel driver workbook file. The fourth sample program is an MFC Automation client program that drives the three components and also runs Excel using the `COleDispatchDriver` class. The last sample is a client program that uses the C++ `#import` directive instead of the MFC `COleDispatchDriver` class.

The MYMFC29A Automation Component EXE Example: No User Interface

The Visual C++ [Autoclik](#) MSDN's example is a good demonstration of an MDI framework application with the **document object** as the **Automation component**. The MYMFC29A example is different from the Autoclik example because MYMFC29A has no user interface. There is one Automation-aware class, and in the first version of the program, a single process supports the construction of multiple **Automation component** objects. In the second version, a new process starts up each time an **Automation client** creates an object. The MYMFC29A example represents a typical use of Automation. A C++ component implements financial transactions. VBA programmers can write User-interface-intensive applications that rely on the audit rules imposed by the Automation component. A production component program would probably use a database, but MYMFC29A is simpler. It implements a bank account with two methods, `Deposit` and `Withdrawal`, and one read-only property, `Balance`. Obviously, `Withdrawal` can't permit withdrawals that make the balance negative. You can use Excel to control the component, as shown in Figure 13.

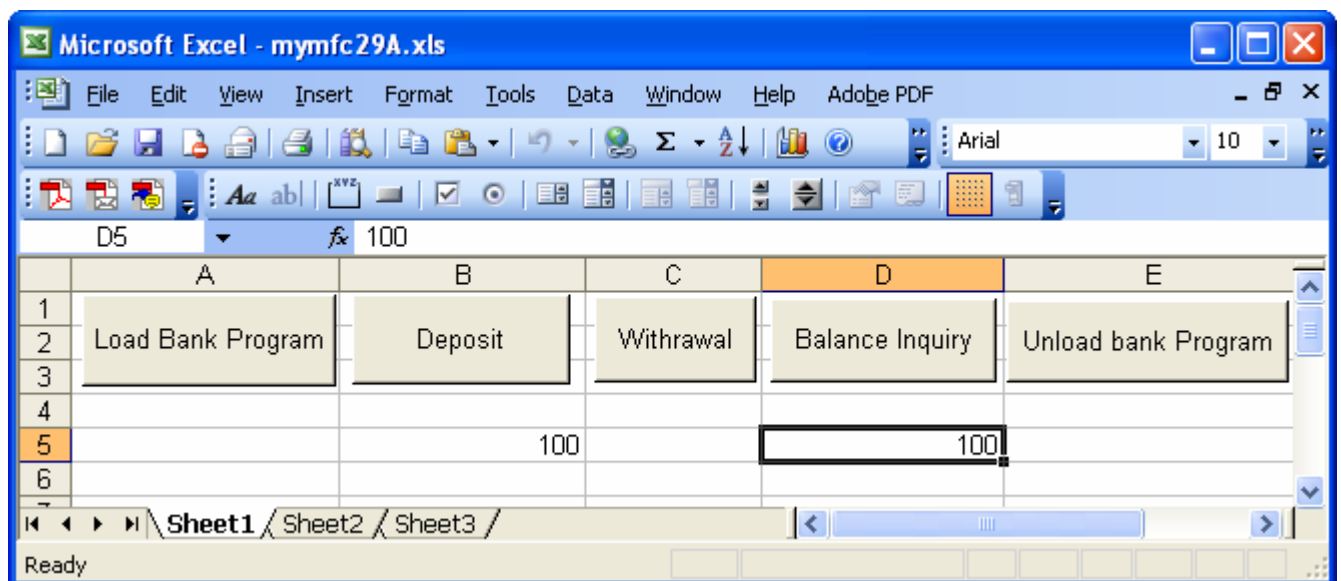


Figure 13: This Excel workbook is controlling the MYMFC29A component.

Here are the steps for creating the MYMFC29A program from scratch:

Run AppWizard to create the MYMFC29A project in the `\mfcproject\mymfc29A` directory or any directory that you have designated for your project.

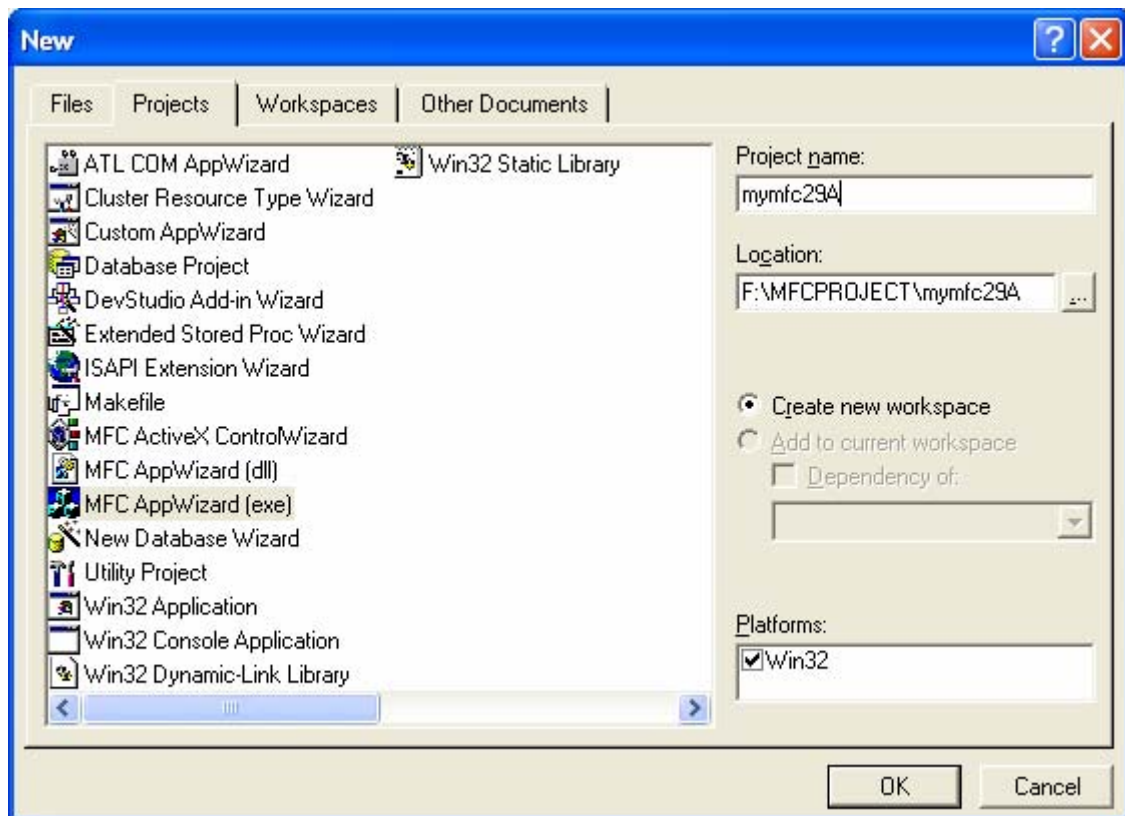


Figure 14: Visual C++ new project dialog, creating MYMFC29A dialog based project.

Select the **Dialog Based** option (Step 1).

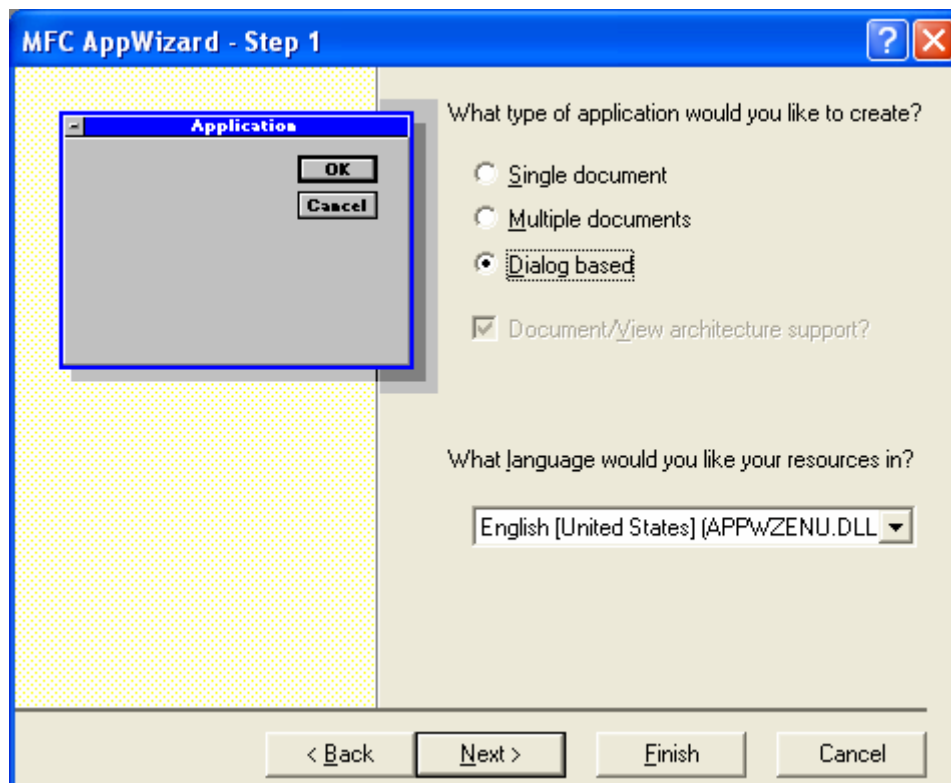


Figure 15: MYMFC29A - AppWizard step 1 of 4.

Deselect all options in Step 2, and accept the remaining default settings. This is the simplest application that AppWizard can generate.

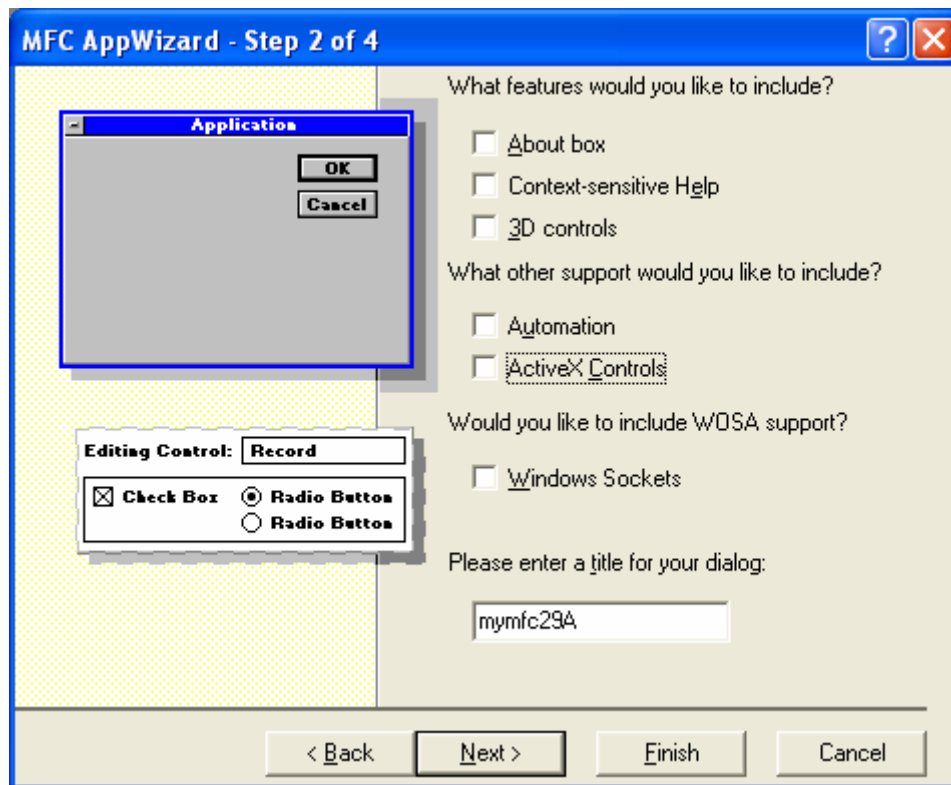


Figure 16: MYMFC29A - AppWizard step 2 of 4.

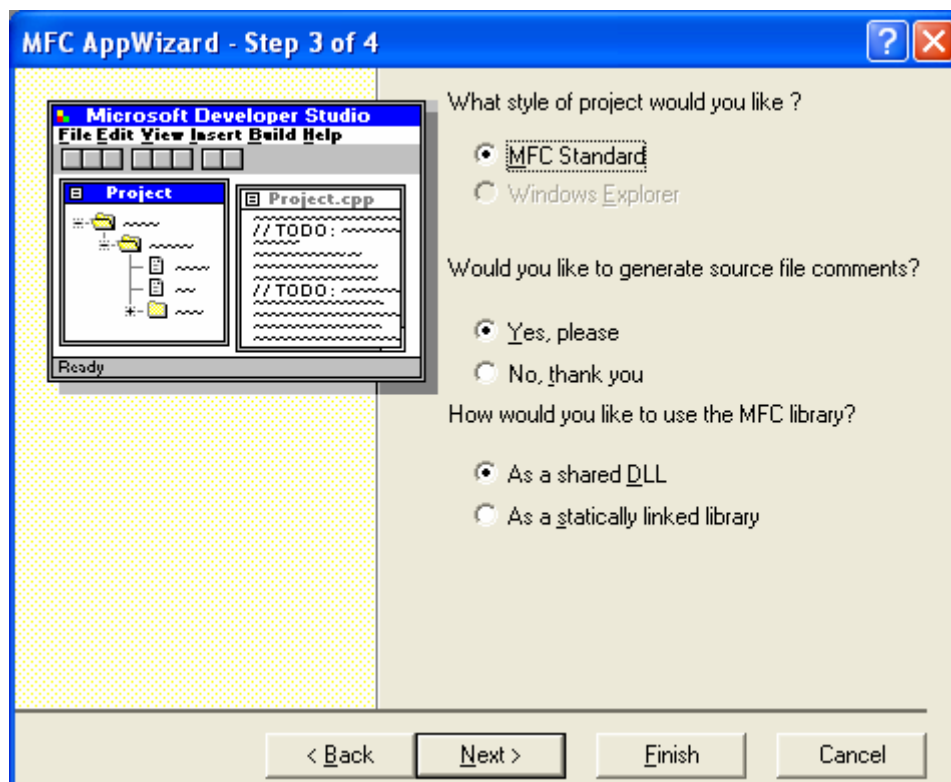


Figure 17: MYMFC29A - AppWizard step 3 of 4.

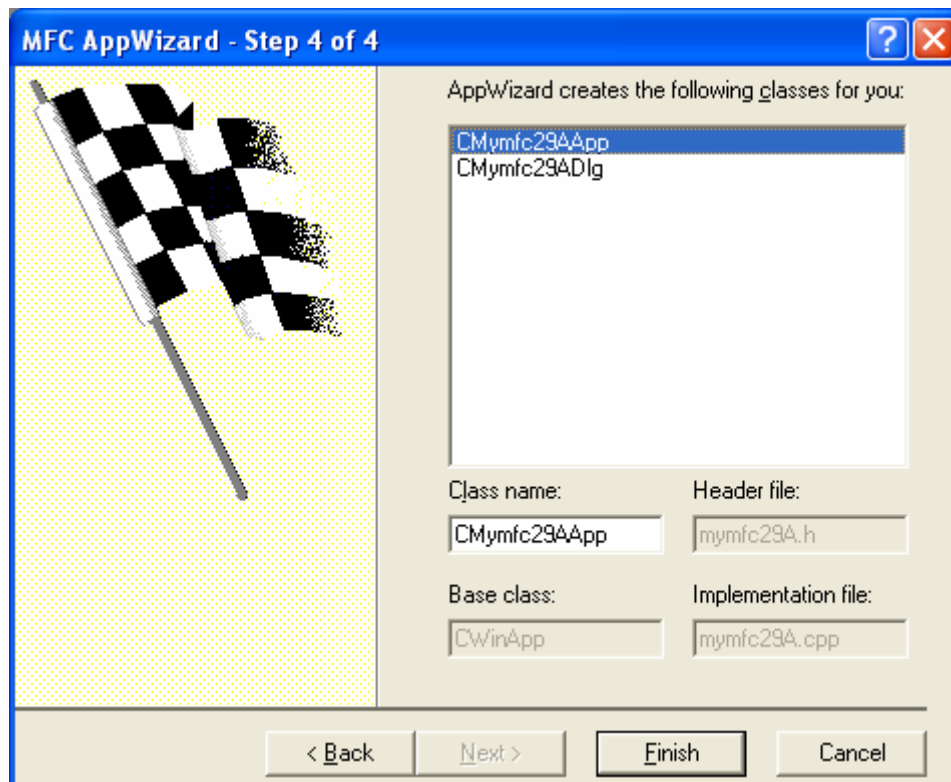


Figure 18: MYMFC29A - AppWizard step 4 of 4.

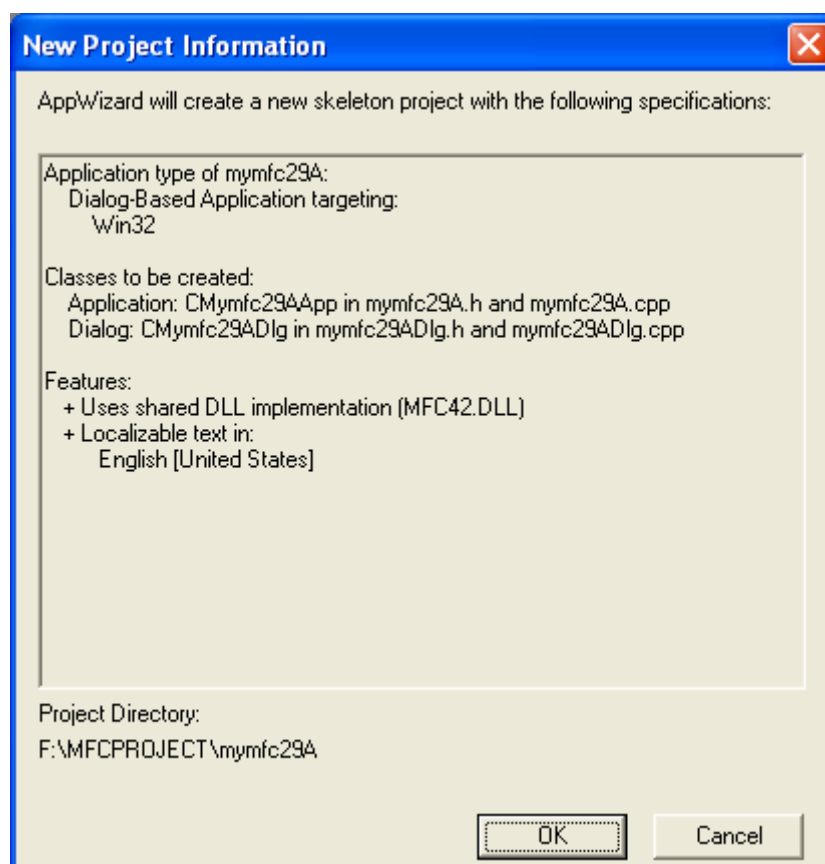


Figure 19: MYMFC29A project summary.

Eliminate the dialog class from the project. Using Windows Explorer or the command-line prompt, delete the files **mymfc29ADlg.cpp** and **mymfc29ADlg.h**. Remove **mymfc29ADlg.cpp** and **mymfc29ADlg.h** from the project by deleting them from the project's Workspace window (**FileView**) by selecting the file and using the **Edit Delete** menu.

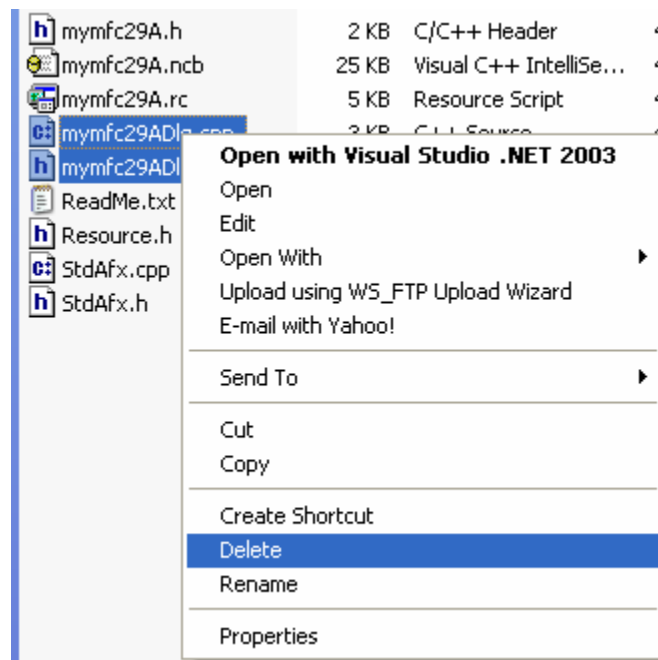


Figure 20: Deleting unnecessary project files.

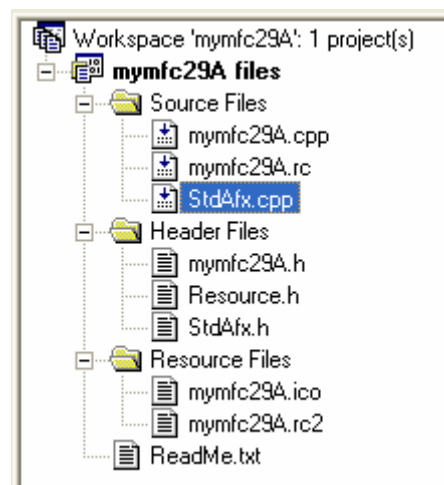


Figure 21: MYMFC29A files.

Edit **mymfc29A.cpp**. Remove the dialog `#include` (or commented out) and remove all dialog-related code from the `InitInstance()` function. In **ResourceView**, delete the `IDD_MYMFC29A_DIALOG` dialog resource template.

```
#include "stdafx.h"
#include "mymfc29A.h"
// #include "mymfc29ADlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif
```

Listing 1.

```
// CMyMfc29AApp initialization
BOOL CMyMfc29AApp::InitInstance()
{
    |
}

```

Listing 2.

Add code to enable Automation. Add this line in **StdAfx.h**:

```
#include <afxdisp.h>

#endif // _AFX_NO_AFXCMN_SUPPORT
#include <afxdisp.h>
//{{AFX_INSERT_LOCATION}}

```

Listing 3.

Then, edit the `InitInstance()` function (in **mymfc29A.cpp**) to look something like this:

```
BOOL CMyMfc29AApp::InitInstance()
{
    AfxOleInit();
    if(RunEmbedded() || RunAutomated())
    {
        // component started by COM
        COleTemplateServer::RegisterAll();
        return TRUE;
    }
    // Component is being run directly by the user
    COleObjectFactory::UpdateRegistryAll();
    AfxMessageBox("Bank component is registered");
    return FALSE;
}

// CMyMfc29AApp initialization
BOOL CMyMfc29AApp::InitInstance()
{
    AfxOleInit();
    if(RunEmbedded() || RunAutomated())
    {
        // component started by COM
        COleTemplateServer::RegisterAll();
        return TRUE;
    }
    // Component is being run directly by the user
    COleObjectFactory::UpdateRegistryAll();
    AfxMessageBox("Bank component is registered");
    return FALSE;
}

```

Listing 4.

Rebuild ClassWizard database. Delete the **CLW** file and invoke the ClassWizard.

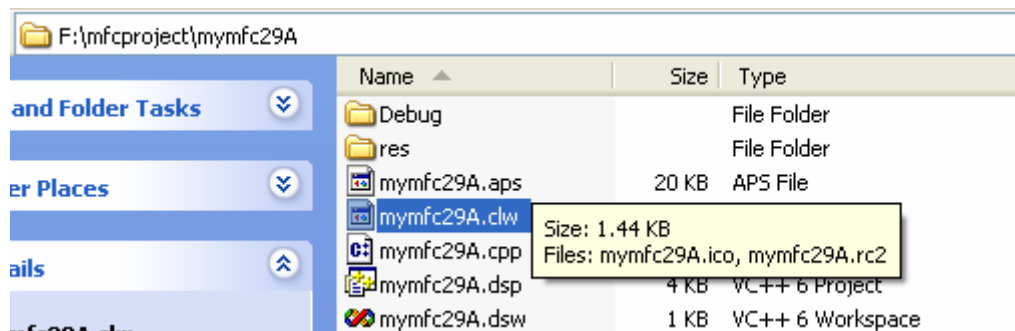


Figure 22: Deleting CLW, a ClassWizard database file.

Just click the **Yes** button.

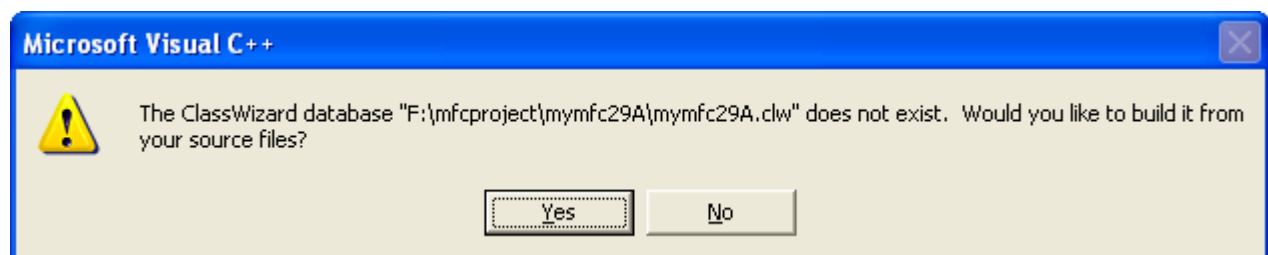


Figure 23: ClassWizard database rebuilding conformation prompt.

And **OK** button to include those files for ClassWizard database rebuilding.

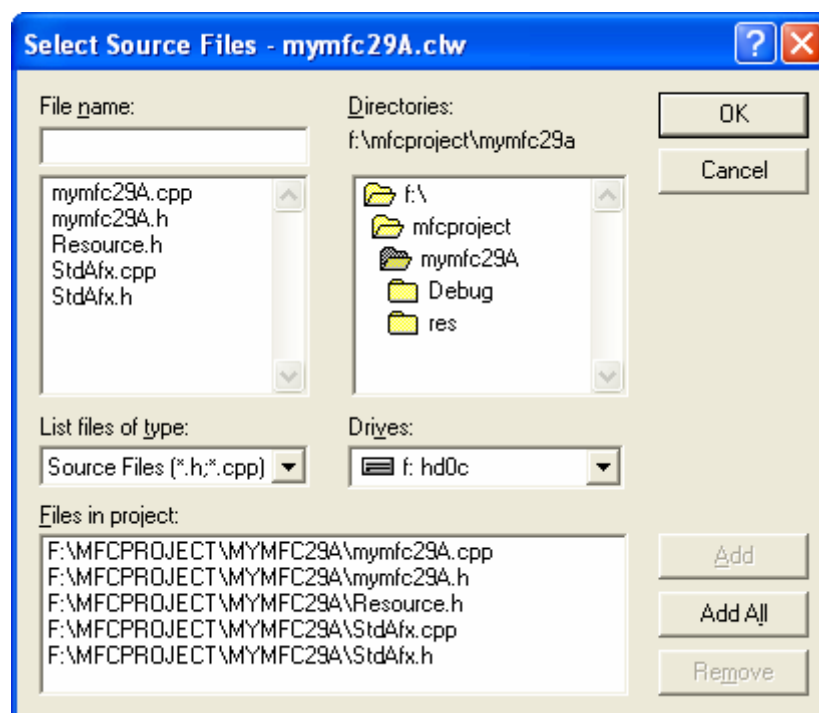


Figure 24: List of files for ClassWizard database rebuilding.

We need to generate the **ODL** file manually in order to create and update the type library (**TLB** file). ODL cannot be rebuilt as ClassWizard. Create an empty ODL file under the project directory using the project name as the file name.






	ReadMe.txt	4 KB	Text Document	4/22/2006 5:14 PM
	Resource.h	1 KB	C/C++ Header	4/22/2006 5:14 PM
	StdAfx.cpp	1 KB	C++ Source	4/22/2006 5:14 PM
	StdAfx.h	1 KB	C/C++ Header	4/22/2006 5:19 PM
	mymfc29A.odl	0 KB	Text Document	4/22/2006 5:38 PM

Figure 25: Creating ODL file.

Then add the ODL file to the project.

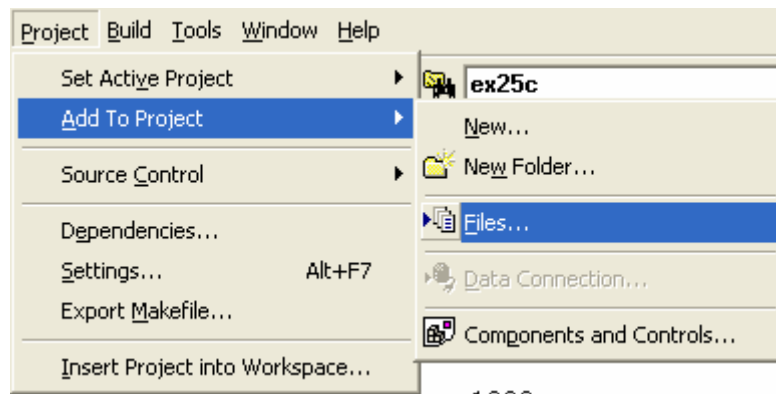


Figure 26: Adding new file to project.

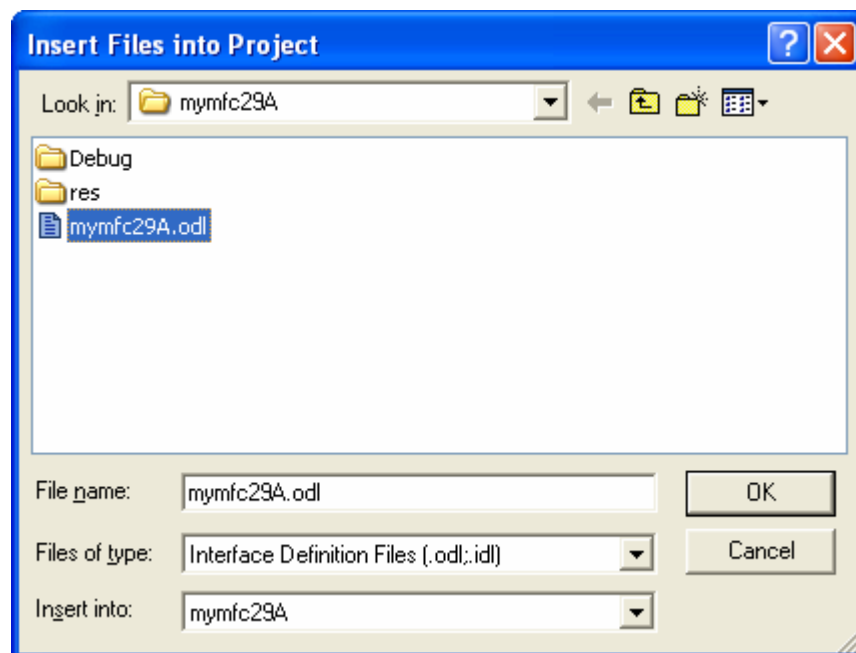


Figure 27: Adding ODL file to project.

Finally enter the following codes into the **mymfc29A.odl** file. The content will be updated automatically when we add components later on.

```
// Originally just a manually created odl template
[ uuid(40980C17-DB2B-498F-85AC-BE88F2B549CF), version(1.0) ]
library mymfc29A
{
    importlib("stdole32.tlb");
    importlib("stdole2.tlb");
}
```

```

    };
    //{{AFX_APPEND_ODL}}

```

The uuid 40980C17-DB2B-498F-85AC-BE88F2B549CF is copied from the **StdAfx.h**'s #if !define directive as shown below.

```

// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
// are changed infrequently
//

#ifdef AFX_STDAFX_H__40980C17_DB2B_498F_85AC_BE88F2B549CF__INCLUDED_
#define AFX_STDAFX_H__40980C17_DB2B_498F_85AC_BE88F2B549CF__INCLUDED_
#endif _MSC_VER > 1000

```

Figure 28: The uuid.

Use ClassWizard to add a new class, CBank, as shown here.

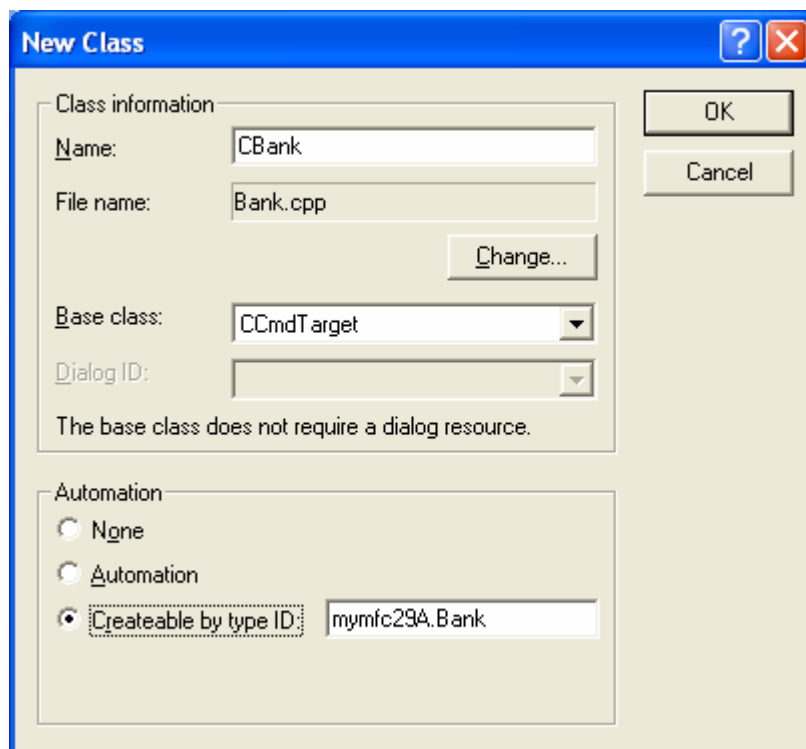


Figure 29: Adding new class to project.

Be sure to select the **Createable By Type ID** option. Now you can see the funny icon of the IBank component in the ClassView window and the ODL file has been updated.

Use ClassWizard to add two methods and a property. Click on the **Automation** tab, and then add a Withdrawal method, as shown here.

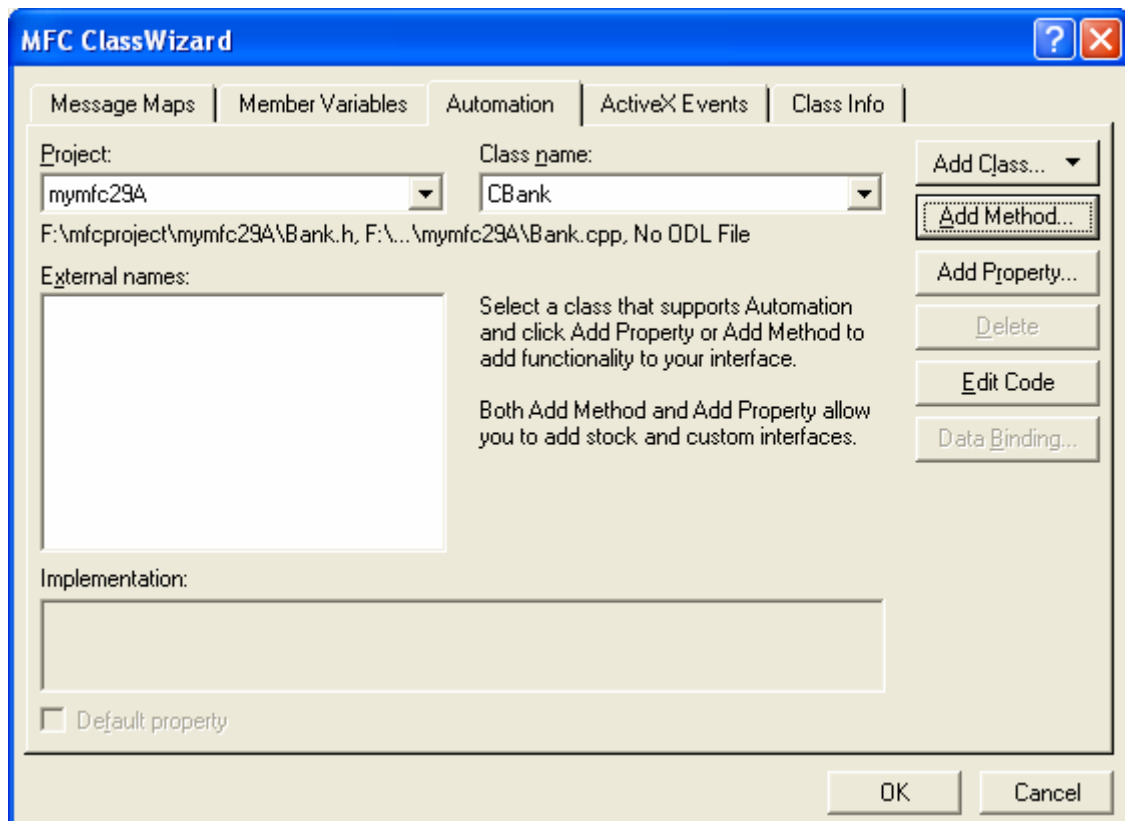


Figure 30: Adding method to project.

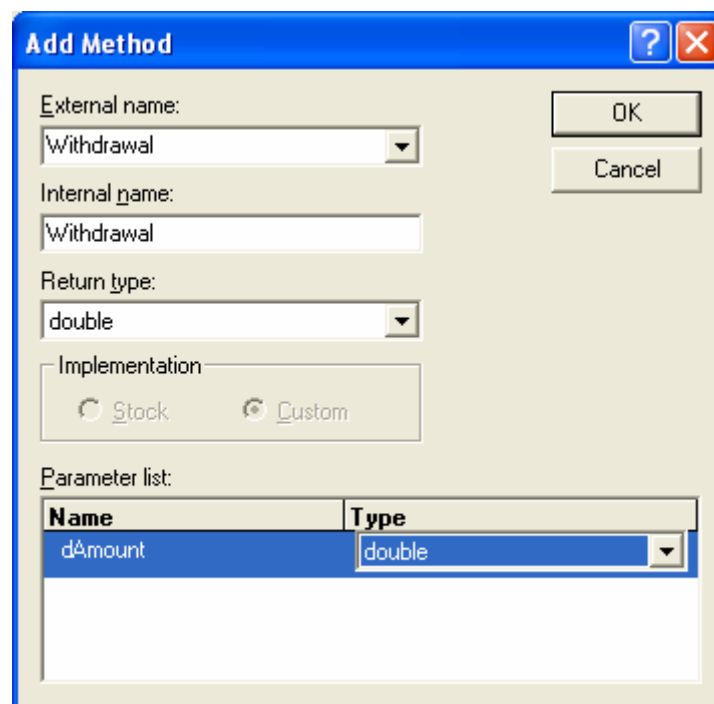


Figure 31: Entering Withdrawal method information.

The dAmount parameter is the amount to be withdrawn, and the return value is the actual amount withdrawn. If you try to withdraw \$100 from an account that contains \$60, the amount withdrawn is \$60. Add a similar Deposit method that returns void.

Add Method

External name: Deposit

Internal name: Deposit

Return type: void

Implementation: ☐ Stock ☒ Custom

Parameter list:

Name	Type
dAmount	double

OK Cancel

Figure 32: Entering Deposit method information.

Then add the Balance property, as shown here. Make sure you select the **Get/Set methods**.

Add Property

External name: Balance

Type: double

Get function: GetBalance

Set function: SetBalance

Implementation: ☐ Stock ☐ Member variable ☒ Get/Set methods

Parameter list:

Name	Type
------	------

OK Cancel

Figure 33: Adding and entering Balance property information.

We could have chosen direct access to a component data member, but then we wouldn't have read-only access. We choose **Get/Set methods** so that we can code the `SetBalance()` function to do nothing. Meanwhile you can check ClassView window for the component, it should display methods and property that we have added. The ODL file content also been updated.

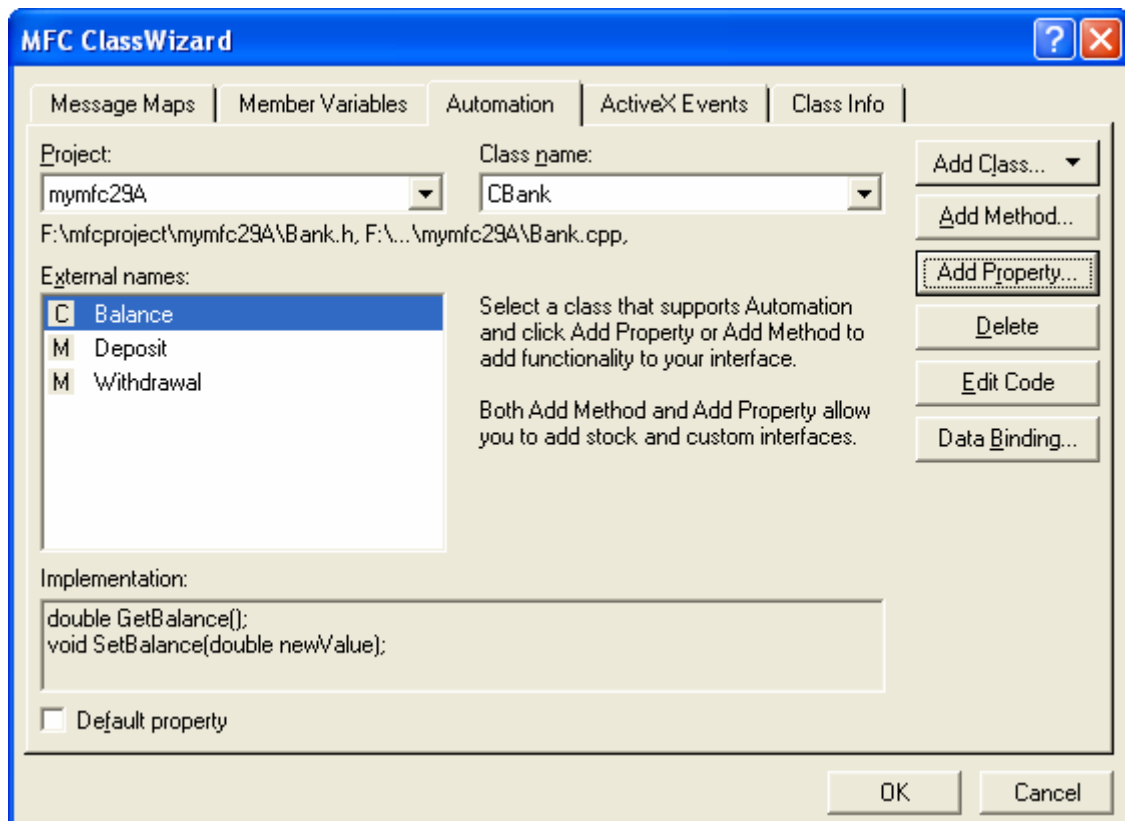


Figure 34: The added methods and property.

Using a ClassView, add a public `m_dBalance` data member of type `double` to the `CBank` class.

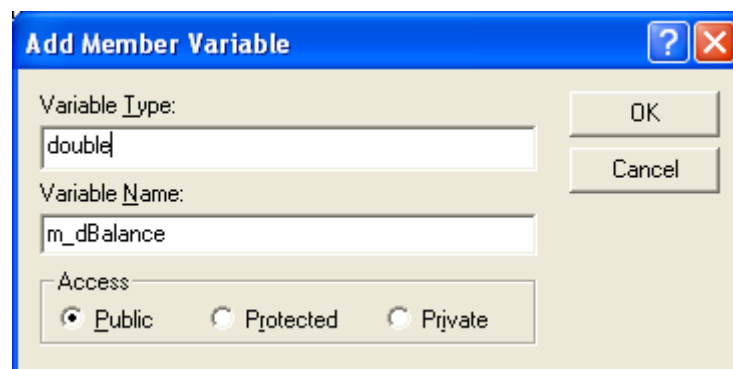


Figure 35: Adding member variable.

```
// Operations
public:
    double m_dBalance;

// Overrides
// ClassWizard generated
```

Listing 5.

Because we've chosen the Get/Set methods option for the `Balance` property, ClassWizard doesn't generate a data member for us. You already declared `m_dBalance` in the **Bank.h** file then you should initialize `m_dBalance` to 0.0 in the `CBank` constructor located in the **Bank.cpp** file manually.

```

CBank::CBank()
{
    m_dBalance = 0.0;

    EnableAutomation();

    // To keep the application running as 1
    // object is active, the constructor c

    AfxOleLockApp();
}

```

Listing 6.

Edit the generated method and property functions. Add the following code (**Bank.cpp**):

```

double CBank::Withdrawal(double dAmount)
{
    if (dAmount < 0.0){
        return 0.0;
    }
    if (dAmount <= m_dBalance){
        m_dBalance -= dAmount;
        return dAmount;
    }
    double dTemp = m_dBalance;
    m_dBalance = 0.0;
    return dTemp;
}

void CBank::Deposit(double dAmount)
{
    if (dAmount < 0.0){
        return;
    }
    m_dBalance += dAmount;
}

double CBank::GetBalance()
{ return m_dBalance; }

void CBank::SetBalance(double newValue)
{ TRACE("Sorry, Doe, I can't do that!\n"); }

```



```

// CBank message handlers
double CBank::Withdrawal(double dAmount)
{
    // TODO: Add your dispatch handler code here
    if (dAmount < 0.0) {
        return 0.0;
    }
    if (dAmount <= m_dBalance) {
        m_dBalance -= dAmount;
        return dAmount;
    }
    double dTemp = m_dBalance;
    m_dBalance = 0.0;
    return dTemp;
}

void CBank::Deposit(double dAmount)
{
    // TODO: Add your dispatch handler code here
    if (dAmount < 0.0) {
        return;
    }
    m_dBalance += dAmount;
}

double CBank::GetBalance()
{
    // TODO: Add your property handler here
    return m_dBalance;
}

void CBank::SetBalance(double newValue)
{
    // TODO: Add your property handler here
    TRACE("Sorry, John Doe, I can't do that!\n");
}

```

Listing 7.

Build the MYMFC29A program; run it once to register the component. The following is the output, just a prompt message box telling us that the component successfully registered.

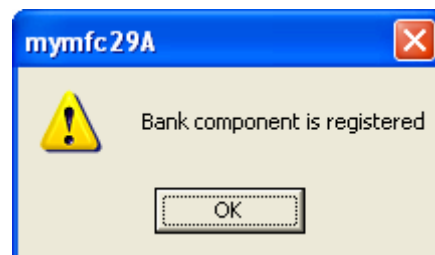


Figure 36: MYMFC29A output, prompting the component successfully registered.

Set up five Excel macros in a new workbook file, [mymfc29A.xls](#). Launch the Excel and save the workbook as **mymfc29A.xls**. Launch the Visual Basic Editor (Alt + F11). Select the **Sheet1** and add the following macro code:

```

Dim Bank As Object
Sub LoadBank()
    Set Bank = CreateObject("Mymfc29A.Bank")
End Sub

Sub UnloadBank()
    Set Bank = Nothing
End Sub

```

```

Sub DoDeposit()
    Range("D4").Select
    Bank.Deposit (ActiveCell.Value)
End Sub

Sub DoWithdrawal()
    Range("E4").Select
    Dim Amt
    Amt = Bank.Withdrawal(ActiveCell.Value)
    Range("E5").Select
    ActiveCell.Value = Amt
End Sub

Sub DoInquiry()
    Dim Amt
    Amt = Bank.Balance()
    Range("G4").Select
    ActiveCell.Value = Amt
End Sub

```

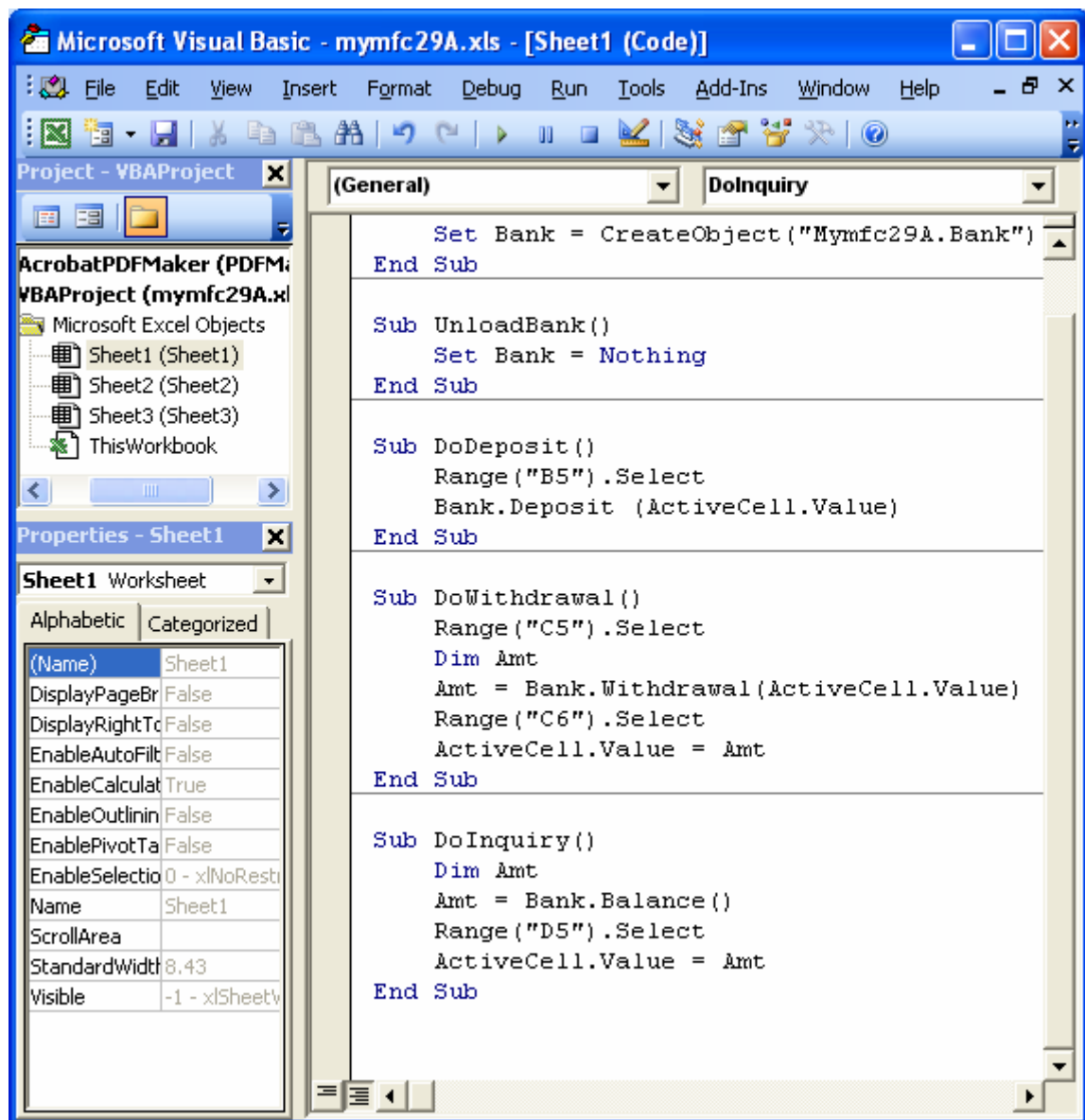


Figure 37: Excel (client) macro code for MYMFC29A (component) testing purpose.

Arrange an Excel worksheet as shown in Figure 39. Attach the macros to the pushbuttons appropriately (by right-clicking the pushbuttons). You can also see all the macros through the **Tools Macro** menu.

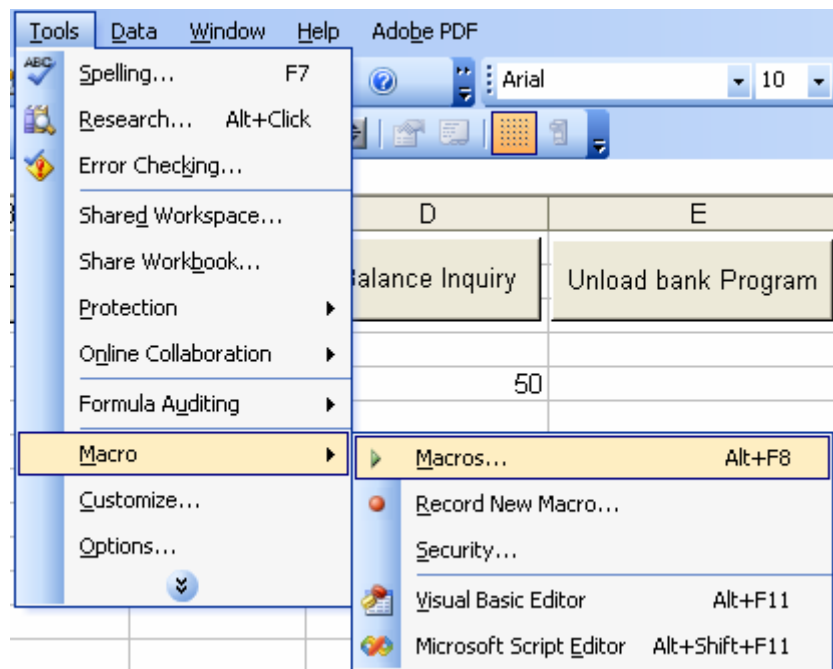


Figure 38: Viewing macros through Macros menu.

Test the MYMFC29A bank component. Click the **Load Bank Program** button, and then enter a deposit value in cell B5 and click the **Deposit** button. Click the **Balance Inquiry** button, and watch the balance appear in cell D5.

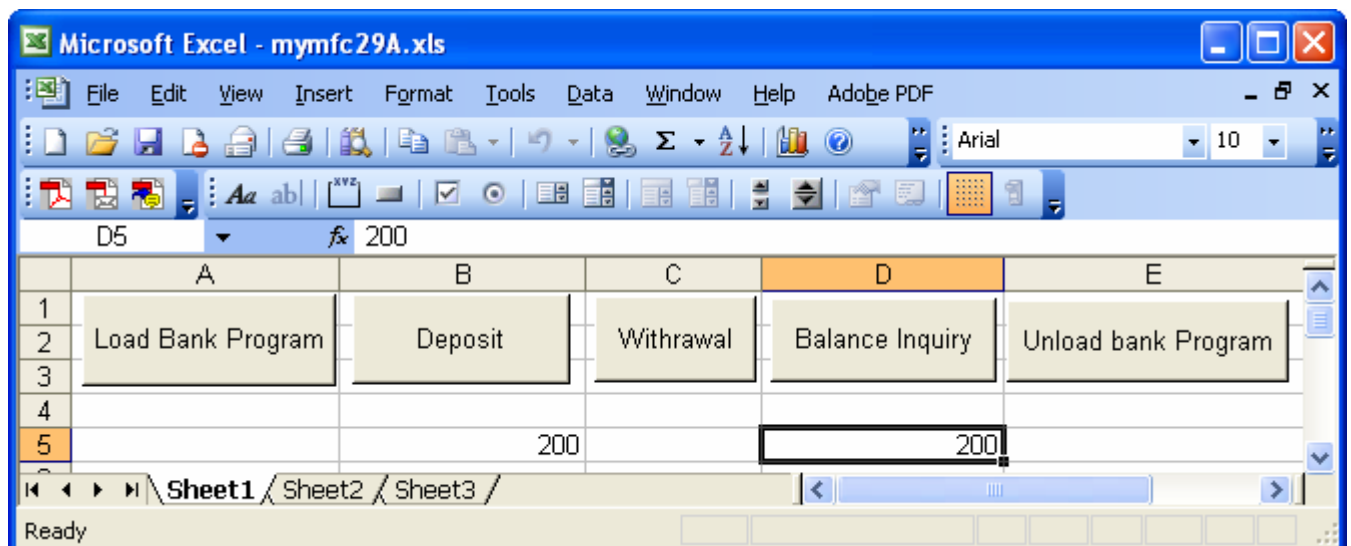


Figure 39: Testing MYMFC29A bank component.

Enter a withdrawal value in cell C5, and click the **Withdrawal** button. To see the balance, click the **Balance Inquiry** button.

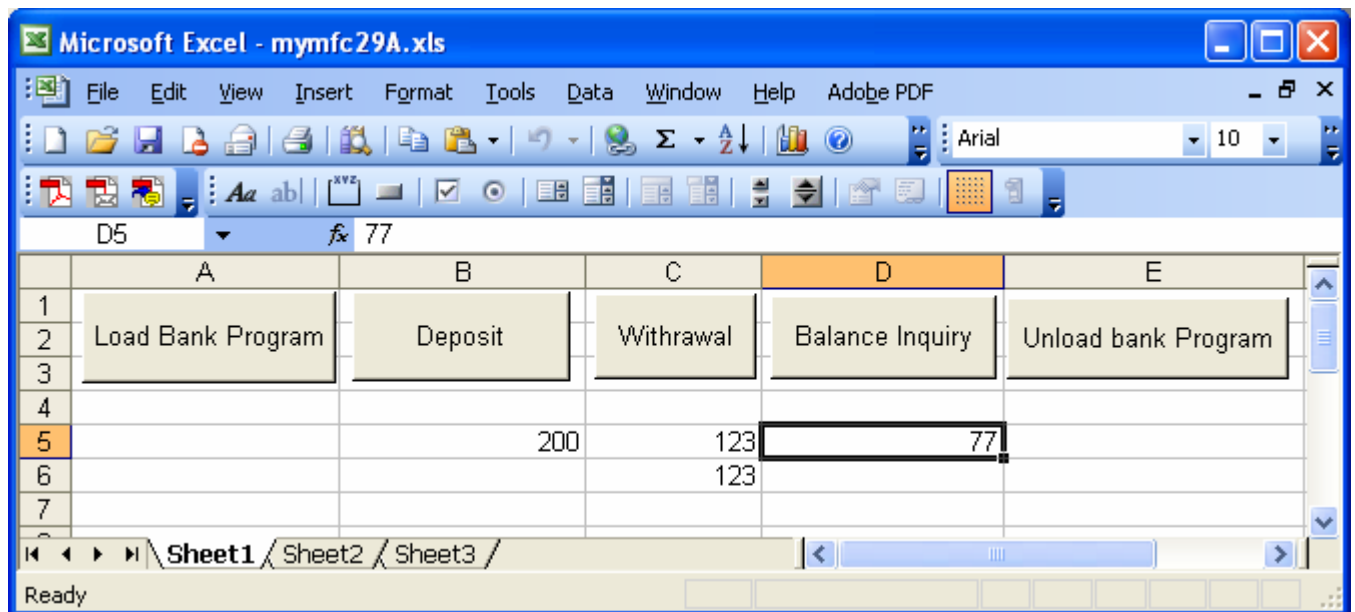


Figure 40: Testing MYMFC29A bank component.

Sometimes you need to click the buttons twice. The first click switches the focus to the worksheet, and the second click runs the macro. If any, the hourglass pointer tells you the macro is working. What's happening in this program? Look closely at the `CMymfc29AApp::InitInstance` function. When you run the program directly from Windows, it displays a message box and then quits, but not before it updates the Registry. The `COleObjectFactory::UpdateRegistryAll` function hunts for global class factory objects, and the `CBank` class's `IMPLEMENT_OLECREATE` macro invocation defines such an object. The `IMPLEMENT_OLECREATE` line was generated because you checked ClassWizard's **Createable By Type ID** check box when you added the `CBank` class. The unique class ID and the program ID, `MYMFC29A.BANK`, are added to the Registry.

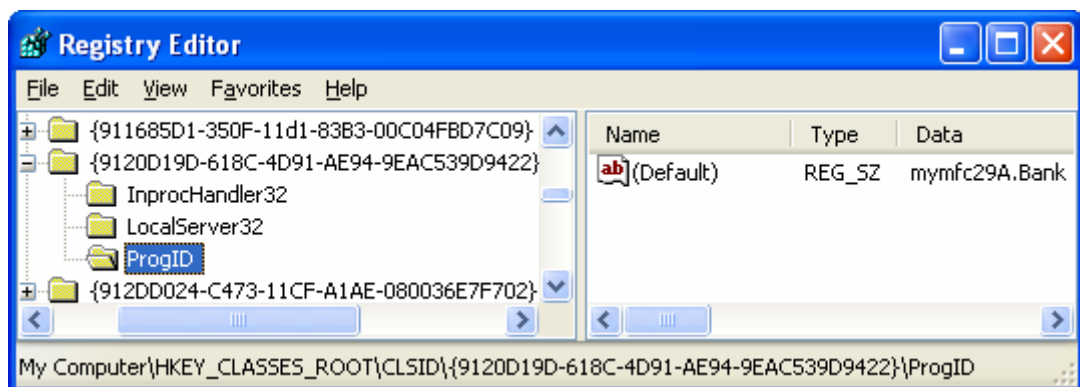


Figure 41: Viewing the registered component in Registry.

MYMFC29A Story

When Excel now calls `CreateObject()`, COM loads the MYMFC29A program, which contains the global factory for `CBank` objects; COM then calls the factory object's `CreateInstance()` function to construct the `CBank` object and return an `IDispatch` pointer. Here's the `CBank` class declaration that ClassWizard generated in the `Bank.h` file, with unnecessary detail (and the method and property functions you've already seen) omitted:

```
class CBank : public CCmdTarget
{
    DECLARE_DYNCREATE(CBank)
public:
```

```

        double m_dBalance;
        // protected constructor used by dynamic creation
        CBank();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CBank)
    public:
    virtual void OnFinalRelease();
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CBank();

    // Generated message map functions
    //{{AFX_MSG(CBank)
        // NOTE - the ClassWizard will add and remove member
        // functions here.
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
    DECLARE_OLECREATE(CBank)

    // Generated OLE dispatch map functions
    //{{AFX_DISPATCH(CBank)
    afx_msg double GetBalance();
    afx_msg void SetBalance(double newValue);
    afx_msg double Withdrawal(double dAmount);
    afx_msg void Deposit(double dAmount);
    //}}AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    DECLARE_INTERFACE_MAP()
    };

```

Here is the code automatically generated by ClassWizard in **Bank.cpp**:

```

IMPLEMENT_DYNCREATE(CBank, CCmdTarget)

CBank::CBank()
{
    EnableAutomation();
    // To keep the application running as long as an OLE automation
    // object is active, the constructor calls AfxOleLockApp.
    AfxOleLockApp();
}

CBank::~~CBank()
{
    // To terminate the application when all objects created with
    // OLE automation, the destructor calls AfxOleUnlockApp.

    AfxOleUnlockApp();
}

void CBank::OnFinalRelease()
{

```

```

        // When the last reference for an automation object is released,
        // OnFinalRelease is called. This implementation deletes the
        // object. Add additional cleanup required for your object
        // before deleting it from memory.
        CCmdTarget::OnFinalRelease
    }

BEGIN_MESSAGE_MAP(CBank, CCmdTarget)
    //{{AFX_MSG_MAP(CBank)
        // NOTE - the ClassWizard will add and remove
        // mapping macros here.
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CBank, CCmdTarget)
    //{{AFX_DISPATCH_MAP(CBank)
        DISP_PROPERTY_EX(CBank, "Balance", GetBalance, SetBalance, VT_R8)
        DISP_FUNCTION(CBank, "Withdrawal", Withdrawal, VT_R8, VTS_R8)
        DISP_FUNCTION(CBank, "Deposit", Deposit, VT_EMPTY, VTS_R8)
    //}}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: we add support for IID_IBank to support type safe binding
// from VBA. This IID must match the GUID that is attached to the
// dispinterface in the .ODL file.

// {A9515AB6-5B85-11D0-848F-00400526305B}
static const IID IID_IBank = { 0xa9515ab6, 0x5b85, 0x11d0, { 0x84, 0x8f, 0x0,
0x40, 0x5, 0x26, 0x30, 0x5b } };

BEGIN_INTERFACE_MAP(CBank, CCmdTarget)
    INTERFACE_PART(CBank, IID_IBank, Dispatch)
END_INTERFACE_MAP()

// {632B1E4C-F287-11CE-B5E3-00AA005B1574}
IMPLEMENT_OLECREATE2(CBank, "MYMFC29A.BANK", 0x632b1e4c, 0xf287,
0x11ce, 0xb5, 0xe3, 0x0, 0xaa, 0x0, 0x5b, 0x15, 0x74)

```

This first version of the MYMFC29A program runs in **single-process mode**, as does the **Autoclik** program. If a second Automation client asks for a new CBank object, COM calls the class factory CreateInstance() function again and the existing process constructs another CBank object on the heap. You can verify this by making a copy of the **mymfc29A.xls** workbook (under a different name) and loading both the original and the copy. Click the **Load Bank Program** button in each workbook, and watch the Debug window. InitInstance() should be called only once.

A small change in the MYMFC29A program makes it behave differently. To have a new MYMFC29A process start up each time a new component object is requested, follow these steps.

Add the following macro in **Bank.h**:

```

#define IMPLEMENT_OLECREATE2(class_name, external_name, \
    l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \
    AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
    \
    RUNTIME_CLASS(class_name), TRUE, _T(external_name)); \
    const AFX_DATADEF GUID class_name::guid = \
    { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } };

```

```

#define IMPLEMENT_OLECREATE2(class_name, external_name, \
    l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8) \
    AFX_DATADEF COleObjectFactory class_name::factory(class_name::guid, \
    RUNTIME_CLASS(class_name), TRUE, _T(external_name)); \
    const AFX_DATADEF GUID class_name::guid = \
    { l, w1, w2, { b1, b2, b3, b4, b5, b6, b7, b8 } };
|
// CBank command target
class CBank : public CCmdTarget

```

Listing 8.

This macro is the same as the standard MFC IMPLEMENT_OLECREATE macro except that the original FALSE parameter (after the RUNTIME_CLASS parameter) has been changed to TRUE.

In **Bank.cpp**, change the IMPLEMENT_OLECREATE macro invocation to IMPLEMENT_OLECREATE2.

```

// {58E7A2CE-C23D-4A09-A1AD-E4710AE28E4D}
IMPLEMENT_OLECREATE2(CBank, "mymfc29A.Bank", 0x58e7a2ce, 0xc23d,
    0x4a09, 0xa1, 0xad, 0xe4, 0x71, 0xa, 0xe2, 0x8e, 0x4d)

// CBank message handlers

```

Listing 9.

Build the program and test it using Excel. Start two Excel processes and then load the bank program from each. Use the Microsoft Windows Task Manager or SPY++ (Visual C++ Tool) to verify that two MYMFC29A processes are running.

Debugging an EXE Component Program

When an Automation client launches an EXE component program, it sets the **/Embedding** command-line parameter. If you want to debug your component, you must do the same. Choose **Settings** from the **Visual C++ Project** menu, and then enter **/Embedding** in the **Program Arguments** box on the **Debug** page, as shown here.

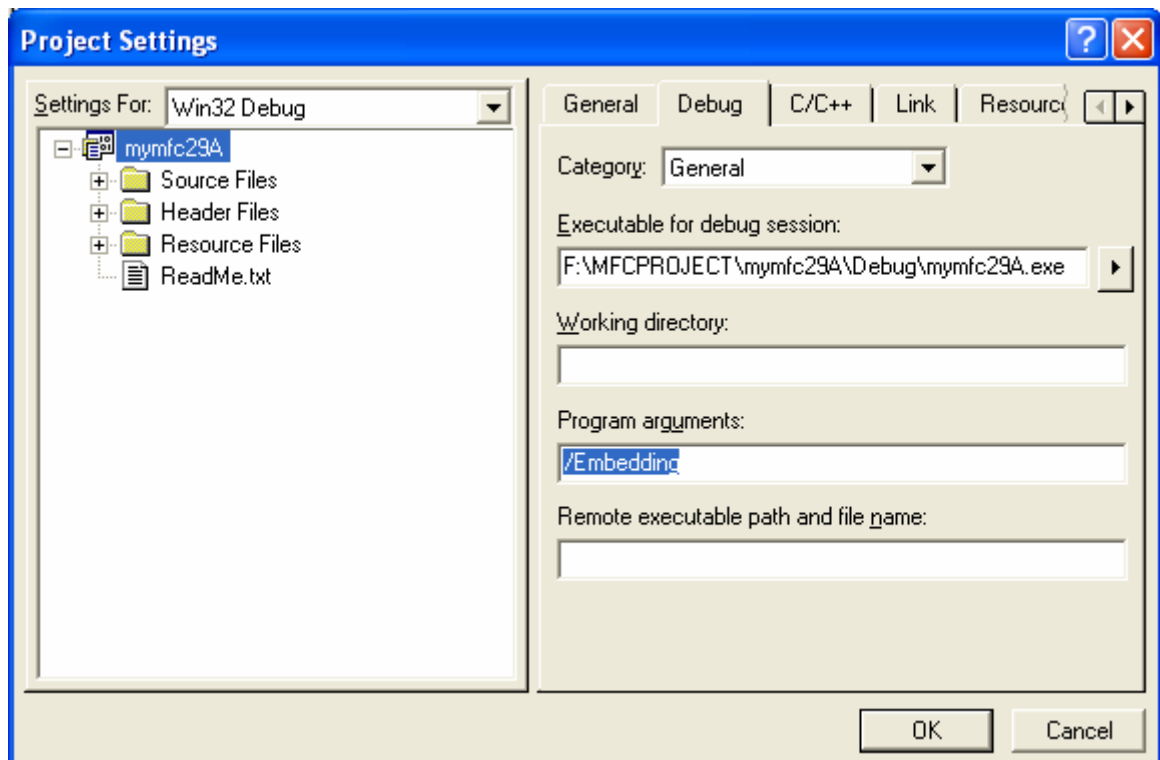


Figure 42: Changing the project Debug setting.

When you click the **Go** button on the **Debug** toolbar, your program will start and then wait for a client to activate it. At this point, you should start the client program from Windows (if it is not already running) and then use it to create a component object. Your component program in the debugger should then construct its object. It might be a good idea to include a **TRACE** statement in the component object's constructor. Don't forget that your component program must be registered before the client can find it. That means you have to run it once without the **/Embedding** flag. Many clients don't synchronize with Registry changes. If your client is running when you register the component, you may have to restart the client.

The MYMFC29B Automation Component DLL Example

You could easily convert MYMFC29A from an EXE to a DLL. The **CBank** class would be exactly the same, and the Excel driver would be similar. It's more interesting, though, to write a new application, this time with a minimal user interface (UI). We'll use a modal dialog box because it's the most complex UI we can conveniently use in an Automation DLL.

Parameters Passed by Reference

So far, you've seen VBA parameters **passed by value**. VBA has pretty strange rules for calling methods. If the method has one parameter, you can use parentheses; if it has more than one, you can't (unless you're using the function's return value, in which case you must use **parentheses**). Here is some sample VBA code that passes the string parameter by value:

```
Object.Method1 parm1, "text"
Object.Method2("text")
Dim s as String
s = "text"
Object.Method2(s)
```

Sometimes, though, VBA passes the address of a parameter (a reference). In this example, the string is passed by reference:

```
Dim s as String
```

```
s = "text"
Object.Method1 parm1, s
```

You can override VBA's default behavior by prefixing a parameter with `ByVal` or `ByRef`. Your component can't predict if it's getting a value or a reference, it must prepare for both. The trick is to test `vt` to see whether its `VT_BYREF` bit is set. Here's a sample method implementation that accepts a string (in a `VARIANT`) passed either by reference or value:

```
void CMyComponent::Method(long nParm1, const VARIANT& vaParm2)
{
    CString str;
    if ((vaParm2.vt & 0x7f) == VT_BSTR)
    {
        if ((vaParm2.vt & VT_BYREF) != 0)
            str = *(vaParm2.pbstrVal); // by reference
        else
            str = vaParm2.bstrVal; // by value
    }
    AfxMessageBox(str);
}
```

If you declare a `BSTR` (`BSTRs` are wide, double-byte (Unicode) strings on 32-bit Windows platforms, and narrow, single-byte strings on 16-bit Windows) parameter, the MFC library does the conversion for you. Suppose your client program passed a `BSTR` reference to an out-of-process component and the component program changed the value. Because the component can't access the memory of the client process, COM must copy the string to the component and then copy it back to the client after the function returns. So before you declare reference parameters, remember that passing references through `IDispatch` is not like passing references in C++.

The MYMFC29B steps From Scratch

The following are the steps for MYMFC29B DLL program starting from scratch.

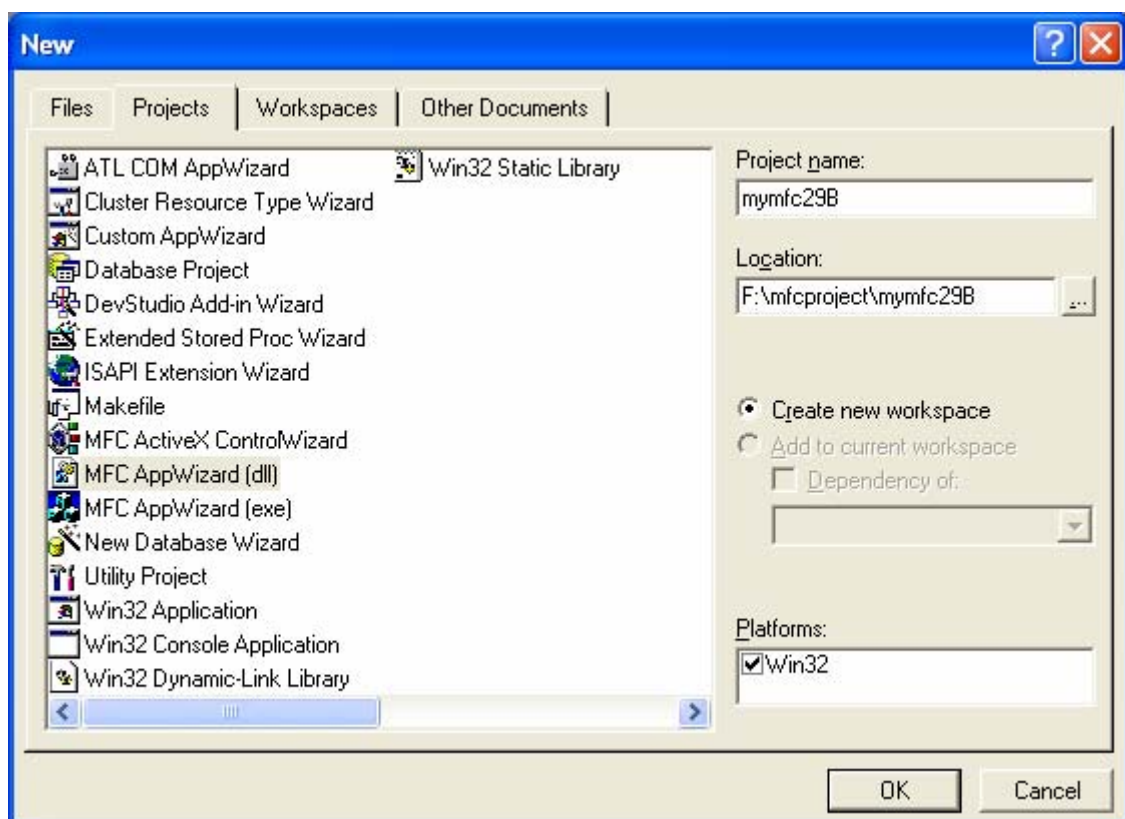


Figure 43: Visual C++ new MYMFC29B DLL project dialog.

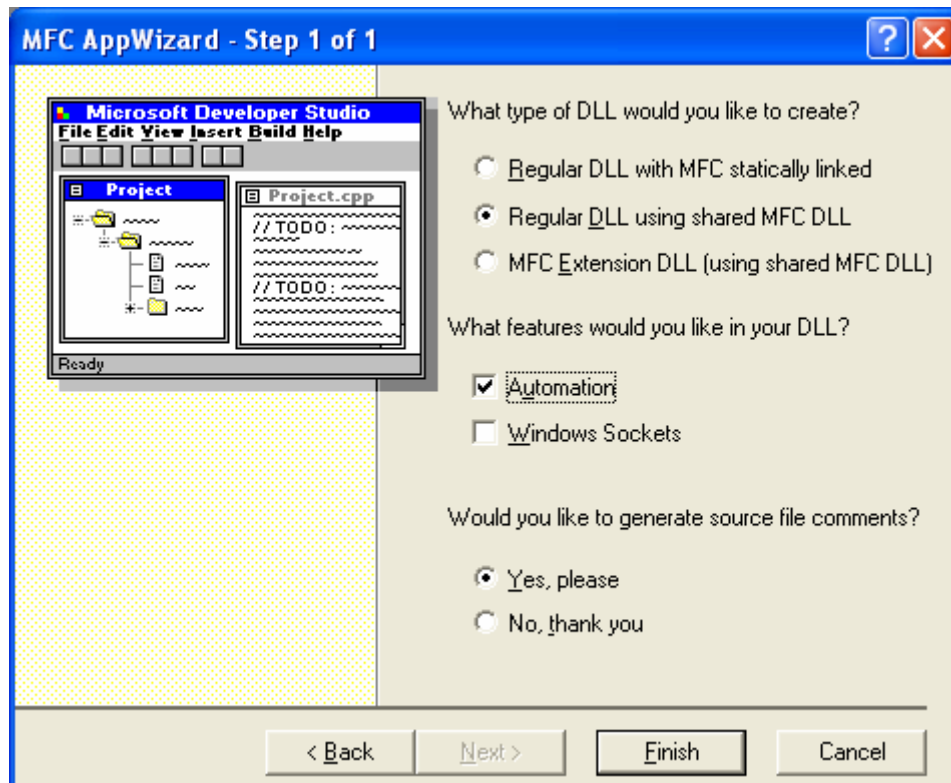


Figure 44: MYMFC29B Regular DLL project, AppWizard step 1 of 1.

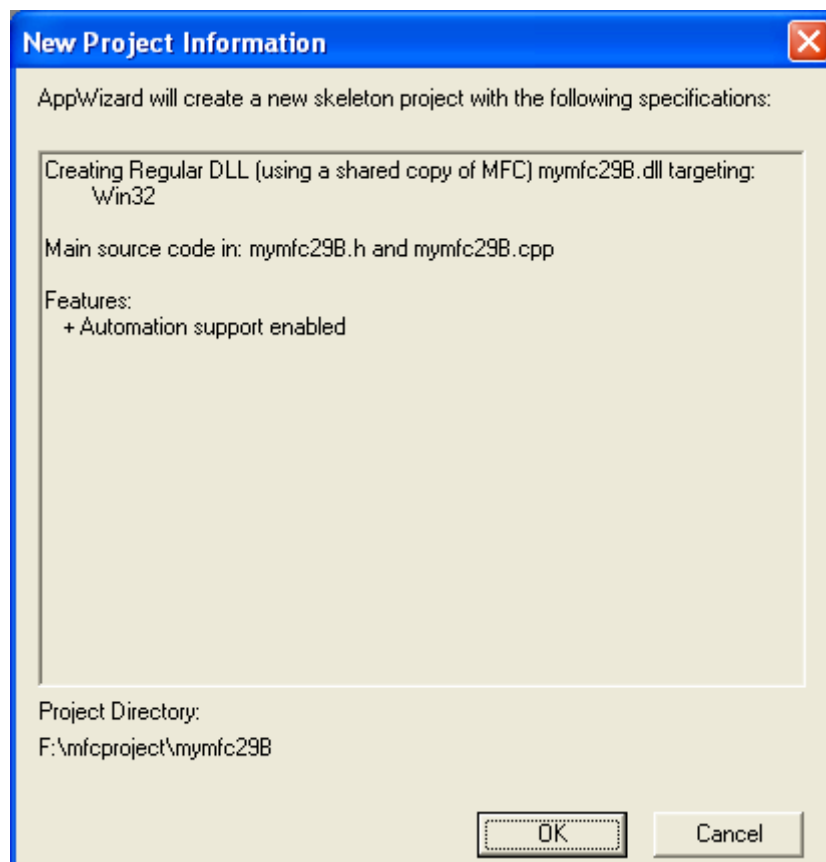


Figure 45: MYMFC29B project summary.

Add dialog and controls.

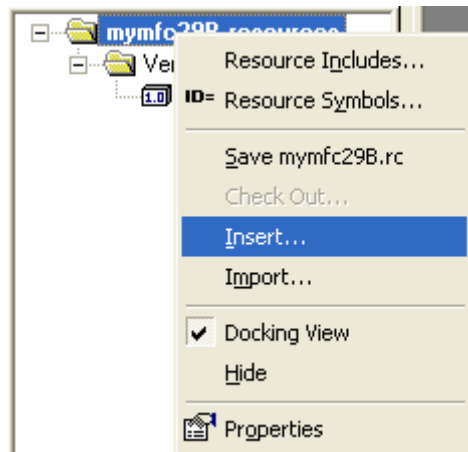


Figure 46: Inserting new resource to project.

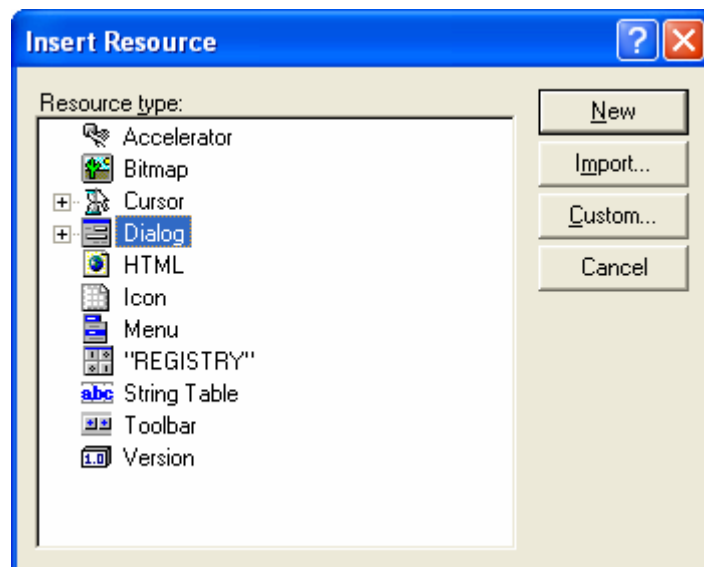


Figure 47: Inserting new dialog to project.

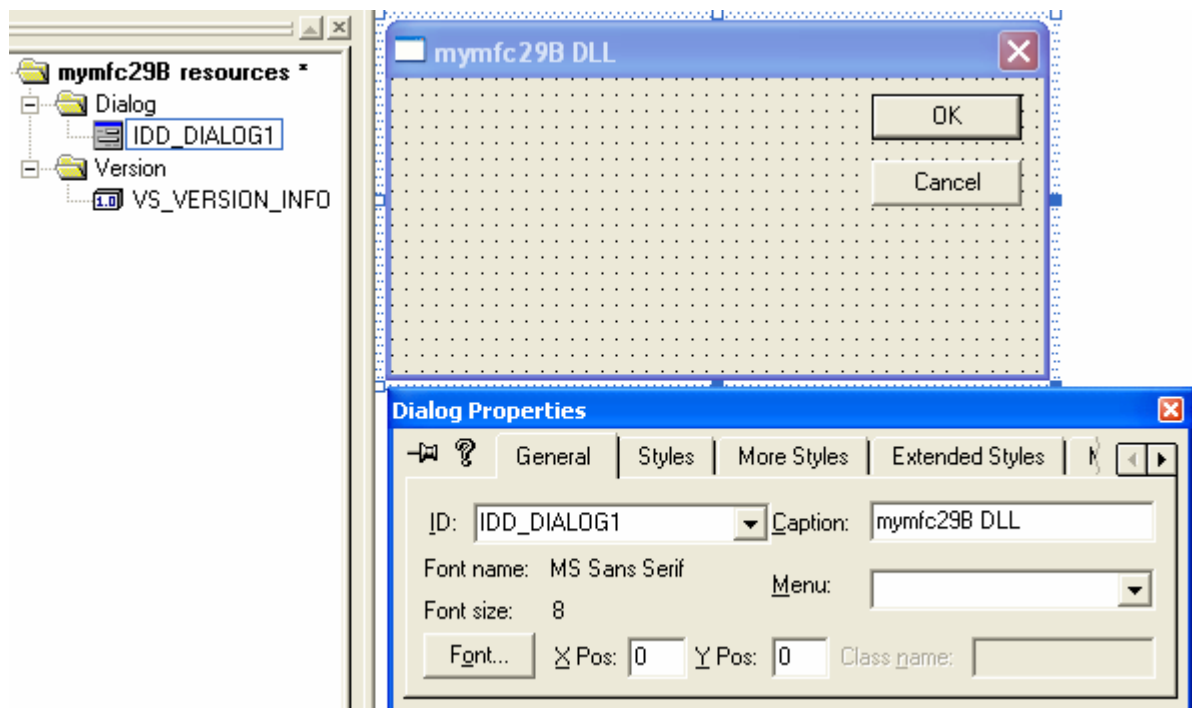


Figure 48: The dialog property.

Add Edit controls.

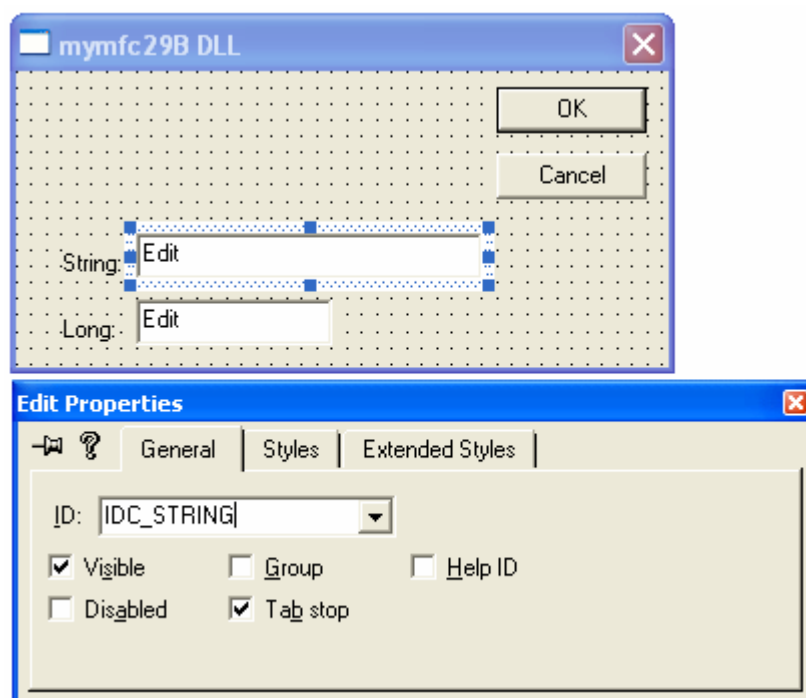


Figure 49: The Edit control property.

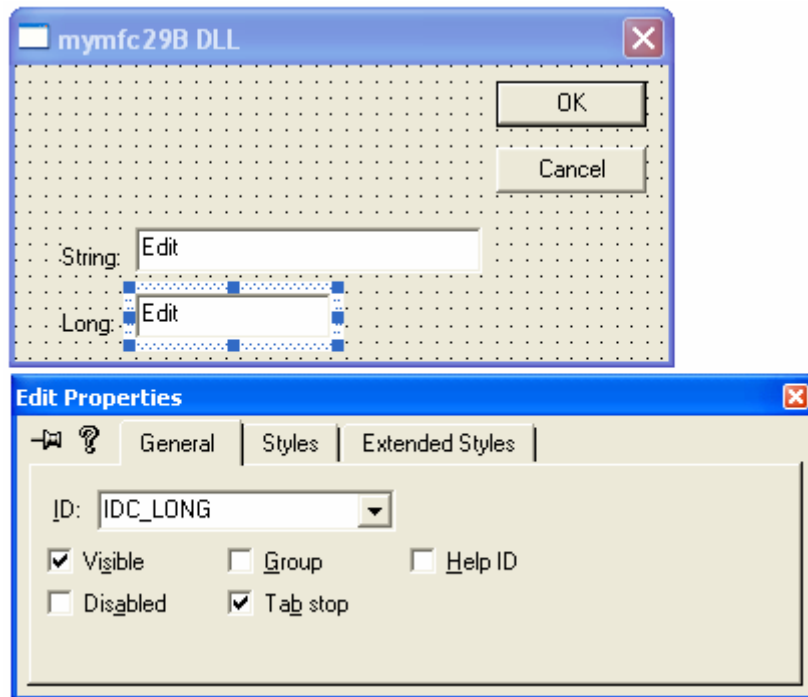


Figure 50: The Edit control property page.

Launch ClassWizard and just click **OK** to create a new class, `CPromptDlg` for the dialog.

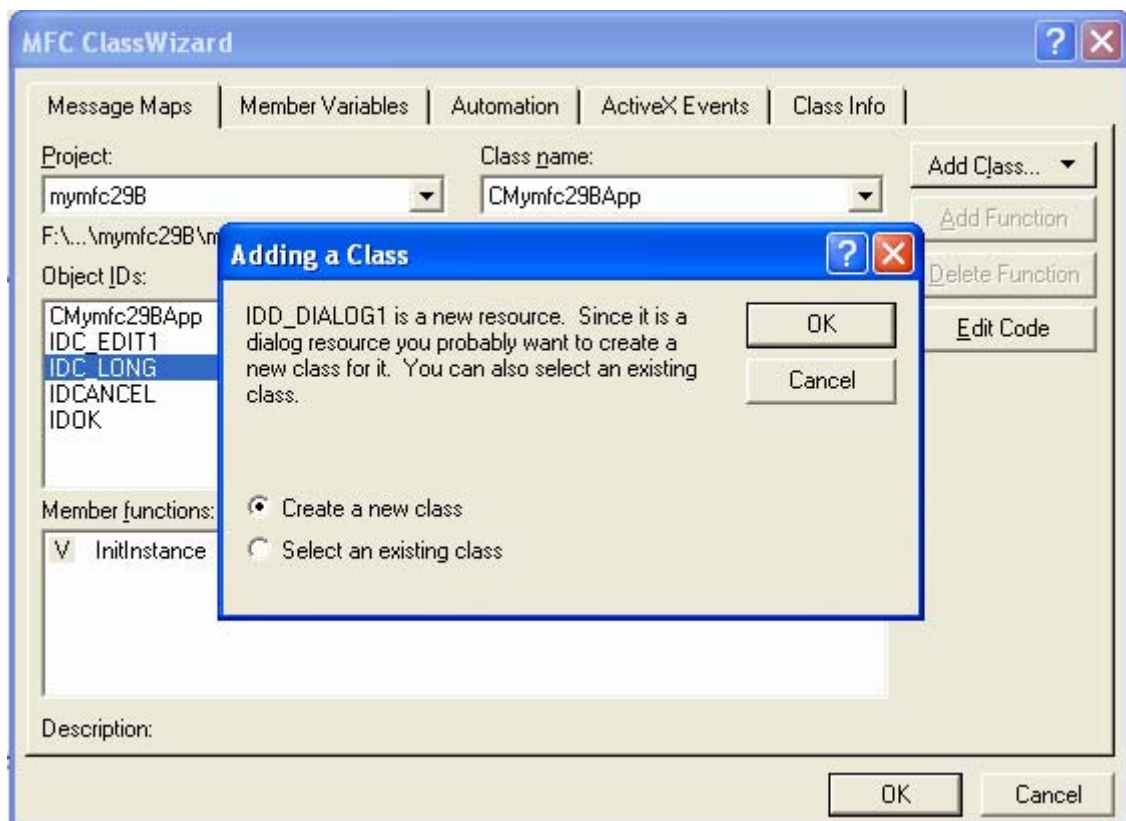


Figure 51: New class addition prompt dialog.

Type in the `CPromptDlg` as a class name.

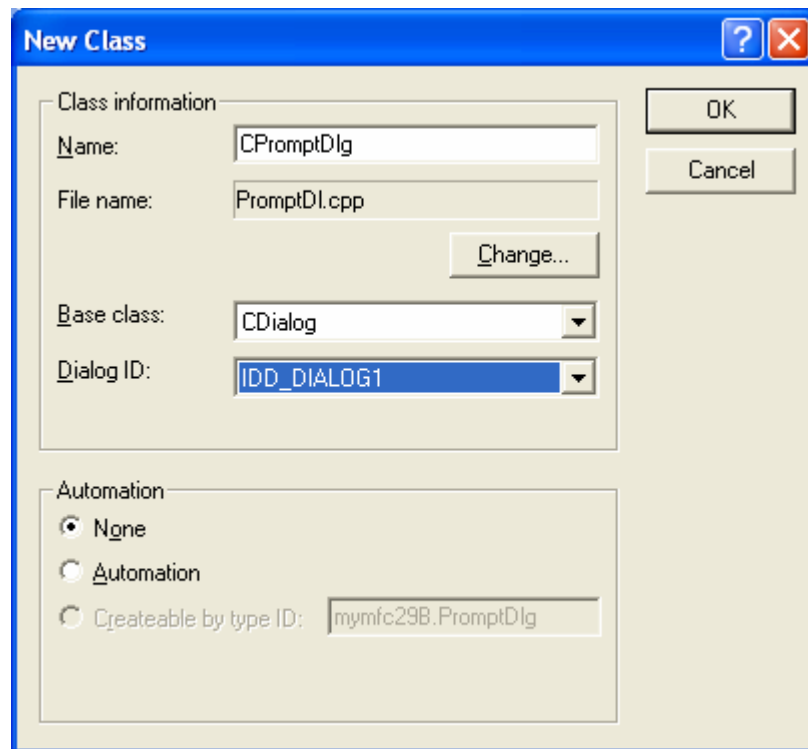


Figure 53: Adding CPromptDlg class, a new class.

Click the **Change** button and change the header and source file for the CPromptDlg class. You can leave it as default.

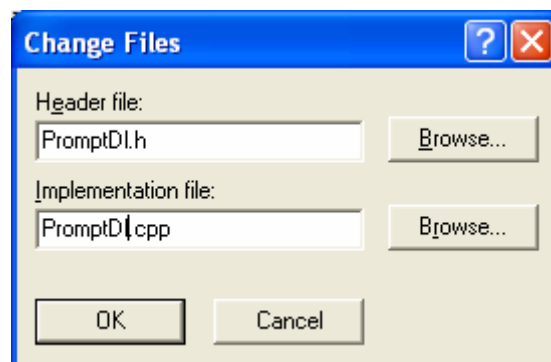


Figure 52: Changing the class header and source files name.

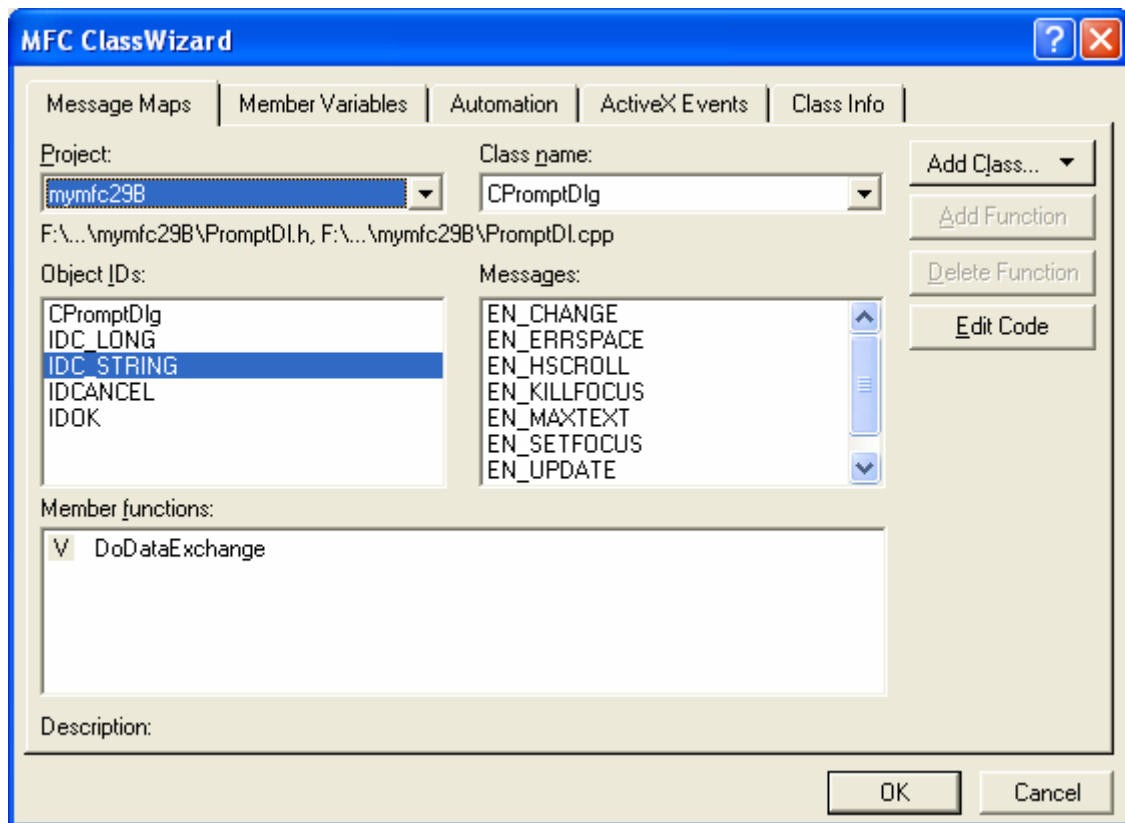


Figure 54: The added CPromptDlg class to ClassWizard database.

Add the following member variables.

```
long m_lData;
CString m_strData;
```

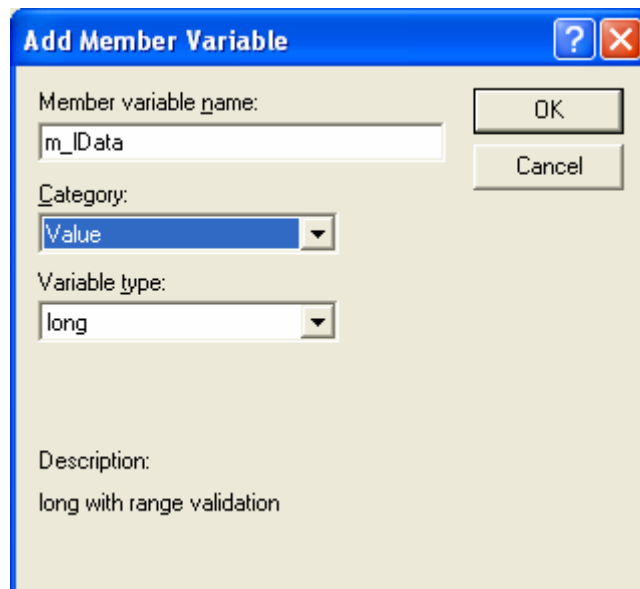


Figure 55: Adding m_lData member variable.

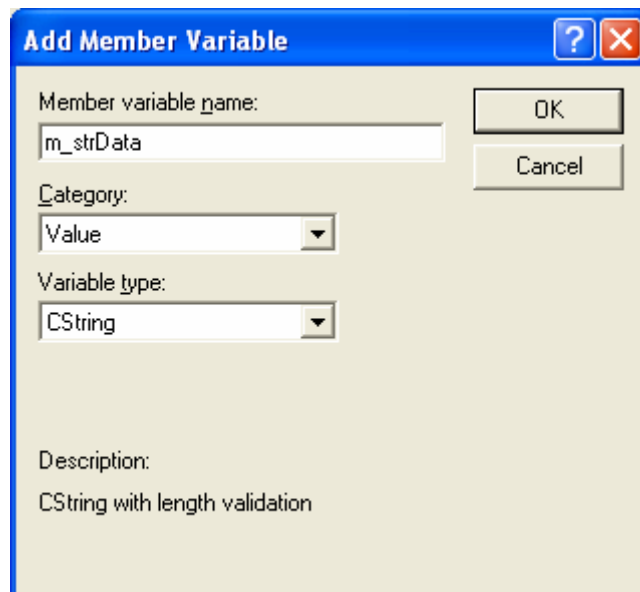


Figure 56: Adding m_strData member variable.

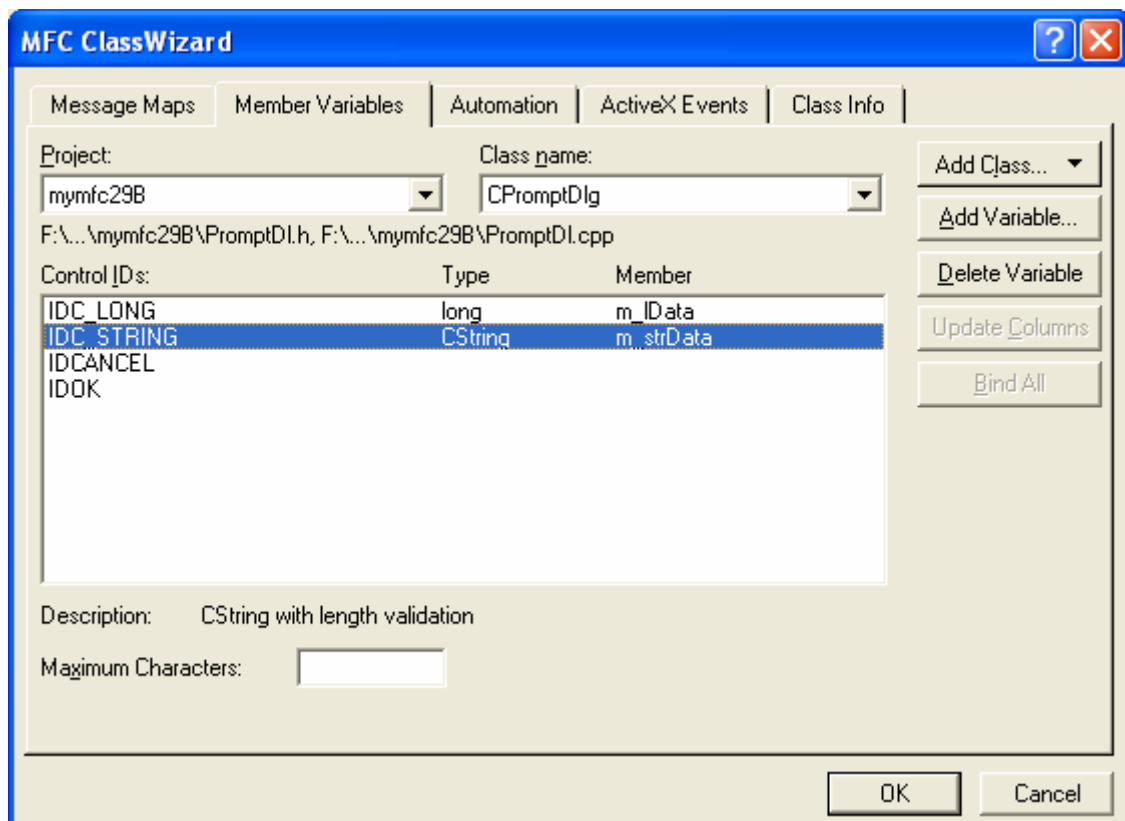
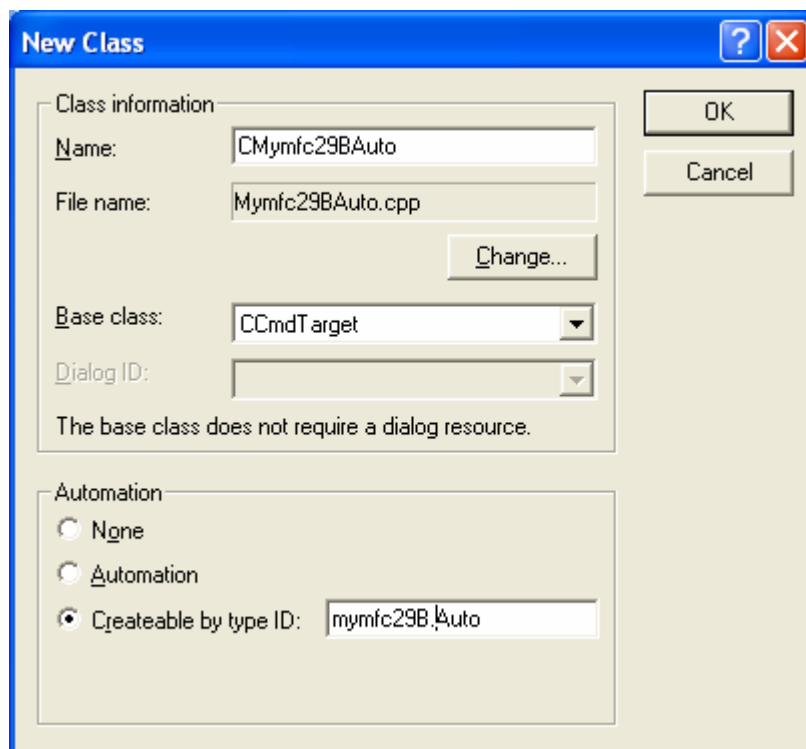


Figure 57: The added member variables using ClassWizard.

Next, add new class, CMymfc29BAuto. Don't forget to select the **Creatable by type ID** option for Automation support.



The 'New Class' dialog box is shown with the following fields and options:

- Class information:**
 - Name: CMyMfc29BAuto
 - File name: Mymfc29BAuto.cpp
 - Base class: CCmdTarget
 - Dialog ID: (empty)
 - Change... button
- Automation:**
 - ☐ None
 - ☐ Automation
 - ☒ Createable by type ID: mymfc29BAuto

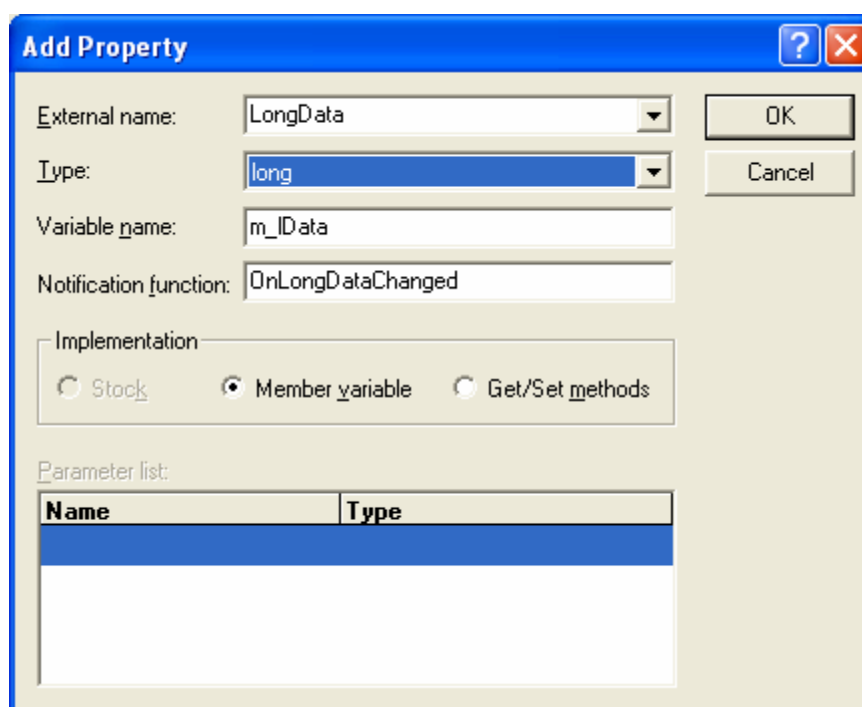
Buttons: OK, Cancel

Figure 58: Adding new class with Automation support.

Add the following method and property through the Automation page of the ClassWizard.

Method/Property	Description
LongData	Long integer property.
TextData	VARIANT property.
DisplayDialog	Method - no parameters, BOOL return.

Table 3.



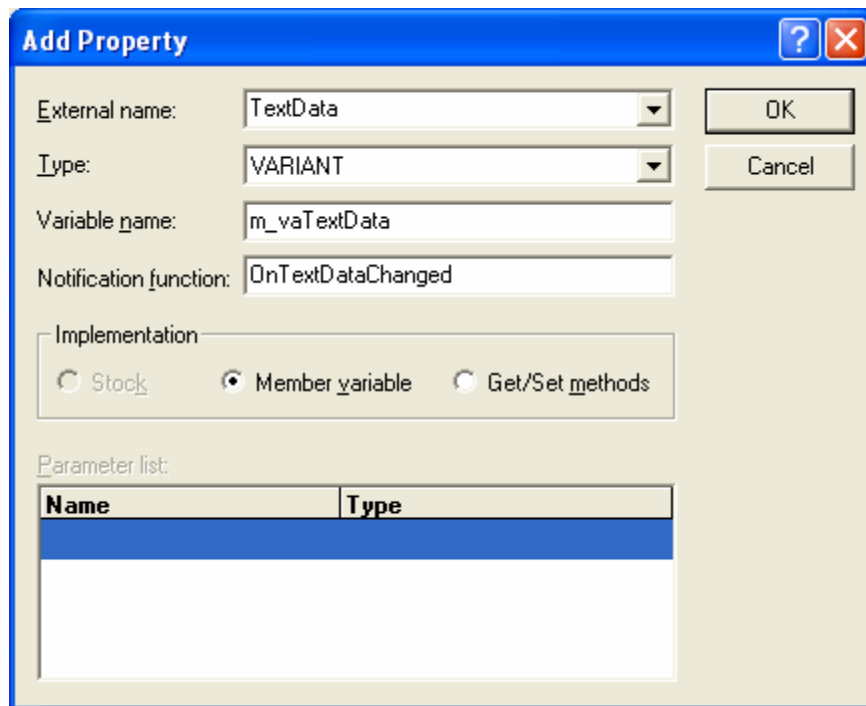
The 'Add Property' dialog box is shown with the following fields and options:

- External name:** LongData
- Type:** long
- Variable name:** m_IData
- Notification function:** OnLongDataChanged
- Implementation:**
 - ☐ Stock
 - ☒ Member variable
 - ☐ Get/Set methods
- Parameter list:**

Name	Type

Buttons: OK, Cancel

Figure 59: Adding LongData property information.

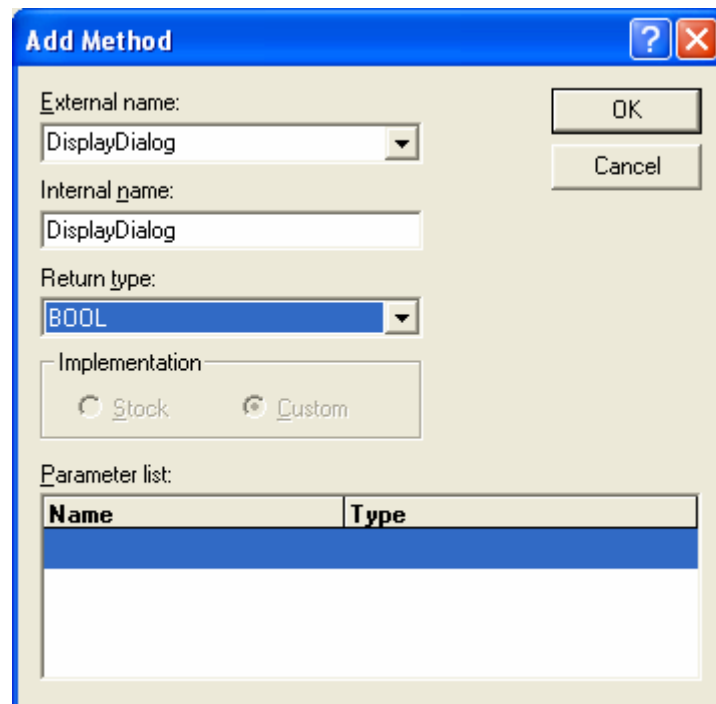


The 'Add Property' dialog box is shown with the following fields and options:

- External name:** TextData
- Type:** VARIANT
- Variable name:** m_vaTextData
- Notification function:** OnTextDataChanged
- Implementation:** ☒ Member variable, ☐ Stock, ☐ Get/Set methods
- Parameter list:** A table with two columns: Name and Type.

Name	Type

Figure 60: Adding TextData property information.



The 'Add Method' dialog box is shown with the following fields and options:

- External name:** DisplayDialog
- Internal name:** DisplayDialog
- Return type:** BOOL
- Implementation:** ☒ Custom, ☐ Stock
- Parameter list:** A table with two columns: Name and Type.

Name	Type

Figure 61: Adding DisplayDialog method information.

Add the `ExitInstance()` handler to `CMymfc29BApp` class.

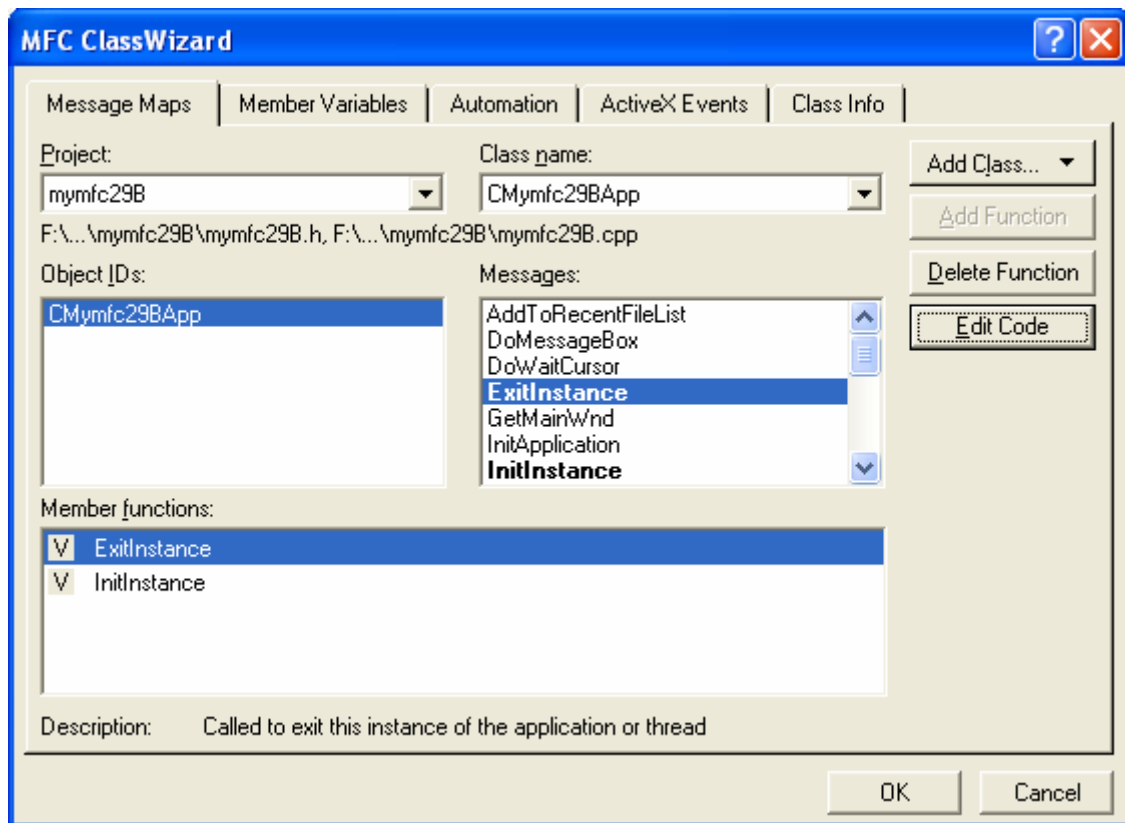


Figure 62: Adding `ExitInstance()` message handler.

Click the **Edit Code** button and add the following code.

```
int CMymfc29BApp::ExitInstance()
{
    // TODO: Add your specialized code here and/or call the base class
    TRACE("CMymfc29BApp::ExitInstance\n");
    return CWinApp::ExitInstance();
}

int CMymfc29BApp::ExitInstance()
{
    // TODO: Add your specialized code here and/or call the base class
    TRACE("CMymfc29BApp::ExitInstance\n");
    return CWinApp::ExitInstance();
}
```

Listing 10.

Modify the `InitInstance()`.

```
BOOL CMymfc29BApp::InitInstance()
{
    TRACE("CMymfc29BApp::InitInstance\n");
    // Register all OLE server (factories) as running. This
    // enables the OLE libraries to create objects from other
    // applications.
    COleObjectFactory::RegisterAll();

    return TRUE;
}
```

```

// CMyMfc29BApp initialization
BOOL CMyMfc29BApp::InitInstance()
{
    TRACE("CMyMfc29BApp::InitInstance\n");
    // Register all OLE server (factories) as running. This
    // enables the OLE libraries to create objects from other
    // applications.
    COleObjectFactory::RegisterAll();

    return TRUE;
}

```

Listing 11.

Add the `#include` statement for `AfxOleRegisterTypeLib()` in **StdAfx.h**

```
#include <afxctl.h>
```

```

#include <afxctl.h>           // for AfxOleRegisterTypeLib
|
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations

```

Listing 12.

Add the header file of the **PromptDl.h** into **Mymfc29BAuto.cpp** implementation file.

```

#include "stdafx.h"
#include "mymfc29B.h"
#include "Mymfc29BAuto.h"
#include "PromptDl.h"

#ifdef _DEBUG

```

Listing 13.

Add the following line of code in the constructor.

```

CMyMfc29BAuto::CMyMfc29BAuto()
{
    EnableAutomation();

    // To keep the application running as long as an OLE automation
    // object is active, the constructor calls AfxOleLockApp.
    ::VariantInit(&m_vaTextData); // necessary initialization
    m_lData = 0;

    AfxOleLockApp();
}

CMyMfc29BAuto::CMyMfc29BAuto()
{
    EnableAutomation();

    // To keep the application running as long as an OLE automation
    // object is active, the constructor calls AfxOleLockApp.
    |
    ::VariantInit(&m_vaTextData); // necessary initialization
    m_lData = 0;

    AfxOleLockApp();
}

```

Listing 14.

Add code for the property and method that is OnLongDataChanged(), OnTextDataChanged() and DisplayDialog().

```
void CMymfc29BAuto::OnLongDataChanged()
{
    // TODO: Add notification handler code
    TRACE("CMymfc29BAuto::OnLongDataChanged\n");
}

void CMymfc29BAuto::OnTextDataChanged()
{
    // TODO: Add notification handler code
    TRACE("CMymfc29BAuto::OnTextDataChanged\n");
}

// CMymfc29BAuto message handlers
void CMymfc29BAuto::OnLongDataChanged()
{
    // TODO: Add notification handler code
    TRACE("CMymfc29BAuto::OnLongDataChanged\n");
}

void CMymfc29BAuto::OnTextDataChanged()
{
    // TODO: Add notification handler code
    TRACE("CMymfc29BAuto::OnTextDataChanged\n");
}
```

Listing 15.

```
BOOL CMymfc29BAuto::DisplayDialog()
{
    // TODO: Add your dispatch handler code here
    TRACE("Entering CMymfc29BAuto::DisplayDialog %p\n", this);
    BOOL bRet = TRUE;
    AfxLockTempMaps(); // See MFC Tech Note #3
    CWnd* pTopWnd = CWnd::FromHandle(::GetTopWindow(NULL));
    try
    {
        CPromptDlg dlg /*(pTopWnd)*/;
        if (m_vaTextData.vt == VT_BSTR)
        {
            // converts double-byte character to single-byte
            // character
            dlg.m_strData = m_vaTextData.bstrVal;
        }
        dlg.m_lData = m_lData;

        if (dlg.DoModal() == IDOK)
        {
            m_vaTextData = COleVariant(dlg.m_strData).Detach();
            m_lData = dlg.m_lData;
            bRet = TRUE;
        }
        else
        {
            bRet = FALSE;
        }
    }
    catch (CException* pe)
```

```

    {
        TRACE("Exception: failure to display dialog\n");
        bRet = FALSE;
        pe->Delete();
    }
    AfxUnlockTempMaps();
    return bRet;
}

BOOL CMymfc29BAuto::DisplayDialog()
{
    // TODO: Add your dispatch handler code here
    TRACE("Entering CMymfc29BAuto::DisplayDialog %p\n", this);
    BOOL bRet = TRUE;
    AfxLockTempMaps(); // See MFC Tech Note #3
    CWnd* pTopWnd = CWnd::FromHandle(::GetTopWindow(NULL));
    try
    {
        CPromptDlg dlg /*(pTopWnd)*/;
        if (m_vaTextData.vt == VT_BSTR)
        {
            // converts double-byte character to single-byte character
            dlg.m_strData = m_vaTextData.bstrVal;
        }
        dlg.m_lData = m_lData;
        if (dlg.DoModal() == IDOK)
        {
            m_vaTextData = COleVariant(dlg.m_strData).Detach();
            m_lData = dlg.m_lData;
            bRet = TRUE;
        }
        else
        {
            bRet = FALSE;
        }
    }
    catch (CException* pe)
    {
        TRACE("Exception: failure to display dialog\n");
        bRet = FALSE;
        pe->Delete();
    }
    AfxUnlockTempMaps();
    return bRet;
}

```

Listing 16.

Build the program and register the DLL. Make sure there is no error during the program building and DLL registering.

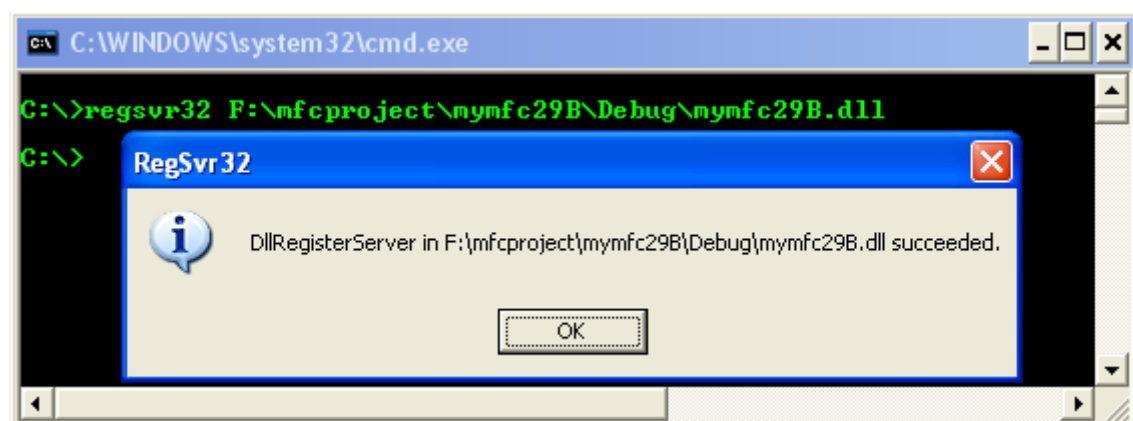


Figure 63: Registering **mymfc29B.dll** using **regsvr32** at command prompt.

Launch Excel and save the Workbook as [mymfc29B.xls](#). Launch the Visual Basic Editor, copy and paste the following three macros.

```
Dim Dllcomp As Object
Private Declare Sub CoFreeUnusedLibraries Lib "OLE32" ()

Sub LoadDllComp()
    Set Dllcomp = CreateObject("Mymfc29B.Auto")
    Range("B3").Select
    Dllcomp.LongData = Selection.Value
    Range("C3").Select
    Dllcomp.TextData = Selection.Value
End Sub

Sub RefreshDllComp() 'Gather Data button
    Range("B3").Select
    Dllcomp.LongData = Selection.Value
    Range("C3").Select
    Dllcomp.TextData = Selection.Value
    Dllcomp.DisplayDialog
    Range("B3").Select
    Selection.Value = Dllcomp.LongData
    Range("C3").Select
    Selection.Value = Dllcomp.TextData
End Sub

Sub UnloadDllComp()
    Set Dllcomp = Nothing
    Call CoFreeUnusedLibraries
End Sub
```

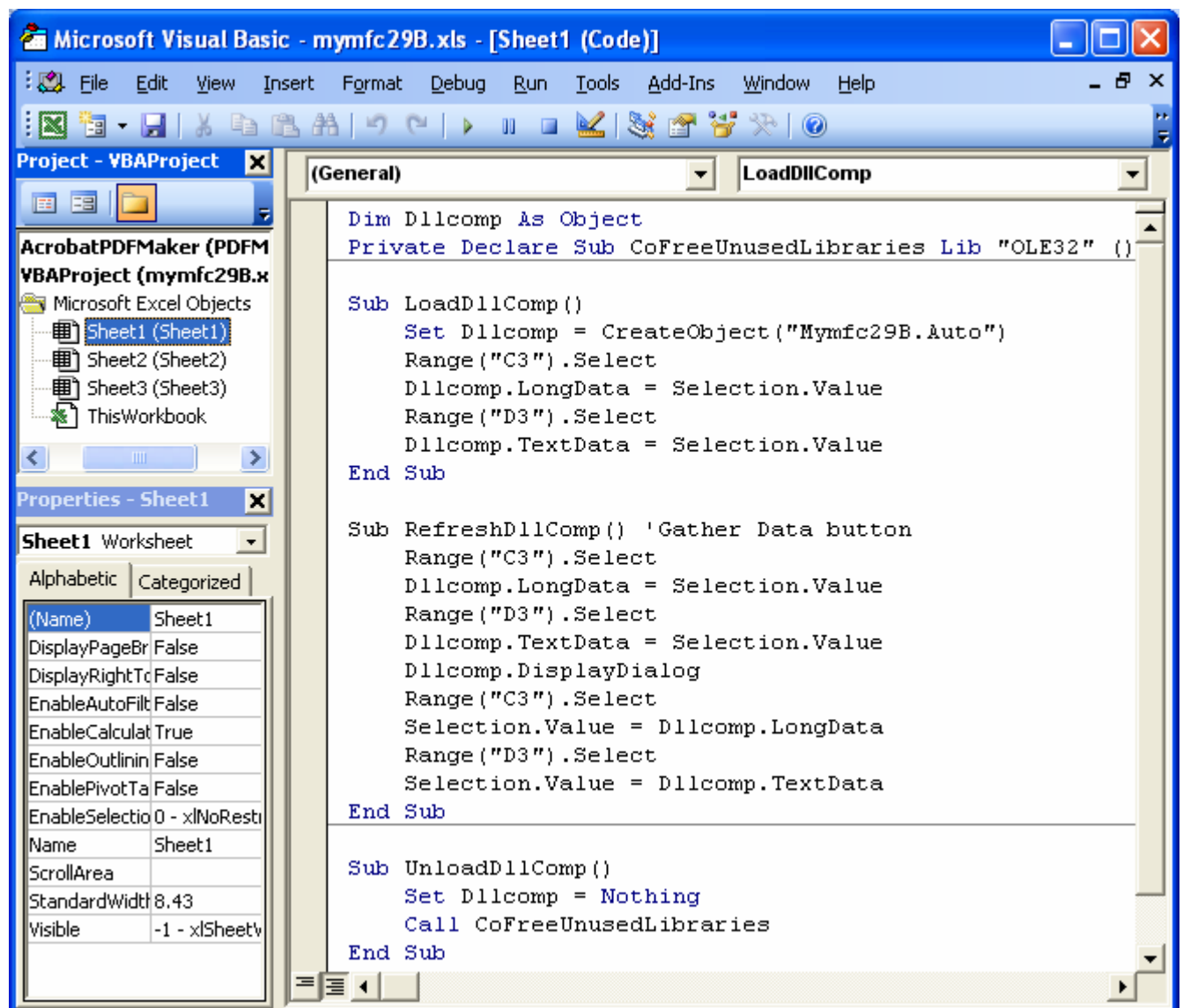



Figure 64: The VBA macro code.

Arrange the three push buttons as shown below and assign appropriate macro to the buttons. Be careful about the column and the button arrangement. They should match with the Excel cells code in macros.

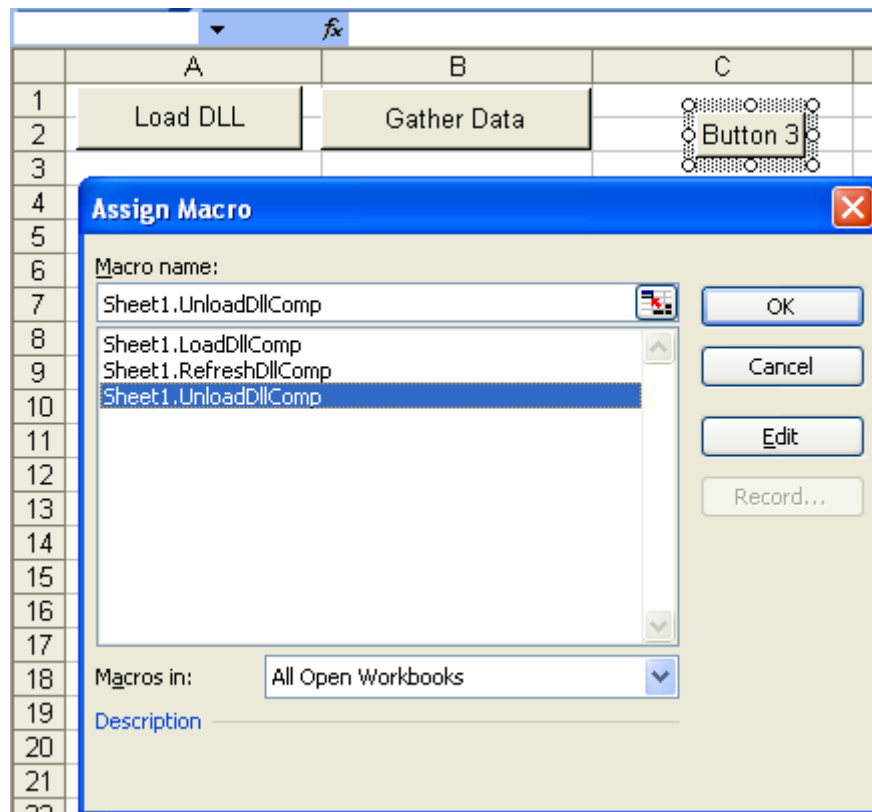


Figure 65: Attaching macro to button in Excel.

Finally we are ready to test our program. Before that, let have some story for what we have done.

The Story

The MYMFC29B program is fairly simple. An **Automation component class**, identified by the registered name **Mymfc29B.Auto**, has the following properties and method.

Method/property	Description
LongData	Long integer property.
TextData	VARIANT property.
DisplayDialog	Method - no parameters, BOOL return.

Table 4.

DisplayDialog displays the MYMFC29B data gathering dialog box shown in Figure 66. An Excel macro passes two cell values to the DLL and then updates the same cells with the updated values.

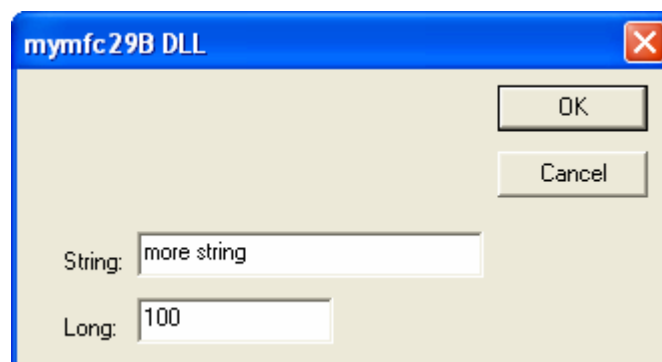


Figure 66: The MYMFC29B DLL dialog in action.

The example was first generated as an MFC AppWizard DLL with the **Regular DLL Using Shared MFC DLL** option and the **Automation** option selected.

Debugging a DLL Component

To debug a DLL, you must tell the debugger which EXE file to load. Choose **Settings** from **Visual C++'s Project** menu, and then enter the controller's full pathname (including the EXE extension) in the **Executable For Debug Session** box on the **Debug** page.

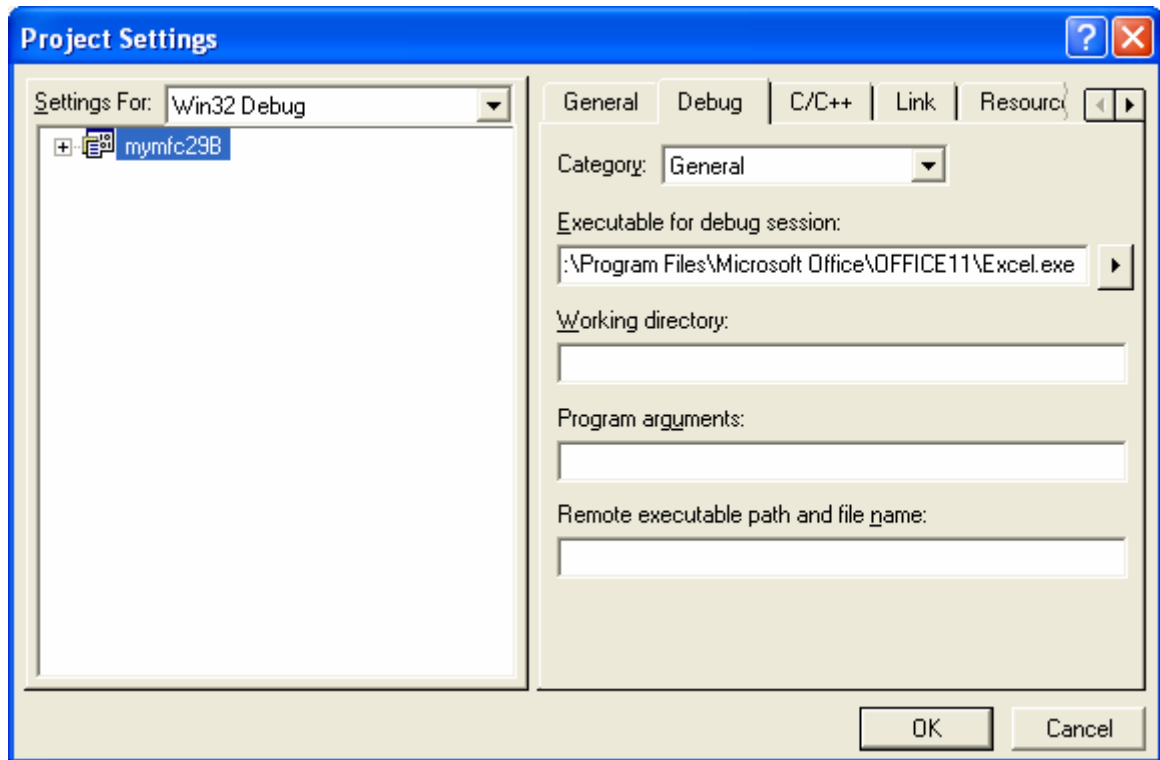


Figure 67: Changing the Visual C++ project Debug setting.

When you click the **Go** button on the **Debug** toolbar, your controller will start (loading the DLL as part of its process) and then wait for you to activate the component.

When you activate the component, your DLL in the debugger should then construct its component object. It might be a good idea to include a `TRACE` statement in the component object's constructor. Don't forget that your DLL must be registered before the client can load it.

Here's another option. If you have the source code for the client program, you can start the client program in the debugger. When the client loads the component DLL, you can see the output from the component program's `TRACE` statements.

Here are the steps for building and testing the MYMFC29B component DLL. From Visual C++, open the \mymfc29B.dsw workspace. Build the project. Register the DLL with the **regsvr32** utility as shown below.

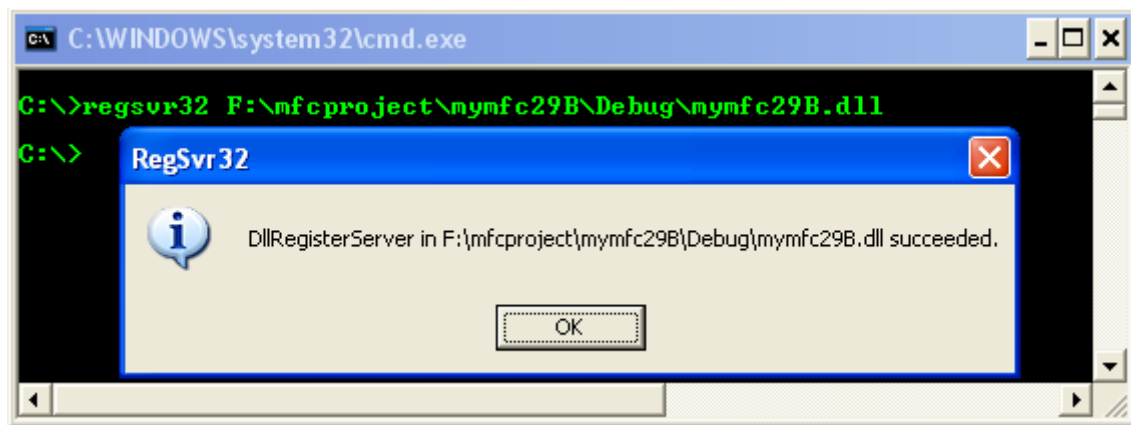


Figure 68: Registering **mymfc29B.dll** using regsvr32 at command prompt.

Start Excel, and then open the **mymfc29B.xls** workbook file.

Click the **Load DLL** button, and then click the **Gather Data** button and dialog as shown in Figure 70 is launched. Edit the **String** and **Long** data, click the **OK**, and watch the new values appear in the spreadsheet.

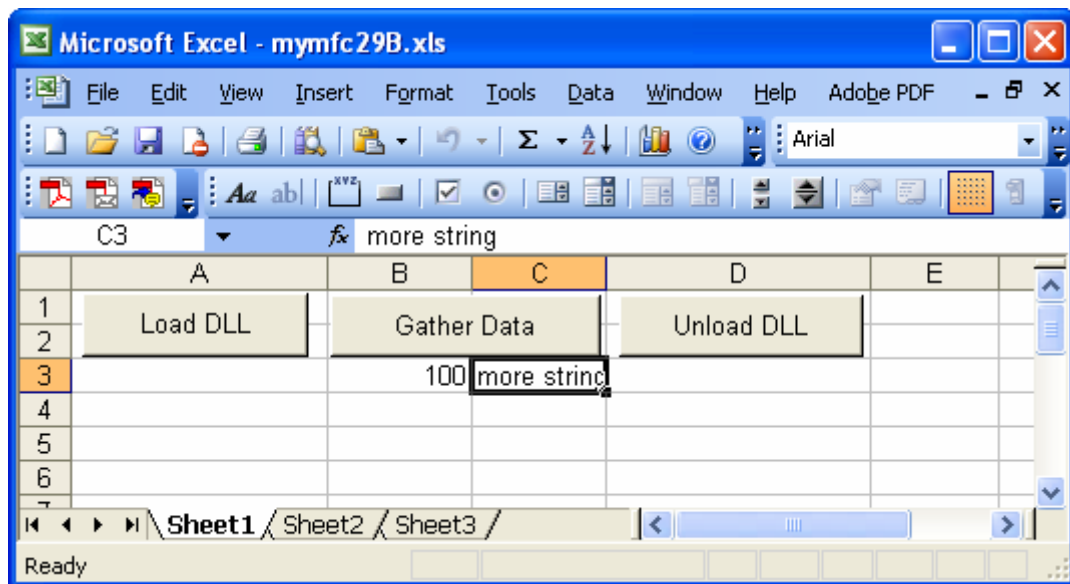


Figure 69: Excel as client to test MYMFC29B component.

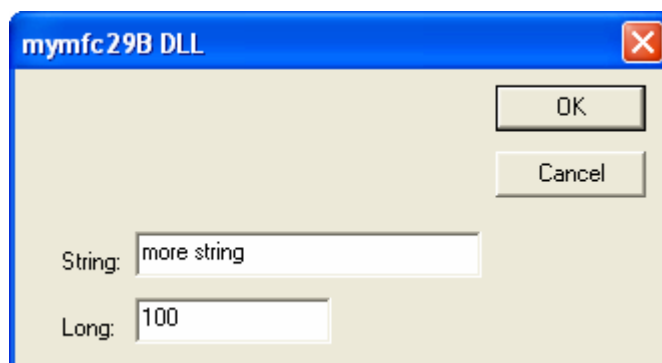


Figure 70: The MYMFC29B DLL dialog in action.

Click the **Unload DLL** button. If you've started the DLL (and Excel) from the debugger, you can watch the **Debug** window to be sure the DLL's `ExitInstance()` function is called.

Now let's look at the MYMFC29B code. Like an MFC EXE, an MFC regular DLL has an application class (derived from CWinApp) and a global application object. The overridden `InitInstance()` member function in **mymfc29B.cpp** looks like this:

```

        BOOL CMymfc29BApp::InitInstance()
        {
            TRACE("CMymfc29BApp::InitInstance\n");
            // Register all OLE server (factories) as running. This
            // enables the OLE libraries to create objects from other
            // applications.
            COleObjectFactory::RegisterAll();

            return TRUE;
        }

BOOL CMymfc29BApp::InitInstance()
{
    TRACE("CMymfc29BApp::InitInstance\n");

    // Register all OLE server (factories) as running. This enables the
    // OLE libraries to create objects from other applications.
    COleObjectFactory::RegisterAll();

    return TRUE;
}

```

Listing 17.

There's also an `ExitInstance()` function for diagnostic purposes only, as well as the following code for the three standard COM DLL exported functions:

```

STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllGetClassObject(rclsid, riid, ppv);
}

STDAPI DllCanUnloadNow(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllCanUnloadNow();
}

STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    COleObjectFactory::UpdateRegistryAll();
    // VERIFY(AfxOleRegisterTypeLib(AfxGetInstanceHandle(), theTypeLibGUID,
    // "mymfc29B.tlb"));
    return S_OK;
}

```

```

// Special entry points required for inproc servers
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid, LPVOID* ppv)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllGetClassObject(rclsid, riid, ppv);
}

STDAPI DllCanUnloadNow(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    return AfxDllCanUnloadNow();
}

// by exporting DllRegisterServer, you can use regsvr.exe
STDAPI DllRegisterServer(void)
{
    AFX_MANAGE_STATE(AfxGetStaticModuleState());
    COleObjectFactory::UpdateRegistryAll();
    //VERIFY(AfxOleRegisterTypeLib(AfxGetInstanceHandle(),
    //    theTypeLibGUID, "mymfc29B.tlb"));
    return S_OK;
}

int CMymfc29BApp::ExitInstance()

```

Listing 18.

The **PromptDL.cpp** file contains code for the **CPromptDlg** class, but that class is a standard class derived from **CDialog**. The file **PromptDL.h** contains the **CPromptDlg** class header.

The **CMymfc29BAuto** class, the Automation component class initially generated by ClassWizard (with the **Createable By Type ID** option), is more interesting. This class is exposed to COM under the program ID **mymfc29B.Auto**. Listing 19 below shows the header file **mymfc29BAuto.h**.

```

MYMFC29BAUTO.H

#ifndef AFX_MYMFC29BAUTO_H__DE94AEB7_260A_4D9D_BB2C_7023D4D6BBCA__INCLUDED_
#define AFX_MYMFC29BAUTO_H__DE94AEB7_260A_4D9D_BB2C_7023D4D6BBCA__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// Mymfc29BAuto.h : header file
//

////////////////////
// CMymfc29BAuto command target

class CMymfc29BAuto : public CCmdTarget
{
    DECLARE_DYNCREATE(CMymfc29BAuto)

    CMymfc29BAuto();           // protected constructor used by dynamic
creation

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{AFX_VIRTUAL(CMymfc29BAuto)
public:

```

```

        virtual void OnFinalRelease();
        //{AFX_VIRTUAL

// Implementation
protected:
    virtual ~CMymfc29BAuto();

    // Generated message map functions
    //{AFX_MSG(CMymfc29BAuto)
    // NOTE - the ClassWizard will add and remove member functions here.
    //{AFX_MSG

    DECLARE_MESSAGE_MAP()
    DECLARE_OLECREATE(CMymfc29BAuto)

    // Generated OLE dispatch map functions
    //{AFX_DISPATCH(CMymfc29BAuto)
    long m_lData;
    afx_msg void OnLongDataChanged();
    VARIANT m_vaTextData;
    afx_msg void OnTextDataChanged();
    afx_msg BOOL DisplayDialog();
    //{AFX_DISPATCH
    DECLARE_DISPATCH_MAP()
    DECLARE_INTERFACE_MAP()

};

////////////////////////////////////

//{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the
previous line.

#endif //
!defined(AFX_MYMFC29BAUTO_H__DE94AEB7_260A_4D9D_BB2C_7023D4D6BBCA__INCLUDED_)

```

Listing 19: Excerpt from the **mymfc29BAuto.h** header file.

Listing 20 shows the implementation file **mymfc29BAuto.cpp**.

```

MYMFC29BAUTO.CPP

// Mymfc29BAuto.cpp : implementation file
//

#include "stdafx.h"
#include "mymfc29B.h"
#include "Mymfc29BAuto.h"
#include "PromptDl.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

////////////////////////////////////
// CMymfc29BAuto

IMPLEMENT_DYNCREATE(CMymfc29BAuto, CCmdTarget)

CMymfc29BAuto::CMymfc29BAuto()
{
    EnableAutomation();

    // To keep the application running as long as an OLE automation

```

```

        //      object is active, the constructor calls AfxOleLockApp.
        :VariantInit(&m_vaTextData); // necessary initialization
        m_lData = 0;

        AfxOleLockApp();
    }

CMymfc29BAuto::~CMymfc29BAuto()
{
    // To terminate the application when all objects created with
    //      with OLE automation, the destructor calls AfxOleUnlockApp.

    AfxOleUnlockApp();
}

void CMymfc29BAuto::OnFinalRelease()
{
    // When the last reference for an automation object is released
    // OnFinalRelease is called. The base class will automatically
    // deletes the object. Add additional cleanup required for your
    // object before calling the base class.

    CCmdTarget::OnFinalRelease();
}

BEGIN_MESSAGE_MAP(CMymfc29BAuto, CCmdTarget)
    //{AFX_MSG_MAP(CMymfc29BAuto)
    // NOTE - the ClassWizard will add and remove mapping macros here.
    //}AFX_MSG_MAP
END_MESSAGE_MAP()

BEGIN_DISPATCH_MAP(CMymfc29BAuto, CCmdTarget)
    //{AFX_DISPATCH_MAP(CMymfc29BAuto)
    DISP_PROPERTY_NOTIFY(CMymfc29BAuto, "LongData", m_lData, OnLongDataChanged,
VT_I4)
    DISP_PROPERTY_NOTIFY(CMymfc29BAuto, "TextData", m_vaTextData,
OnTextDataChanged, VT_VARIANT)
    DISP_FUNCTION(CMymfc29BAuto, "DisplayDialog", DisplayDialog, VT_BOOL,
VTS_NONE)
    //}AFX_DISPATCH_MAP
END_DISPATCH_MAP()

// Note: we add support for IID_IMymfc29BAuto to support type safe binding
// from VBA. This IID must match the GUID that is attached to the
// dispinterface in the .ODL file.

// {7A97BA38-BF4A-4586-93C6-72B5EE7E0DC2}
static const IID IID_IMymfc29BAuto =
{ 0x7a97ba38, 0xbf4a, 0x4586, { 0x93, 0xc6, 0x72, 0xb5, 0xee, 0x7e, 0xd, 0xc2 } };

BEGIN_INTERFACE_MAP(CMymfc29BAuto, CCmdTarget)
    INTERFACE_PART(CMymfc29BAuto, IID_IMymfc29BAuto, Dispatch)
END_INTERFACE_MAP()

// {39D9E31F-25CB-4511-B5A8-5406E29BA565}
IMPLEMENT_OLECREATE(CMymfc29BAuto, "mymfc29B.Auto", 0x39d9e31f, 0x25cb, 0x4511,
0xb5, 0xa8, 0x54, 0x6, 0xe2, 0x9b, 0xa5, 0x65)

////////////////////////////////////
// CMymfc29BAuto message handlers

void CMymfc29BAuto::OnLongDataChanged()
{
    // TODO: Add notification handler code
    TRACE("CMymfc29BAuto::OnLongDataChanged\n");
}

```



```

void CMymfc29BAuto::OnTextDataChanged()
{
    // TODO: Add notification handler code
    TRACE( "CMymfc29BAuto::OnTextDataChanged\n" );
}

BOOL CMymfc29BAuto::DisplayDialog()
{
    // TODO: Add your dispatch handler code here
    TRACE("Entering CMymfc29BAuto::DisplayDialog %p\n", this);
    BOOL bRet = TRUE;
    AfxLockTempMaps(); // See MFC Tech Note #3
    CWnd* pTopWnd = CWnd::FromHandle(::GetTopWindow(NULL));
    try
    {
        CPromptDlg dlg /*(pTopWnd)*/;
        if (m_vaTextData.vt == VT_BSTR)
        {
            // converts double-byte character to single-byte
            // character
            dlg.m_strData = m_vaTextData.bstrVal;
        }
        dlg.m_lData = m_lData;

        if (dlg.DoModal() == IDOK)
        {
            m_vaTextData = COleVariant(dlg.m_strData).Detach();
            m_lData = dlg.m_lData;
            bRet = TRUE;
        }
        else
        {
            bRet = FALSE;
        }
    }
    catch (CException* pe)
    {
        TRACE("Exception: failure to display dialog\n");
        bRet = FALSE;
        pe->Delete();
    }
    AfxUnlockTempMaps();
    return bRet;
}

```

Listing 20: The **mymfc29BAuto.cpp** implementation file.

The two properties, LongData and TextData, are represented by class data members `m_lData` and `m_vaTextData`, both initialized in the constructor. When the LongData property was added in ClassWizard, a notification function, `OnLongDataChanged()`, was specified. This function is called whenever the controller changes the property value. Notification functions apply only to properties that are represented by data members. Don't confuse this notification with the notifications that ActiveX controls give their container when a bound property changes.

The `DisplayDialog()` member function, which is the `DisplayDialog` method, is ordinary except that the `AfxLockTempMaps()` and `AfxUnlockTempMaps()` functions are necessary for cleaning up temporary object pointers that would normally be deleted in an EXE program's idle loop.

What about the Excel VBA code? Here are the three macros and the global declarations:

```

Dim Dllcomp As Object
Private Declare Sub CoFreeUnusedLibraries Lib "OLE32"()

Sub LoadDllComp()
    Set Dllcomp = CreateObject("Mymfc29B.Auto")

```

```

Range("B3").Select
Dllcomp.LongData = Selection.Value
Range("C3").Select
Dllcomp.TextData = Selection.Value
End Sub

Sub RefreshDllComp() 'Gather Data button
Range("B3").Select
Dllcomp.LongData = Selection.Value
Range("C3").Select
Dllcomp.TextData = Selection.Value
Dllcomp.DisplayDialog
Range("B3").Select
Selection.Value = Dllcomp.LongData
Range("C3").Select
Selection.Value = Dllcomp.TextData
End Sub

Sub UnloadDllComp()
Set Dllcomp = Nothing
Call CoFreeUnusedLibraries
End Sub

```

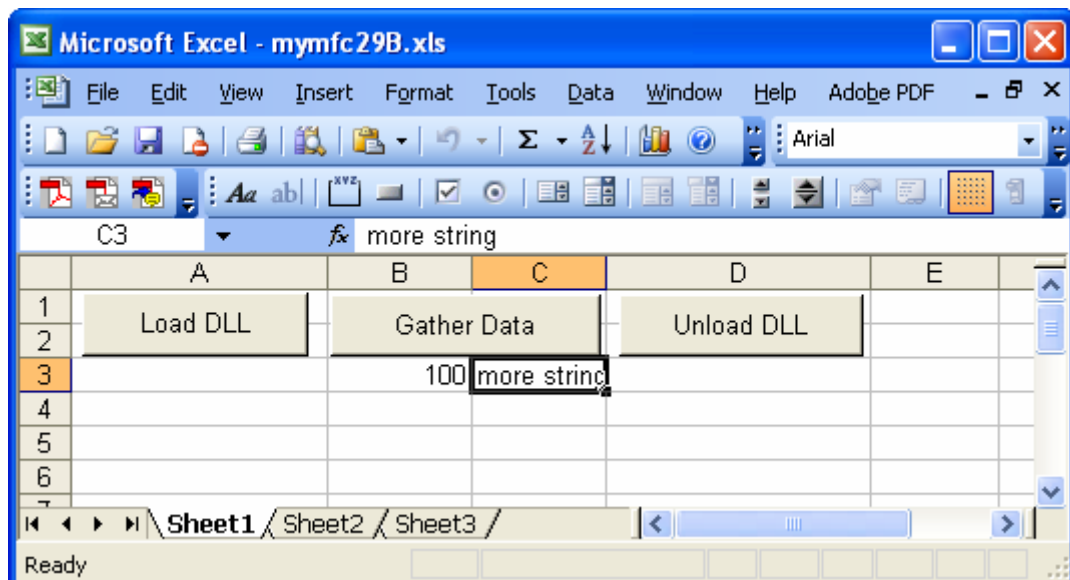


Figure 71: Testing MYMFC29B component using Excel as client.

The first line in `LoadDllComp()` creates a component object as identified by the registered name **Mymfc29B.Auto**. The `RefreshDllComp()` macro accesses the component object's `LongData` and `TextData` properties. The first time you run `LoadDllComp()`, it loads the DLL and constructs an `Mymfc29B.Auto` object. The second time you run `LoadDllComp()`, something curious happens: a second object is constructed, and the original object is destroyed. If you run `LoadDllComp()` from another copy of the workbook, you get two separate `Mymfc29B.Auto` objects. Of course, there's only one mapping of **mymfc29B.dll** in memory at any time unless you're running more than one Excel process.

Look closely at the `UnloadDllComp()` macro. When the "Set `Dllcomp = Nothing`" statement is executed, the DLL is disconnected, but it's not unmapped from Excel's address space, which means the component's `ExitInstance()` function is not called. The `CoFreeUnusedLibraries()` function calls the exported `DllCanUnloadNow()` function for each component DLL and, if that function returns `TRUE`, `CoFreeUnusedLibraries()` frees the DLL. MFC programs call `CoFreeUnusedLibraries()` in the idle loop (after a one-minute delay), but Excel doesn't. That's why `UnloadDllComp()` must call `CoFreeUnusedLibraries()` after disconnecting the component.

The `CoFreeUnusedLibraries()` function doesn't do anything in Windows NT 3.51 unless you have Service Pack 2 (SP2) installed.

IDispatch Interface Info

IDispatch interface exposes objects, methods and properties to programming tools and other applications that support Automation. COM components implement the IDispatch interface to enable access by Automation clients, such as Visual Basic. Methods in Vtable Order is listed below

Unknown Methods	Description
QueryInterface()	Returns pointers to supported interfaces.
AddRef()	Increments reference count.
Release()	Decrements reference count.

Table 5.

IDispatch Methods	Description
GetTypeInfoCount()	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
TypeInfo()	Gets the type information for an object.
GetIDsOfNames()	Maps a single member and an optional set of argument names to a corresponding set of integer DISPID.
Invoke()	Provides access to properties and methods exposed by an object.

Table 6.

IDispatch is located in the **Oleauto.h** header file on 32-bit systems and in **Dispatch.h** on 16-bit and Macintosh systems. The following Tables summarize the methods information. The QueryInterface(), AddRef() and Release() have been explained in the previous module.

Item	Description
Function	IDispatch::TypeInfoCount
Use	Retrieves the number of type information interfaces that an object provides (either 0 or 1).
Prototype	<code>HRESULT TypeInfoCount(unsigned int FAR* pctinfo);</code>
Parameters	<code>pctinfo</code> Points to a location that receives the number of type information interfaces provided by the object. If the object provides type information, this number is 1; otherwise the number is 0.
Return value	S_OK - Success. E_NOTIMPL - Failure.
Include file	-
Remark	<p>The function may return zero, which indicates that the object does not provide any type information. In this case, the object may still be programmable through IDispatch or a VTBL, but does not provide run-time type information for browsers, compilers, or other programming tools that access type information. This can be useful for hiding an object from browsers.</p> <p>Example</p> <p>This code from the MSDN's Lines sample file Lines.cpp implements the TypeInfoCount() member function for the CLines class (ActiveX or OLE object).</p> <pre>STDMETHODIMP CLines::TypeInfoCount(UINT FAR* pctinfo) { if (pctinfo == NULL) { return E_INVALIDARG; } *pctinfo = 1; return NOERROR; }</pre>

Table 7.

Item	Description
Function	IDispatch::GetTypeInfo
Use	Retrieves the type information for an object, which can then be used to get the type information for an interface.
Prototype	<code>HRESULT GetTypeInfo(unsigned int iTypeInfo, LCID lcid, ITypeInfo FAR* ppTypeInfo);</code>
Parameters	<p><code>iTypeInfo</code> The type information to return. Pass 0 to retrieve type information for the IDispatch implementation.</p> <p><code>lcid</code> The locale identifier for the type information. An object may be able to return different type information for different languages. This is important for classes that support localized member names. For classes that do not support localized member names, this parameter can be ignored.</p> <p><code>ppTypeInfo</code> Receives a pointer to the requested type information object.</p>
Return value	<p>The return value obtained from the returned HRESULT is one of the following:</p> <p>S_OK - Success; the type information element exists.</p> <p>DISP_E_BADINDEX - Failure; iTypeInfo argument was not 0.</p>
Include file	-
Remark	<p>Example</p> <p>The following code from the sample file MSDN's Lines.cpp implements the member function GetTypeInfo():</p> <pre>// This function implements GetTypeInfo for the CLines // collection. STDMETHODIMP CLines::GetTypeInfo(UINT iTypeInfo, LCID lcid, ITypeInfo FAR* ppTypeInfo) { if (ppTypeInfo == NULL) return E_INVALIDARG; *ppTypeInfo = NULL; if (iTypeInfo != 0) return DISP_E_BADINDEX; m_ptinfo->AddRef(); // AddRef and return pointer to cached // typeinfo for this object. *ppTypeInfo = m_ptinfo; return NOERROR; }</pre>

Table 8.

Item	Description
Function	IDispatch::GetIDsOfNames
Use	Maps a single member and an optional set of argument names to a corresponding set of integer DISPIDs, which can be used on subsequent calls to IDispatch::Invoke. The dispatch function DispGetIDsOfNames() provides a standard implementation of GetIDsOfNames().
Prototype	<code>HRESULT GetIDsOfNames(</code>

	<pre> REFIID riid, OLECHAR FAR* FAR* rgpszNames, unsigned int cNames, LCID lcid, DISPID FAR* rgDispId); </pre>
Parameters	<p>riid Reserved for future use. Must be IID_NULL.</p> <p>rgpszNames Passed-in array of names to be mapped.</p> <p>cNames Count of the names to be mapped.</p> <p>lcid The locale context in which to interpret the names.</p> <p>rgDispId Caller-allocated array, each element of which contains an identifier (ID) corresponding to one of the names passed in the rgpszNames array. The first element represents the member name. The subsequent elements represent each of the member's parameters.</p>
Return value	<p>The return value obtained from the returned HRESULT is one of the following:</p> <p>S_OK - Success.</p> <p>E_OUTOFMEMORY - Out of memory.</p> <p>DISP_E_UNKNOWNNAME - One or more of the names were not known. The returned array of DISPIDs contains DISPID_UNKNOWN for each entry that corresponds to an unknown name.</p> <p>DISP_E_UNKNOWNLCID - The locale identifier (LCID) was not recognized.</p>
Include file	-
Remark	<p>An IDispatch implementation can associate any positive integer ID value with a given name. Zero is reserved for the default, or Value property; -1 is reserved to indicate an unknown name; and other negative values are defined for other purposes. For example, if GetIDsOfNames () is called, and the implementation does not recognize one or more of the names, it returns DISP_E_UNKNOWNNAME, and the rgDispId array contains DISPID_UNKNOWN for the entries that correspond to the unknown names.</p> <p>The member and parameter DISPIDs must remain constant for the lifetime of the object. This allows a client to obtain the DISPIDs once, and cache them for later use.</p> <p>When GetIDsOfNames () is called with more than one name, the first name (rgpszNames[0]) corresponds to the member name, and subsequent names correspond to the names of the member's parameters.</p> <p>The same name may map to different DISPIDs, depending on context. For example, a name may have a DISPID when it is used as a member name with a particular interface, a different ID as a member of a different interface, and different mapping for each time it appears as a parameter.</p> <p>GetIDsOfNames () is used when an IDispatch client binds to names at run time. To bind at compile time instead, an IDispatch client can map names to DISPIDs by using the type information interfaces. This allows a client to bind to members at compile time and avoid calling GetIDsOfNames () at run time.</p> <p>The implementation of GetIDsOfNames () is case insensitive. Users that need case-sensitive name mapping should use type information interfaces to map names to DISPIDs, rather than call GetIDsOfNames ().</p> <p>Examples</p> <p>The following code from the MSDN's Lines sample file Lines.cpp implements the GetIDsOfNames () member function for the CLine class. The ActiveX or OLE object uses the standard implementation, DispGetIDsOfNames (). This implementation relies on DispGetIDsOfNames () to validate input arguments. To help minimize security risks, include code that performs more robust validation of the input arguments.</p>

	<pre> STDMETHODIMP CLine::GetIDsOfNames(REFIID riid, OLECHAR FAR* FAR* rgpszNames, UINT cNames, LCID lcid, DISPID FAR* rgDispId) { return DispGetIDsOfNames(m_ptinfo, rgpszNames, cNames, rgDispId); } </pre> <p>The following code might appear in an ActiveX client that calls GetIDsOfNames to get the DISPID of the CLine Color property.</p> <pre> HRESULT hresult; IDispatch FAR* pdisp = (IDispatch FAR*)NULL; DISPID dispid; OLECHAR FAR* szMember = "color"; // Code that sets a pointer to the dispatch (pdisp) is omitted. hresult = pdisp->GetIDsOfNames(IID_NULL, &szMember, 1, LOCALE_SYSTEM_DEFAULT, &dispid); </pre>
--	--

Table 9.

Item	Description
Function	IDispatch::Invoke
Use	Provides access to properties and methods exposed by an object. The dispatch function DispInvoke() provides a standard implementation of IDispatch::Invoke.
Prototype	<pre> HRESULT Invoke(DISPID dispIdMember, REFIID riid, LCID lcid, WORD wFlags, DISPPARAMS FAR* pDispParams, VARIANT FAR* pVarResult, EXCEPINFO FAR* pExcepInfo, unsigned int FAR* puArgErr); </pre>
Parameters	<p>dispIdMember Identifies the member. Use GetIDsOfNames() or the object's documentation to obtain the dispatch identifier.</p> <p>riid Reserved for future use. Must be IID_NULL.</p> <p>lcid The locale context in which to interpret arguments. The lcid is used by the GetIDsOfNames() function, and is also passed to IDispatch::Invoke to allow the object to interpret its arguments specific to a locale. Applications that do not support multiple national languages can ignore this parameter.</p> <p>wFlags Flags describing the context of the Invoke() call, include:</p> <p>DISPATCH_METHOD - The member is invoked as a method. If a property has the same name, both this and the DISPATCH_PROPERTYGET flag may be set. DISPATCH_PROPERTYGET - The member is retrieved as a property or data member. DISPATCH_PROPERTYPUT - The member is changed as a property or data member. DISPATCH_PROPERTYPUTREF - The member is changed by a reference assignment, rather than a value assignment. This flag is valid only when the property accepts a reference to an object.</p>

	<p>pDispParams Pointer to a structure containing an array of arguments, an array of argument DISPIDs for named arguments, and counts for the number of elements in the arrays.</p> <p>pVarResult Pointer to the location where the result is to be stored or NULL if the caller expects no result. This argument is ignored if DISPATCH_PROPERTYPUT or DISPATCH_PROPERTYPUTREF is specified.</p> <p>pExcepInfo Pointer to a structure that contains exception information. This structure should be filled in if DISP_E_EXCEPTION is returned. Can be NULL.</p> <p>puArgErr The index within rgvarg of the first argument that has an error. Arguments are stored in pDispParams->rgvarg in reverse order, so the first argument is the one with the highest index in the array. This parameter is returned only when the resulting return value is DISP_E_TYPEMISMATCH or DISP_E_PARAMNOTFOUND. This argument can be set to null.</p>
Return value	<p>The return value obtained from the returned HRESULT is one of the following:</p> <p>S_OK - Success.</p> <p>DISP_E_BADPARAMCOUNT - The number of elements provided to DISPPARAMS is different from the number of arguments accepted by the method or property.</p> <p>DISP_E_BADVARTYPE - One of the arguments in rgvarg is not a valid variant type.</p> <p>DISP_E_EXCEPTION - The application needs to raise an exception. In this case, the structure passed in pExcepInfo should be filled in.</p> <p>DISP_E_MEMBERNOTFOUND - The requested member does not exist, or the call to Invoke () tried to set the value of a read-only property.</p> <p>DISP_E_NONAMEDARGS - This implementation of IDispatch does not support named arguments.</p> <p>DISP_E_OVERFLOW - One of the arguments in rgvarg could not be coerced to the specified type.</p> <p>DISP_E_PARAMNOTFOUND - One of the parameter DISPIDs does not correspond to a parameter on the method. In this case, puArgErr should be set to the first argument that contains the error.</p> <p>DISP_E_TYPEMISMATCH - One or more of the arguments could not be coerced. The index within rgvarg of the first parameter with the incorrect type is returned in the puArgErr parameter.</p> <p>DISP_E_UNKNOWNINTERFACE - The interface identifier passed in riid is not IID_NULL.</p> <p>DISP_E_UNKNOWNLCID - The member being invoked interprets string arguments according to the LCID, and the LCID is not recognized. If the LCID is not needed to interpret arguments, this error should not be returned.</p> <p>DISP_E_PARAMNOTOPTIONAL - A required parameter was omitted.</p>
Include file	-
Remark	<p>Generally, you should not implement Invoke () directly. Instead, use the dispatch interface create functions CreateStdDispatch () and DispInvoke ().</p> <p>If some application-specific processing needs to be performed before calling a member, the code should perform the necessary actions, and then call ITypeInfo::Invoke to invoke the member. ITypeInfo::Invoke acts exactly like IDispatch::Invoke. The standard implementations of IDispatch::Invoke created by CreateStdDispatch () and DispInvoke () defer to ITypeInfo::Invoke.</p> <p>In an ActiveX client, IDispatch::Invoke should be used to get and set the values of properties, or to call a method of an ActiveX object. The dispIdMember argument identifies the member to invoke. The DISPIDs that identify members are defined by the implementer of the object and can be determined by using the object's documentation, the IDispatch::GetIDsOfNames function, or the ITypeInfo interface.</p> <p>The information that follows addresses developers of ActiveX clients and others who use code to expose ActiveX objects. It describes the behavior that users of exposed objects should expect.</p>

Table 10.

Item	Description
Function	CreateStdDispatch()
Use	Creates a standard implementation of the IDispatch interface through a single function call. This simplifies exposing objects through Automation.
Prototype	<pre>HRESULT CreateStdDispatch(IUnknown FAR* punkOuter, void FAR* pvThis, ITypeInfo FAR* ptinfo, IUnknown FAR* FAR* ppunkStdDisp);</pre>
Parameters	<p>punkOuter Pointer to the object's IUnknown implementation.</p> <p>pvThis Pointer to the object to expose.</p> <p>ptinfo Pointer to the type information that describes the exposed object.</p> <p>ppunkStdDisp This is the private unknown for the object that implements the IDispatch interface QueryInterface() call. This pointer is Null if the function fails.</p>
Return value	<p>The return value obtained from the returned HRESULT is one of the following:</p> <p>S_OK - Success.</p> <p>E_INVALIDARG - One of the first three arguments is invalid.</p> <p>E_OUTOFMEMORY - There was insufficient memory to complete the operation.</p>
Include file	Declared in oleauto.h, use oleaut32.lib
Remark	<p>You can use CreateStdDispatch() when creating an object instead of implementing the IDispatch member functions for the object. However, the implementation that CreateStdDispatch() creates has these limitations:</p> <ul style="list-style-type: none"> Supports only one national language. Supports only dispatch-defined exception codes returned from Invoke(). <p>LoadTypeLib(), GetTypeInfoOfGuid(), and CreateStdDispatch() comprise the minimum set of functions that you need to call to expose an object using a type library. CreateDispTypeInfo() and CreateStdDispatch() comprise the minimum set of dispatch components you need to call to expose an object using type information provided by the INTERFACEDATA structure.</p>
Example	<p>The following code implements the IDispatch interface for the CCalc class using CreateStdDispatch().</p> <pre>CCalc FAR* CCalc::Create() { HRESULT hresult; CCalc FAR* pcalc; CArith FAR* parith; ITypeInfo FAR* ptinfo; IUnknown FAR* punkStdDisp; extern INTERFACEDATA NEARDATA g_idataCCalc; if((pcalc = new FAR CCalc()) == NULL) return NULL; pcalc->AddRef(); parith = &(pcalc->m_arith);</pre>


```

// Build type information for the functionality on this object that
// is being exposed for external programmability.
hresult = CreateDispTypeInfo(
    &g_idataCCalc, LOCALE_SYSTEM_DEFAULT, &ptinfo);
if(hresult != NOERROR)
    goto LError0;

// Create an aggregate with an instance of the default
// implementation of IDispatch that is initialized with
// type information.
hresult = CreateStdDispatch(
    pcalc,          // Controlling unknown.
    parith,         // Instance to dispatch on.
    ptinfo,         // Type information describing the instance.
    &punkStdDisp);

ptinfo->Release();

if(hresult != NOERROR)
    goto LError0;

pcalc->m_punkStdDisp = punkStdDisp;

return pcalc;

LError0:;
pcalc->Release();
return NULL;
}

```

Table 11.

IDispatchEx Interface

IDispatchEx, an extension of the IDispatch interface, supports features appropriate for dynamic languages such as scripting languages. This section describes the IDispatchEx interface itself, the differences between IDispatch and IDispatchEx, and the rationale for the extensions. It is expected that readers are familiar with IDispatch and have access to the IDispatch documentation.

IDispatch was developed essentially for Microsoft® Visual Basic®. The primary limitation of IDispatch is that it assumes that objects are static. In other words, since objects do not change during run time, type information can fully describe them at compile time. Dynamic run-time models that are found in scripting languages such as Visual Basic Scripting Edition (VBScript) and JScript and object models such as Dynamic HTML require a more flexible interface.

IDispatchEx was developed to provide all the services of IDispatch as well as some extensions that are appropriate for more dynamic late-bound languages such as scripting languages. The additional features of IDispatchEx beyond those provided by IDispatch are:

- Add new members to an object ("expando") — use GetDispID() with the fdexNameEnsure flag.
- Delete members of an object — use DeleteMemberByName() or DeleteMemberByDispID().
- Case-sensitive dispatch operations — use fdexNameCaseSensitive or fdexNameCaseInsensitive.
- Search for member with implicit name — use fdexNameImplicit.
- Enumerate DISPID's of an object — use GetNextDispID().
- Map from DISPID to element name — use GetMemberName().
- Obtain properties of object members — use GetMemberProperties().
- Method invocation with this pointer — use InvokeEx() with DISPATCH_METHOD.
- Allow browsers that support the concept of name spaces to obtain the name space parent of an object, use GetNameSpaceParent().

Objects that support `IDispatchEx` might also support `IDispatch` for backward compatibility. The dynamic nature of objects that support `IDispatchEx` has a few implications for the `IDispatch` interface of those objects. For example, `IDispatch` makes the following assumption:

The member and parameter `DISPIDs` must remain constant for the lifetime of the object. This allows a client to obtain `DISPIDs` once and cache them for later use.

Since `IDispatchEx` allows the addition and deletion of members, the set of valid `DISPIDs` does not remain constant. However, `IDispatchEx` requires that the mapping between `DISPID` and member name remain constant. This means that if a member is deleted:

- The `DISPID` cannot be reused until a member with the same name is created.
- The `DISPID` must remain valid for `GetNextDispID()`.
- The `DISPID` must be accepted gracefully by any of the `IDispatch`/`IDispatchEx` methods; they must recognize the member as deleted and return an appropriate error code (usually `DISP_E_MEMBERNOTFOUND` or `S_FALSE`).

IDispatchEx Methods

The following are `IDispatchEx` methods.

Method	Description
<code>IDispatchEx::DeleteMemberByDispID</code>	Deletes a member by <code>DISPID</code> .
<code>IDispatchEx::DeleteMemberByName</code>	Deletes a member by name.
<code>IDispatchEx::GetDispID</code>	Maps a single member name to its corresponding <code>DISPID</code> , which can then be used on subsequent calls to <code>IDispatchEx::InvokeEx</code> .
<code>IDispatchEx::GetMemberName</code>	Retrieves the name of a member.
<code>IDispatchEx::GetMemberProperties</code>	Retrieves a member's properties.
<code>IDispatchEx::GetNamespaceParent</code>	Retrieves the interface for the namespace parent of an object.
<code>IDispatchEx::GetNextDispID</code>	Enumerates the members of the object.
<code>IDispatchEx::InvokeEx</code>	Provides access to properties and methods exposed by an <code>IDispatchEx</code> object.

Table 12.

Continue on next module...part 2

-----End Automation part 1-----

Further reading and digging:

1. [DCOM](#) at MSDN.
2. [COM+](#) at MSDN.
3. [COM](#) at MSDN.
4. [MSDN MFC 6.0 class library online documentation](#) - used throughout this Tutorial.
5. [MSDN MFC 7.0 class library online documentation](#) - used in .Net framework and also backward compatible with 6.0 class library
6. [MSDN Library](#)
7. [Windows data type](#).
8. [Win32 programming Tutorial](#).
9. [The best of C/C++, MFC, Windows and other related books](#).
10. Unicode and Multibyte character set: [Story](#) and [program examples](#).