

Archived content. No warranty is made as to technical accuracy. Content may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Visual Basic 4.0 Technical Articles

Creating OLE Servers in Visual Basic to Simplify Windows Function Calls

Ken Lassesen

Microsoft Developer Network Technology Group

July 7, 1995

Abstract

This article explains how to build both 16-bit and 32-bit out-of-process OLE servers in Visual Basic® version 4.0, servers that can encapsulate common Microsoft® Windows® application programming interface (API) calls or functions in third-party dynamic-link libraries (DLLs). Creating an OLE server wrapper around Windows calls permits 16-bit applications to call 32-bit API functions and 32-bit applications to call 16-bit API functions. The API call is also simplified by:

- The addition of named properties to the API call.
- The removal of memory and structure allocations from your project.
- The removal of API declarations from your project.

The reader is assumed to have passed Microsoft Certification in Microsoft Visual Basic for Windows version 3.0 or have equivalent knowledge in making Windows API calls from Visual Basic.

The API Journey Continues

This article describes how to build a simple out-of-process OLE server in Visual Basic® that allows you to make application programming interface (API) calls using this OLE server's properties and methods instead of using Declare statements. The Win32® Software Development Kit (SDK) and Win16 SDK are procedure-oriented and not object-oriented the way OLE servers are. I use OLE servers to place C-like calls out of sight and to stop rewriting the code to make the same API calls constantly.

For most Microsoft® Office developers, the most important feature of an OLE server is that with an out-of-process server (an executable), an application can call a 16-bit API from a 32-bit application or call a 32-bit API from a 16-bit application with no difficulty. OLE will move the data across processes automatically—no thinking or determining the "bitness" of the engine by the developer. (See my articles listed in the "Bibliography" section for further information on this issue.) A second feature of this OLE server is the addition of named properties for API arguments (and the changing of some arguments to being optional). A third feature of this OLE server is the ability to access the API calls as classes of functions—classes that can reflect your view of API calls. The fourth feature of this OLE server is the simple way it provides for other developers to leverage the API knowledge of an expert. All of these features do come with the cost of slightly slower execution of API calls and slightly greater use of memory.

The OLE server implements an approach similar to the one I used in my earlier article ["Creating Useful Native Visual Basic and Microsoft Access Functions."](#) There is no need for the application to allocate space for the API-call arguments because the OLE server will allocate the space for the arguments and Visual Basic for Applications, Access Basic, and Visual Basic will release the space. This approach can also be used in C language as an alternative to thinking.

Using an OLE server is a better approach for the 16-bit/32-bit interoperability problems than those described in my articles "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office" and "Corporate Developer's Guide to Office 95 API Issues." There is no need for the solution code to determine the "bitness" of the application engine.

Creating an OLE server is so easy in Visual Basic that it should be taught in every beginning Visual Basic class!

A Map for the Journey

For an OLE server to be callable from both 16-bit and 32-bit applications, the server must be out-of-process—that is, an executable (.EXE) file. This article will show you how to:

- Create an invisible OLE server in Visual Basic that can make an API call.
- Use this OLE server from an Visual Basic for Applications product (Microsoft Excel) and from Microsoft Access version 2.0.

Before proceeding too far, let's look at how our solution code will change in appearance. Calling the GetProfileString function in Visual Basic for Applications has resulted in code like the following:

```
REM Following line is needed ONCE in project
Declare Function GetProfileString Lib "Kernel" (ByVal lpAppName As String,
    ByVal lpKeyName As String, ByVal lpDefault As String, ByVal lpReturnedString
    As String, ByVal nSize As Integer) As Integer
.
.
.
Dim Buffer As String * 255, BufferLen As Integer, RC%, Default As String
BufferLen=255
RC% = GetProfileString("TEST", "KEY", "", Buffer, BufferLen)
If RC% > 0 Then
    TestKey$ = Left$(Buffer, RC%)
Else
    TestKey$ = ""
End If
```

When an OLE server is used, the code becomes much simpler and much clearer:

```
Dim WinIni As New MyWinAPI.WinINI
.
.
.
TestKey$=WinIni.Value(AppName:="TEST", KeyName:="KEY")
```

MyWinAPI.WinINI is the OLE server class used for API calls. After you declare and create an instance of the OLE server, the server's Value property calls the appropriate API automatically. The use of named properties makes the code easier to read and understand. The OLE server contains any needed declarations or type libraries to actually make the API call.

So, how do we make this technological miracle happen?

Creating an API OLE Server in Visual Basic

I will start by assuming that you are an experienced programmer in Visual Basic 3.0 who has just opened up Visual Basic 4.0 Professional or Enterprise Edition and that you have not had time to look at the manuals. I will go step by step through the process of building an out-of-process OLE server by building a OLE server that has one class module, WinIni, and one sample property, Value. For further details on building OLE servers, see the book *Creating OLE Servers in the Visual Basic 4.0 documentation*.

Follow these four steps to construct an invisible OLE server in Visual Basic:

1. Create an OLE server project. (I use the 16-bit version of Visual Basic for this sample.)
2. Create a Main subroutine.
3. Create a class module for the properties and methods.
4. Create the properties and methods in this class module of the OLE server.

Steps 3 and 4 may be repeated in one project as often as needed. Once the server's code is completed, it can be used as an OLE server.

Creating an OLE Server Project

The first step is to create an OLE server project:

1. Start a new project in Visual Basic 4.0.
2. From the Tools menu, choose Options, and then click the Project tab.
3. Select Sub Main in the Startup Form list.
4. Type MyWinAPI in the Project Name text box. This is the name of the OLE object.
5. Type My First WINAPI OLE Server in the Application Description text box.
6. Click OLE Server in the StartMode group, and click OK. See Figure 1.

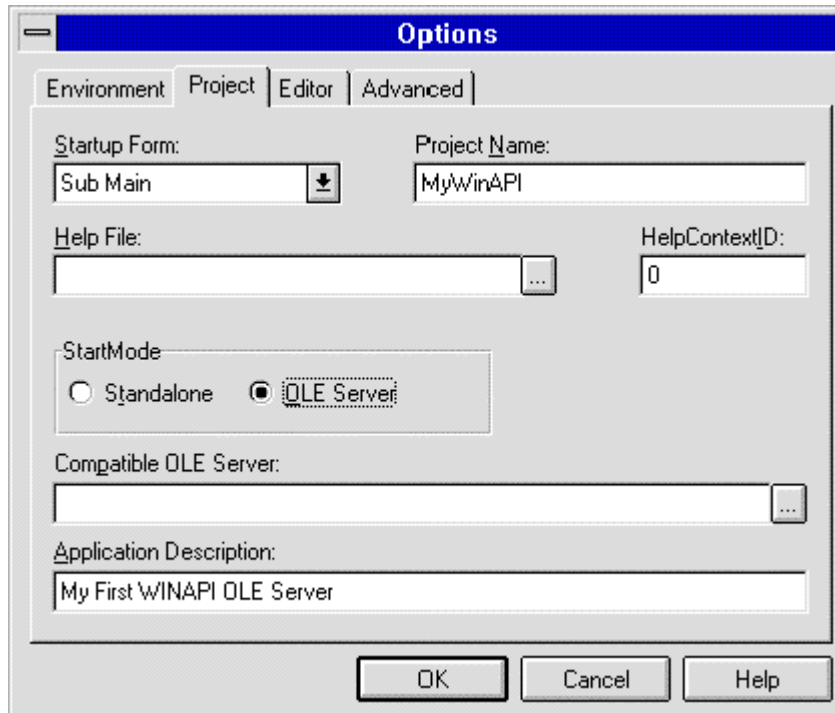


Figure 1. The Options dialog box set for an OLE server project

Creating a Main Subroutine

The second step is to delete the default Form1 and create a Main subroutine.

1. From the File menu, choose Remove File.
2. From the Insert menu, choose Module.
3. From the Insert menu, choose Procedure. In the Insert Procedure dialog box, type Main in the Name text box, click Sub in the Type group, click Public in the Scope group, and then click OK. See Figure 2.

I will leave this procedure empty so that the executable terminates if directly executed. You may wish to insert an informative message box instead.

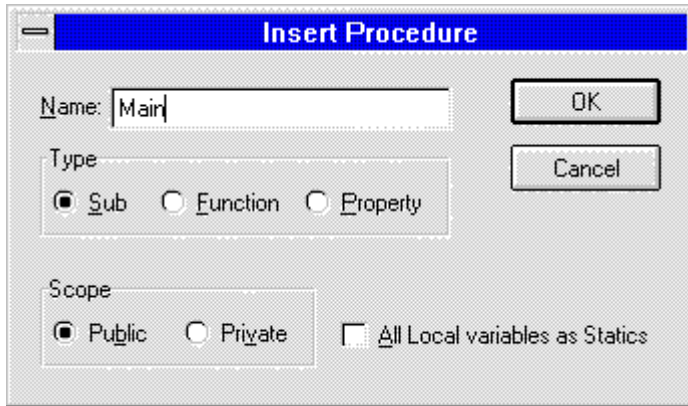


Figure 2. Creating the Main subroutine

Creating a Class Module

The third step is to create a class module for the properties and methods.

1. From the Insert menu, choose Class Module.
2. Press F4 to display the Property Sheet.
3. Type WinINI as the Name property, set the Instancing property to "2 - Creatable MultiUse," and the Public property to "True" as in Figure 3.
4. Close the property sheet by pressing ALT+F4.

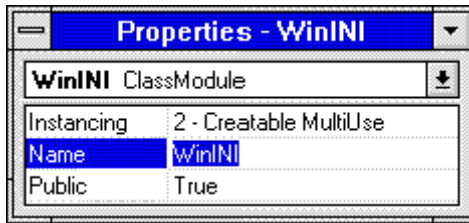


Figure 3. The property sheet of a class module

Creating the Properties and Methods

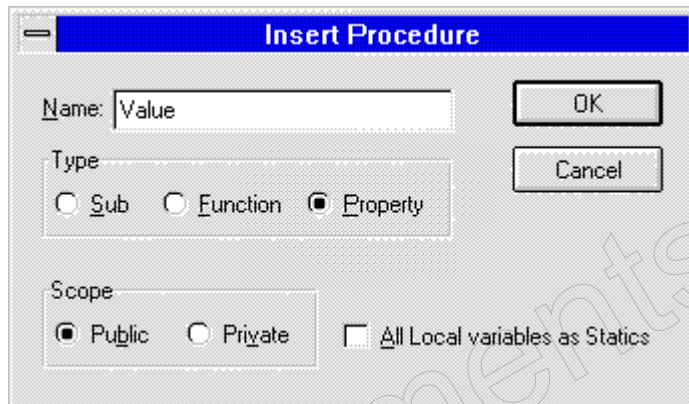


Figure 4. Creating a new property in a class module

The fourth step is to create the properties and methods of the OLE server. The following process is repeated as often as needed:

1. From the Insert menu, choose Procedure.

2. Type Value in the Name text box, click Property in the Type group, click Public in the Scope group, and then click OK. See Figure 4.

The following code is the result of this process:

```
Public Property Get Value()
End Property
Public Property Let Value(vNewValue)
End Property
```

Note A property can be on both the left and the right side of an equal sign (that is, you can both assign and read its values). A method is a procedure that can appear only on the right side of an equal sign.

The default procedures for a property are shown above. Properties can be Get or Let or both. For my example property, a string property, I will use both Get and Let. The next step is to put your code to call the API into the procedures and add appropriate arguments. The names of the arguments will appear in the Visual Basic for Application Object Browser, so I would recommend dropping Hungarian notation prefixes and carefully choosing the names of all arguments. Do not change the name of vNewValue—just add data type information. The code below is based on samples Q82158 and Q142388 in the Visual Basic Knowledge Base. (See the "Bibliography" section for the full reference.)

Note For readability on the CD, we have added returns at the end of long lines of code (for instance, the "Private" lines), which would not normally be the case. To make the code compile properly, you will have to remove the returns.—Ed.

```
Option Explicit
Private Declare Function GetProfileString Lib "Kernel" (ByVal lpAppName
  As String, ByVal lpKeyName As String, ByVal lpDefault As String, ByVal
  lpReturnedString As String, ByVal nSize As Integer) As Integer
Private Declare Function WriteProfileString Lib "Kernel" (ByVal
  lpApplicationName As String, ByVal lpKeyName As Any, ByVal lpString As Any)
  As Integer
Const BufferLen = 255 'Max Size of All INI Strings
Public Property Get Value(AppName As String, KeyName As String) As String
Dim Buffer As String * BufferLen, RC%, Default As String
MsgBox AppName + KeyName + Buffer, BufferLen
RC% = GetProfileString(AppName, KeyName, Default, Buffer, BufferLen)
Value = Left$(Buffer, RC%)
End Property
Public Property Let Value(AppName As String, KeyName As String, vNewValue$)
Dim RC%
RC% = WriteProfileString(AppName, KeyName, vNewValue$)
End Property
```

Running the Server

The next stage is to make our project into an executable. This is done in the usual way, but when you compile it, your executable is also registered as an OLE server automatically. If you run it by pressing F5, the project is temporarily registered as an OLE server. If you open the References dialog box in Microsoft Excel or some other Visual Basic for Applications product, you will see the OLE server twice as shown in Figure 5. One reference points to the executable and the other to the debug version.

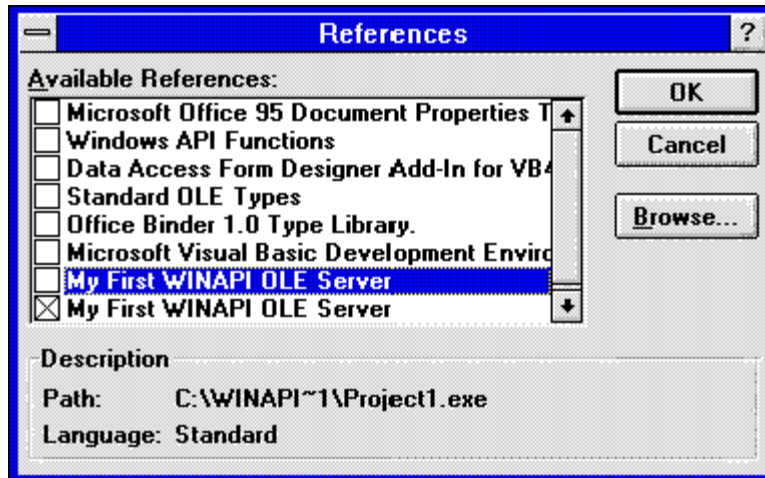


Figure 5. The References dialog box showing both compiled and debug OLE servers

The extension on the file indicates whether it is the debugging version (.VBP) or the compiled version (.EXE or .DLL). When I need to debug my code, I select the server with the .VBP extension. For more information on debugging, see the documentation for Visual Basic 4.0. Next I will show you how I use this OLE server.

Using a WinAPI OLE Server from Microsoft Excel and from Microsoft Access 2.0

Visual Basic 3.0 and Microsoft Access 2.0 support only ordered arguments for procedures, while later versions of these products and all products incorporating Visual Basic for Applications support both ordered arguments and named arguments. In this section I will illustrate the use of my OLE server in products that support named arguments—specifically 32-bit Microsoft Excel (version 5.0 or higher)—and then illustrate the equivalent use in products that support ordered arguments only (for example, 16-bit Microsoft Access, version 2.0 or lower.)

Using an OLE Server from Visual Basic for Applications

Visual Basic for Applications is available in Visual Basic 4.0 or higher, Microsoft Excel 5.0 or higher, Microsoft Project 4.0 or higher, and Microsoft Access 95 or higher. Visual Basic for Applications may be added to other products in future versions, so this is the language to master for corporate developers or solution providers. After I have selected my OLE server in the References dialog box (see Figure 5), I press F2 to bring up the Object Browser, which I can use to paste in the OLE server code. The code is very simple, as shown below in a Microsoft Excel module sheet. "MyWinAPI" in the declaration points to the project name I entered into the Options dialog box in Figure 1. "WinINI" in the declaration points to the Class Module name I entered into the module property sheet above in Figure 3.

```
Function WhereisMSINFO$()
Dim WINini As New MyWinAPI.WinINI
WhereisMSINFO$ = WINini.Value(AppName:="MSAPPS", KeyName:="msinfo")
End Function
```

I could dimension the WINini variable either as Object or as MyWinAPI.WinINI. I always do the latter by declaring the MyWinAPI.WinINI template as New because this results in early binding with very significant performance improvements. The Object used with CreateObject or GetObject results in late binding and is considerably slower. See the OLE references in the "Bibliography" section for further information about early and late binding of OLE servers.

Using an OLE Server from Microsoft Access 2.0 or Lower

Two problems are associated with using an OLE server in Microsoft Access 2.0 or lower and in Visual Basic 3.0 or lower: poor performance and the absence of named properties (and the associated Object Browser). The code must

always use Object and thus does late binding. My code above becomes:

```
Function WhereisMSINFO$ ()
Dim WINini As Object
Set WINini = CreateObject("MyWinAPI.WinINI")
WhereisMSINFO$ = WINini.Value("MSAPPS", "msinfo")
End Function
```

The code is longer and is more obtuse because of the absence of named properties.

OLE Server or Visual Basic Class Module

The creation of OLE servers to handle all of your API calls cuts development time, simplifies your code, and adds reusability of compiled code across multiple products. If you are working strictly in 32-bit mode, you may wish to create a version of the OLE server as an in-process server by compiling it as a dynamic-link library (.DLL) instead of an executable (.EXE). An in-process OLE server performs significantly faster than an out-of-process server and almost as fast as an internal procedure.

If you are working in the Visual Basic 4.0 environment, you have another option: You could include the class modules in your project instead of referring to an OLE server. This results in slightly better performance, but it loses many benefits available from an OLE server. The last question is always, "What is the performance penalty?" If moving API calls to procedures from in-line code is acceptable to you, using an in-process OLE server will be very acceptable—the percentage increase is far less.

"Rolling Your Own" OLE Server

That's it! The simplicity of this solution and the ease of implementation seems anticlimactic. An interesting aspect of this solution is that it has been available for years—the undiscovered OLE server.

I can now create additional properties and methods for my own server. Actually, I would create three server projects that use the same class modules because there are differences between the API calls in 16-bit Windows and 32-bit Windows (the registry functions, for example). Also, some API calls should be done as in-process OLE server calls—for example, GetCurrentProcess and GetPriorityClass. The best performance is always with an in-process OLE server with early binding (a 32-bit DLL declared using the New keyword). The in-process OLE server will perform almost as fast as a class module or a subroutine in your program. Table 1 summarizes my suggested design.

Table 1. A group of WinAPI OLE Servers for Visual Basic for Applications

Name (note extensions)	OLE Server Type	16-bit/32-bit	Server Name (see Figure 1)
APIOLE16.EXE	Out-of-process	16-bit	APIOLE16
APIOLE32.EXE	Out-of-process	32-bit	APIOLE32
APIOLE32.DLL	In-process (not accessible from 16-bit)	32-bit	APIOLE32P

The actual classes you choose to implement are what you need to examine next: Which API calls go into which class and which server, and what names are you going to give to them? (Keep in mind that a blind copying of the SDK cannot be done for properties.) My journey in this article is finished; your journey has just started.

Bibliography

Brockschmidt, Kraig. Inside OLE 2: The Fast Track to Building Powerful Object-Oriented Applications with Windows Objects. Redmond, WA: Microsoft Press, 1994. (MSDN Library, Books)

"How to Apply OLE 2 Technologies in Applications." (MSDN Library Archive, Backgrounders)

Knowledge Base Q82158. "How to Set Windows System Colors Using API and Visual Basic." (MSDN Library, Knowledge Base)

Knowledge Base Q142388. "Changing WIN.INI Printer Settings from VB using Windows API." (MSDN Library,

Knowledge Base)

Lassenen, Ken. "Corporate Developer's Guide to Office 95 API Issues." (MSDN Library, Technical Articles, Applications)

Lassenen, Ken. "Issues to Consider When Porting 16-bit Office Solutions to Windows 95." Developer Network News 4 (May 1995). (MSDN Library, Periodicals, Microsoft Developer Network News)

Lassenen, Ken. "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office." (MSDN Library, Technical Articles, Applications)

"Microsoft OLE Today and Tomorrow: Technology Overview." (MSDN Library Archive, Backgrounders and White Papers, Operating System Extensions)

"Object Linking and Embedding 2.0 Backgrounder." (MSDN Library Archive, Backgrounders and White Papers, Operating System Extensions)

[Manage Your Profile](#) | [Legal](#) | [Contact Us](#) | [MSDN Flash Newsletter](#)

© 2007 Microsoft Corporation. All rights reserved. [Terms of Use](#) | [Trademarks](#) | [Privacy Statement](#)



Requirements 2004/11/18