Office Development (General) Technical xArticles

Corporate Developer's Guide to Office 95 API Issues

Ken Lassesen
Microsoft Developer Network Technology Group

April 28, 1995

Abstract

This technical article examines how to write Microsoft® Office solution code that will successfully make application programming interface (API) calls using both 16-bit and 32-bit versions of Office products. In particular, this article applies to Microsoft Access®, Visual Basic®, Microsoft Word, Microsoft Excel, and Microsoft Project. There are three ways to make such API calls: using **REGISTER**, using **Declare** statements, and using a type library. This article examines each way of making an API call, accompanied by examples. The reader is assumed to be familiar with my earlier technical article, "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office," and is assumed to be an experienced Office developer who needs to port or write solutions to meet the new 16-bit to 32-bit interoperability requirement.

Introduction

The challenge of writing a solution for several generations of Microsoft® Office products across several platforms is unfamiliar to most compiled-language developers. Compiled-language application developers enjoy the luxury of picking a time slice of a compiler's evolution, encapsulating the solution into an executable file, and rarely having to deal with issues caused by the evolution of a product. An Office solution should have one solution file (an .XLS, .MDB, .MPP, or .DOT file) that will execute correctly with all Office products and on all operating systems. Office developers must write solutions that are backward-compatible with versions released five years ago and forward-compatible with the next versions of Office. For example, an expense report that runs on Microsoft Excel 4.0 and Microsoft Excel 5.0 should be able to run on Microsoft Excel NT, Microsoft Excel 95, and Microsoft Excel 2001 (assuming there is one).

Office 95 introduces 32-bit Office products into the mainstream corporate environment, where they will coexist with 16-bit Office products for many years. My earlier technical article, "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office," describes the problems of not being able to call 16-bit APIs from a 32-bit application and not being able to call 32-bit APIs from a 16-bit application. The 16-bit/32-bit interoperability solution presented in that article is examined in depth in this article.

> **Note** A rule of thumb that will convert many API calls in Windows® version 3.1 to Win32® application programming interface (API) calls is what I call the Rule of 32A: "Add a *32* to the DLL name and an *A* (for ANSI) to the function name if a string is used in any of the call's parameters." This rule of thumb will work for the majority of API calls, but is not a true solution because it will fail to work for some functions.

The 16-bit/32-bit interoperability solution requires more than porting a 16-bit API call to its equivalent 32-bit API call (in other words, using the 32A Rule); it requires that the solution select the appropriate API call when the solution is executed in a 16-bit or a 32-bit Office product.

Terminology

To reduce confusion over the word *application*, the following table defines the terms used in this technical article.

Term	Definition
Office product	A product that does not compile code to an executable. One of the

	following products: Microsoft Project, Microsoft Access, Word for Windows, Microsoft Excel.
Compiled language	A product that compiles code to create an executable, for example, Visual Basic®, FORTRAN PowerStation™, Visual FoxPro®, and Visual C++™.
Solution	An application developed by a third party or a developer using an Office product and in its file format, such as Microsoft Excel (.XLS), Word (.DOT), Microsoft Project (.MPP), or Microsoft Access (.MDB).
Solution code	The code (XLM or Basic or both) in which the solution is written.
Platforms	Microsoft® Windows and Apple® Macintosh® operating systems. Some Office products exist for all versions of Windows and Macintosh operating systems. Microsoft Excel can execute the same applications (with some restrictions) on both a Macintosh and a Windows PC by copying the solution to the other operating system.

API-Calling Methods

Solution code can be written in one of three ways—using Microsoft Excel macros (XLM), Basic (Access Basic, Visual Basic, Visual Basic for Applications, and WordBasic), or a combination of XLM and Basic. XLM uses the **REGISTER** and the **CALL** commands to make API calls while Basic uses a **Declare** statement, a type library, or both to make API calls. A Microsoft Excel solution that has been passed from developer to developer may use all these ways of making API calls. All three ways will work in the latest version of Microsoft Excel.

Table 1. API-Calling Methods Described

Method	Language	Technique for Making API Calls
REGISTER	XLM macros	Uses the **REGISTER** and **CALL** functions from macros .
Declare	Access Basic, Visual Basic, Visual Basic for Applications, WordBasic	Uses a **Declare** statement.
Type library	Visual Basic for Applications only	Select type library method from the Object Browser.

These methods are not product version-specific but are dependent on developers' code-writing preferences or the range of products on which the solution must run. For example, a developer may write an solution in XLM for Microsoft Excel 95 because the solution must also work with Microsoft Excel 4.0. Table 2 shows which API-calling methods work in which Microsoft products.

Table 2. API-Calling Methods for Microsoft Products

Product	Version	REGISTER Method	Declare Method	Type Library Method
Microsoft Excel	3.0, 4.0	X		
Microsoft Excel	5.0, 5.0 NT, 95	X	X	X
Word	2.0, 6.0, 95		X	
Visual Basic	1.0, 2.0, 3.0		X	
Visual Basic	4.0		X	X

Project	4.0, 95		X	
Microsoft Access	1.0, 1.1, 2.0		X	
Microsoft Access	95		X	X
FoxPro	2.5		X	
FoxPro	3.0		X	X

As we look at each method of making 16-bit and 32-bit API calls, we will examine the same simple API call: **GetTickCount**.

REGISTER Method

API calls using the **REGISTER** method are unfamiliar to developers who started writing Office solutions after they started to code in Visual Basic. The older generations of Microsoft Excel developers make API calls from XLM and avoid using Visual Basic for Applications. XLM should not be viewed as old technology—even though it has not been changed since Microsoft Excel 4.0. It may be the best choice if performance is a critical issue. The Microsoft Excel 5.0 Developer's Kit states one advantage of making an API call directly from XLM:

> *Because the C API is optimized for use from the Microsoft Excel macro language and the worksheet, it is not a very good mechanism for writing external functions to be used by Visual Basic (although Visual Basic and the C API can be combined into hybrid solutions).*

To make an API call from XLM requires the use of the **REGISTER** function to reference the dynamic-link library (DLL) and the **CALL** function to execute it. The **REGISTER** function performs the equivalent of a **Declare** statement in Visual Basic for Applications. The recommended **REGISTER** syntax is:

```
REGISTER(module_text,procedure,type_text,function_text,argument_text,macro_type,ca
```

To return to our sample API call, we can implement **GetTickCount** in a few lines of XLM code.

```
TestGetTickCount16
=REGISTER("User","GetTickCount","J","GetTickCount16",,1,9)
=SELECT("R1C1")
=FORMULA("GetTickCount - 16 bits")
=SELECT("R1C2")
=FORMULA("=Macro1!GetTickCount16()")
=RETURN()
```

This code produces the following output in 16-bit Microsoft Excel:

GetTickCount - 16 bits	7055256

The code fails if we run it with 32-bit Microsoft Excel. The API call returns #VALUE!, indicating a failure to pass values to or from the DLL:

GetTickCount - 16 bits	#VALUE!

We must convert it to a 32-bit API call, so we use the 32A Rule of adding a *32* to the DLL name and an *A* if it's a string. Code that implements the 32A Rule is shown below.

```
TestGetTickCount32A
=REGISTER("User32","GetTickCount","J","GetTickCount32A",,1,9)
=SELECT("R2C1")
=FORMULA("GetTickCount - 32A Rule")
=SELECT("R2C2")
=FORMULA("=Macro1!GetTickCount32A()")
=RETURN()
```

When we call **TestGetTickCount32A**, we receive a different error. The #NAME? value is returned, meaning the function does not exist in the DLL. It's either the wrong DLL or the wrong function name. (Remember, function names are case-sensitive.) This must be one of the exceptions to the 32A Rule. (I confess, I picked this function on purpose.)

GetTickCount - 32A Rule	#NAME?

In the MSDN Library, look up **GetTickCount** in the Platform SDK using the Keyword Index. The Quick Info jump at the top of the topic tells you the function is in the KERNEL32 library. (It was in the USER library in Windows 3.*x*.) The code below shows the corrected macro for 32-bit Microsoft Excel.

```
TestGetTickCount32
=REGISTER("Kernel32","GetTickCount","J","GetTickCount32",,1,9)
=SELECT("R3C1")
=FORMULA("GetTickCount - 32 bit")
=SELECT("R3C2")
=FORMULA("=Macro1!GetTickCount32()")
=RETURN()
```

When this code is run with 32-bit Microsoft Excel, we obtain the correct results:

GetTickCount - 32 bit	7057286

To illustrate the way API calls behave with 16-bit and 32-bit Microsoft Excel, examine Table 3, which shows the results of these macros. The macro that uses the 32A Rule ("32A bits") produces the same result in both 16-bit and 32-bit solutions. This illustrates the ability of the **REGISTER** function to locate (or fail to locate) the function name in both 16-bit and 32-bit DLLs from either 16-bit and 32-bit solutions. The return of #VALUE! indicates problems in passing or receiving parameters.

Table 3. Results of Making Calls Across 16-Bit and 32-Bit Layers from Microsoft Excel

Microsoft Excel 4.0 (16-bit version)		Microsoft Excel 95 (32-bit version)	
GetTickCount - 16-bit	9285676	GetTickCount - 16-bit	#VALUE!
GetTickCount - 32A Rule	#NAME?	GetTickCount - 32A Rule	#NAME?
GetTickCount - 32-bit	#VALUE!	GetTickCount - 32-bit	9463648

We can write macros that make 16-bit API calls *or* 32-bit API calls—but we *must* write macros that can make 16-bit API calls *and* 32-bit API calls, depending on the version of the Office product. Solution code must run on *both* 16-bit Microsoft Excel and 32-bit Microsoft Excel. The Visual Basic for Applications function **Engine32**, described in my earlier article ("Porting Your 16-Bit Office-Based Solutions to 32-Bit Office") does not work with Microsoft Excel 4.0. (Microsoft Excel 4.0 does not include Visual Basic for Applications.) **Engine32** *does* work for Microsoft Excel 5.0 or higher. A Microsoft Excel 4.0-compatible **Engine32** function must use macro code.

Engine32 Function for XLM

For XLM, the solution is similar to the Visual Basic for Applications solution. The function **Info ("osversion")** will contain *32* if Microsoft Excel is a 32-bit version. The **Engine32** macro shown below returns TRUE if Microsoft Excel is a 32-bit version, FALSE if Microsoft Excel is a 16-bit version.

	D
1	Engine32
2	=RESULT(4)
3	=INFO("OsVersion")
4	=FIND("32",D3)
5	=ISERR(D4)
6	=RETURN(D5)

Sample REGISTER-method solution

With **Engine32** defined, the macro code for **GetTickCount** is simple:

	A
1	GetTickCount
2	=RESULT(1)
3	=REGISTER("User","GetTickCount","J","GetTickCount16",,1,9)
4	=REGISTER("Kernel32","GetTickCount","J","GetTickCount32",,1,9)
5	=IF(Engine32(),GetTickCount32(),GetTickCount16())
6	=RETURN(A5)

Engine32 will work correctly with all versions of Microsoft Excel. The **GetTickCount** function performs the same as the API call did in Windows 3.1.

Note If you are concerned about performance, you should register all the API calls when you load the solution.

Steps of the REGISTER-method solution

If you are converting Microsoft Excel solutions to run on both 16-bit and 32-bit products, I suggest the following steps:

1. Create a new macro sheet called **APICALLS**.
2. Create the **Engine32** function in the **APICALLS** macro sheet**.**
3. Locate all the **REGISTER** functions in the solution and move them to **APICALLS**, one per column.
4. Using the macro code above in the sample **REGISTER** solution as a template, create functions for each API on your macro sheets.
   - Add the needed **ARGUMENT** lines.
   - Add the needed **RESULT** lines.
   - Add the 16-bit API **REGISTER** line, appending *16* to the function text.
   - Add the 32-bit API **REGISTER** line, appending *32* to the function text.
   - Do any needed data manipulation to make the API call.
   - Add an **If** line to call the appropriate API.
   - **RETURN** the return value (if any).
5. Test the function.
6. Define the function to Microsoft Excel.

This process allows existing API calls in macro sheets to be left untouched. Once you have created the macro sheet **APICALLS** (and tested it), you may import it into other solutions and reuse the macros, thus cutting conversion time. The content of this macro sheet is solution-independent (it contains only Windows API calls) and reusable in other solutions. This macro sheet **APICALLS** becomes a Rosetta stone for future 16-bit/32-bit solution development.

Declare Method

The introduction of Visual Basic for Applications gives Microsoft Excel developers an alternative to XLM. Visual Basic, WordBasic, and Microsoft Access developers are able to also write Microsoft Excel and Microsoft Project solutions with little difficulty. Basic code can be exchanged among a wide variety of products. The younger generation of Office developers will write code in Visual Basic for Applications and rarely use XLM.

To call an API from Visual Basic for Applications requires an API **Declare** statement. A declared API is callable anywhere in Basic code (or in a macro). There are two methods of doing **Declare** statements, as shown below.

Word Declare statements

```
'Word was first in doing Declare statements.
'Its format was frozen for backward compatability.
Declare Sub SubName Lib LibName$ [(ArgumentList)] [ Alias Routine$]
Declare Function FunctionName[$] Lib LibName$ [( ArgumentList)] [Alias Routine$] A
```

Visual Basic for Applications, Basic, and Microsoft Access Declare statements

```
Declare Sub globalname Lib "libname" [Alias "aliasname" ][([ argumentlist])]
Declare Function globalname Lib libname [Alias aliasname ] [([ argumentlist])] [As
```

Because I have already discussed some of the issues in "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office" and derived a methodology above, I will cut to a sample function and then show the steps of the **Declare** solution.

Basic Engine32 function

The **Engine32** function returns **True** if 32-bit API calls will work (and 16-bit API calls will fail), and it returns **False** if 16-bit API calls will work (and 32-bit API calls will fail). The **Engine32** functions given in "Porting Your 16-Bit Office-Based Solutions to 32-Bit Office" were designed to show how they differ in each Office product. The versions given below are modified for better performance and to work with more versions.

Performance was improved by initializing static variables on the first function call. All subsequent calls use the static variables instead of repeating additional function calls. Another method for improving performance is to initialize a **Global** or **Public** variable; however, this has the drawback that if a **Reset** occurs in some products, all subsequent API calls may fail until the **Global** variable is reinitialized.

Microsoft Excel 5 or higher and Project 4 or higher

The **Application**.**OperatingSystem** property in Microsoft Excel and Microsoft Project always contains *32* if the product is a 32-bit product. The product version number is not sufficient because Microsoft Excel 5.0 is available as both a 16-bit and a 32-bit product.

```
Function Engine32%()
Static sEngine32%,SEval%  'Statics are used to improve performance.
If SEval% Then Engine32%=sEngine32%: Exit Function
If instr(Application.OperatingSystem,"32") then sEngine32%=True
Seval%=True
Engine32%=sEngine32%
End Function
```

Microsoft Access 1.1 or higher

Microsoft Access has no product version that is both 16-bit and 32-bit. Checking the version number by calling **SysCmd** determines which version of Microsoft Access is being used. Microsoft Access 1.1 does not have a version number constant built in, so we always use 7 to ensure that the code will work with Microsoft Access 1.1. This method should be used to see whether your solution code should make a 16-bit or 32-bit API call.

```
Function Engine32% ()
Static sEngine32%,SEval%
If SEval% Then Engine32%=sEngine32%: Exit Function
If SysCmd(7) > 2 Then sEngine32% = True
```

```
Seval%=True
End Function
```

Word for Windows 2.0 or higher

Word must evaluate the **Engine32** function each time because Word does not support static variables. First we check to see if the product version number is high enough to indicate it may be a 32-bit version, and then we check the version of the operating system to see whether it is a 32-bit one. This two-stage process is needed because **GetSystemInfo** is not available on versions before Word 6.0, and Word 6.0 is available in both 16-bit and 32-bit versions.

```
Function Engine32
Engine32 = 0
If  Val(AppInfo$(2)) > 5 Then
   OS$ = GetSystemInfo$(23)
   If Val(OS$) > 6.3 Or Len(OS$) = 0 Then Engine32 = - 1
End If
End Function
```

Visual Basic

Although Visual Basic does not use solution code, Basic code is often exchanged with the products covered above. Visual Basic 4.0 does not have the **Application.OperatingSystem** property (this is not technically a part of Visual Basic for Applications), but uses conditional compilation with #IF and #ELSE. If you intend to share your code with other Microsoft products, you should create the following function (and not use conditional compilation elsewhere):

```
Function Engine32%()
'This is for VB4 only.
#IF WIN16
   Engine32% = False
#ELSE
   Engine32% = True
#ENDIF
End Function
```

For earlier versions of Visual Basic, use the following:

```
Function Engine32%()
'This is for VB1 - VB3; #IF is not supported.
Engine32% = False
End Function
```

Sample Declare-method solution

The following code demonstrates the **Declare**-method solution, except for Word:

```
Declare Function GetTickCount32 Lib "Kernel32" Alias "GetTickCount" () As Long
Declare Function GetTickCount16 Lib "USER" Alias "GetTickCount" () As Long

Function GetTickCount() As Long
If Engine32() Then
    GetTickCount = GetTickCount32()
Else
    GetTickCount = GetTickCount16()
End If
End Function
```

The **Engine32** function is used to determine which API call to make. The **Declare** statements indicate the actual API function name as an **Alias** to avoid accidental changes of case (32-bit API calls are case-sensitive) and then indicate the bitness of the API function by adding *16* or *32* to the end of the function.

This code can be copied and pasted in all Office products except Word. WordBasic existed before the Basic used in the other Office products and is *different*. We will examine the Word

solution later.

Steps of the Declare-method solution

If you are converting solutions to run on 16-bit and 32-bit products, I suggest the following steps:

1. Create a new module called **APICalls**.
2. Create the **Engine32** function in **APICalls.**
3. Locate all the **Declare** functions in the solution and move them to **APICalls**.
4. Using the code in the "Sample Declare-method solution" above as a template, create functions for each API.
    a. The arguments should match the Windows version 3.1 API calls.
    b. The results should be Win32 API call results. (Visual Basic will automatically convert to Windows version 3.1 if needed.)
    c. Add the 16-bit API Declare line in the Declarations; append 16 to the function name .
    d. Add the 32-bit API Declare line in the Declarations; append 32 to the function name.
    e. Do any needed data manipulation.
    f. Add an If line to call the appropriate API.
    g. Return the return value (if any).
5. Test the function.

This process allows existing calls in other modules to be left untouched. Once the developer defines and tests the **APICalls** module, she or he may import and reuse the module in other solutions, thus cutting conversion time. The content of this module is solution-independent and the developer may use the module again. This module, **APICalls**, becomes a component for future 16-bit/32-bit solution development in Visual Basic for Applications.

API-wrapper alternative solution

I usually put a wrapper around API calls instead of exposing the API call in the code. For example, I will wrap code around **GetProfileString** to create a function called **vbGetWinIni** that takes the same arguments but returns the string instead.

If you code in this style, you may wish to modify the API-wrapper function to call the appropriate API instead of creating additional functions.

Word sample Declare-method solution

Word has a different **Declare** format and syntax. The Word solution is more complex because you cannot place both the 16-bit and 32-bit **Declare** statements in the same macro. The solution is to create three macro libraries: **APICALL16** and **APICALL32**, that contain the **Declare** statements for each operating environment, and a 16-bit/32-bit interoperability macro, **APICALLS**. This may sound very confusing, so let us go through it step by step.

First, we create a macro library called **APICALL16**. This macro contains all the 16-bit API **Declare** statements.

```
'This is APICALL16 -- all 16-bit Declare statements are placed here.
Declare Function GetTickCount16 Lib "USER" Alias "GetTickCount"() As Long
Function GetTickCount
GetTickCount = GetTickCount16
End Function
```

Second, we create a macro library called **APICALL32**. This macro contains all the 32-bit API **Declare** statements.

```
'This is APICALL32 -- all 32-bit Declare statements are placed here.
Declare Function GetTickCount32 Lib "KERNEL32"() Alias "GetTickCount" As Long
```

```
Function GetTickCount
GetTickCount = GetTickCount32
End Function
```

Third, we create a macro library called **APICALLS**. This macro contains **Engine32** and the procedures your solution code will call.

```
'This is APICALLS -- no Declare statements may be in this macro.
Function Engine32
Engine32 = 0
If  Val(AppInfo$(2)) > 5 Then
   OS$ = GetSystemInfo$(23)
   If Val(OS$) > 6.3 Or Len(OS$) = 0 Then Engine32 = - 1
End If
End Function

Function GetTickCount
If Engine32 Then
   GetTickCount = APICall32.GetTickCount
Else
   GetTickCount = APICall16.GetTickCount
End If
End Function
'Other API function calls are placed here.
```

You can now call this function from your solution code. You must preface your calls with **APICALLS**, for example:

```
Sub MAIN
MsgBox Str$(APICalls.GetTickCount)
End Sub
```

Steps of the Word Declare-method solution

If you are converting Word solutions to run on 16-bit and 32-bit products, I suggest the following steps:

1. Create a new module called **APICALLS**.
2. Create the **Engine32** function in **APICALLS.**
3. Create a new module called **APICALL16**.
4. Locate all the 16-bit **Declare** statements in the solution and move them to **APICALL16**.
5. Create a new module called **APICALL32**.
6. Create the equivalent 32-bit **Declare** statements and put them to **APICALL32**.
7. Using the template above, create functions for each API in each of the three macro libraries.
8. Add **APICALLS** before all calls to the API in your solution code.
9. Test each function.

This process allows existing calls in other modules to be left untouched. Once the developer defines and tests these macros, she or he may add them to the NORMAL.DOT template and reuse the macros in other solutions so as to cut conversion time.

Type Library Method

The type library method of making API calls is new to most developers. Bruce McKinney's forthcoming Microsoft Press® book, *Hard Core Visual Basic*, includes the Windows API Functions type library for 16-bit API calls (WIN16.TLB) and a matching type library for 32-bit API calls (WIN32.TLB). Once these type libraries are registered, the appropriate type library is loaded for the 16-bit or 32-bit version of the Office product.

A type library provides easy access to the API calls with the products Microsoft Excel 5.0 or higher, Microsoft Project 4.0 or higher, Visual Basic 4.0 or higher, and Microsoft Access 95 or higher. All the Windows API calls become native functions. Because this approach is new to many of my readers, I will go through the steps in detail.

Registering the type library

The following steps will add a type library called Windows API Functions to your registry. This makes this type library available for *all* products that use Visual Basic for Applications, not only the product in which you registered it.

1. Open any Visual Basic for Applications product (for example, Microsoft Excel 5.0).
2. To create a module, choose Macro Module from the Insert menu.
3. Choose References from the Tools menu. The References dialog box is displayed as in Figure 1.

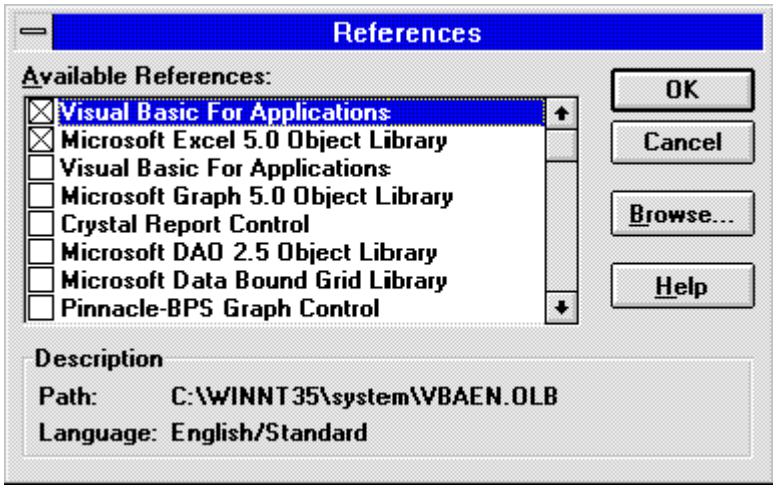

Figure 1. The References dialog box

4. Assuming that you have not registered the Windows API Functions type library, click the Browse button and locate WIN32.TLB, and click OK. Repeat with WIN16.TLB. Both WIN16.TLB and WIN32.TLB will be registered. See Figure 2.

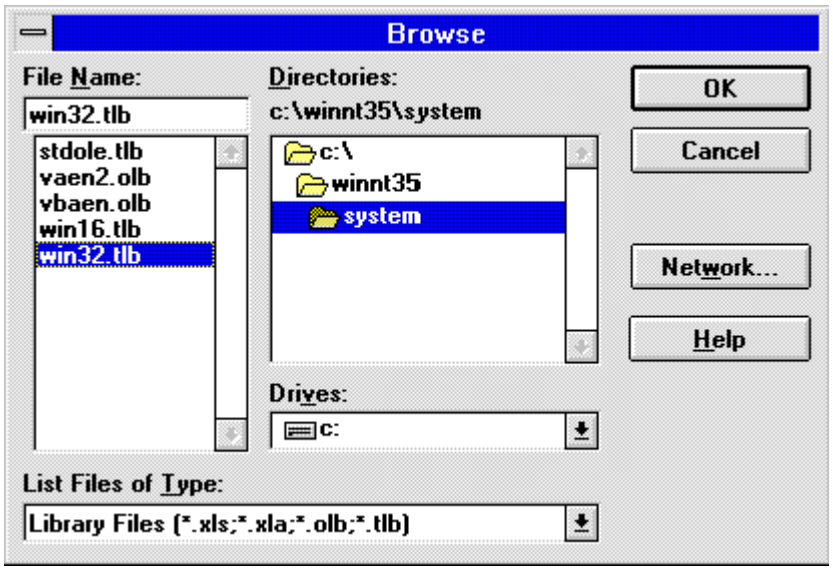

Figure 2. Selecting the type library

5. After the Browse dialog closes, scroll to the bottom of the References dialog box, and you

will see the Windows API Functions type library in the list (Figure 3).

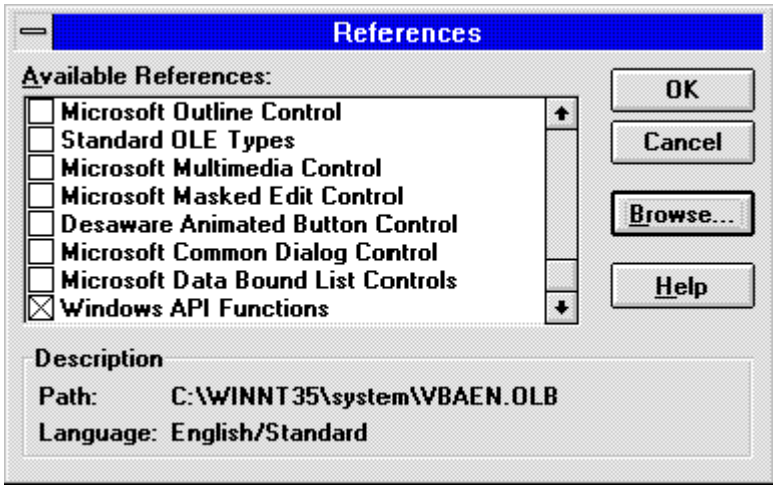

Figure 3. Registering Windows API Functions

6. Close the References dialog box.
7. From the View menu, choose Object Browser.
8. In the Object Browser dialog box, select Win (Windows API Functions type library) in the Libraries/Workbooks drop-down list box. All the available functions will appear in the Objects/Modules and Methods/Properties list boxes below (Figure 4).

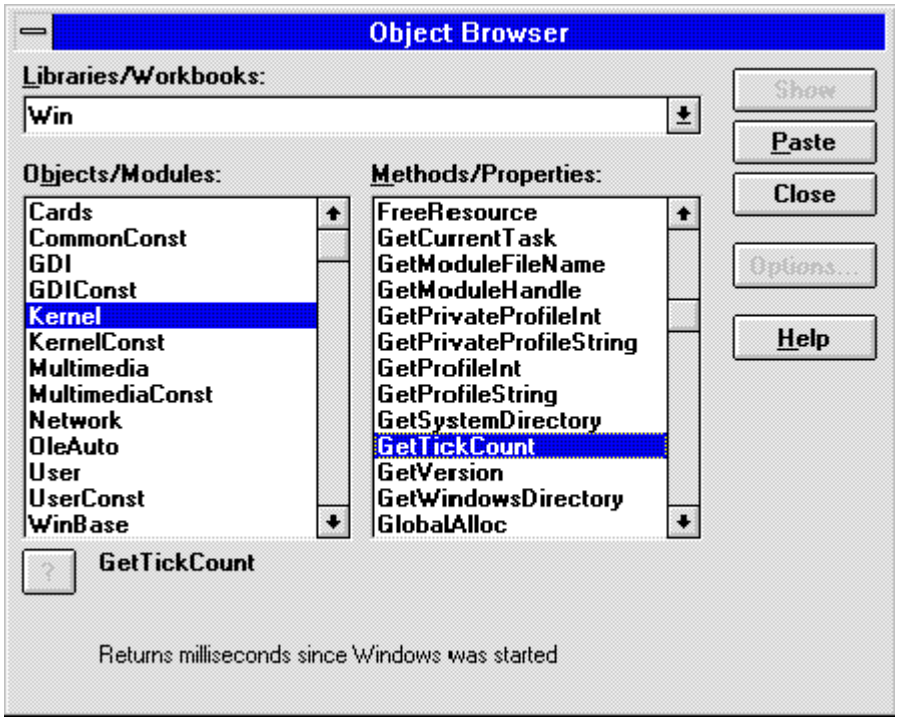

Figure 4. Microsoft Excel 5.0 Object Browser showing objects and methods available in Bruce McKinney's Windows API Functions type library

9. In the Objects/Modules list box, select Kernel; then in the Methods/Properties list box, select GetTickCount.
10. Click the Paste button. GetTickCount appears in the Module.

The following shows the code to display the value returned by the **GetTickCount** API call in a message box.

```
Sub Demo()
MsgBox Str$(GetTickCount)
End Sub
```

No **REGISTER** command nor **Declare** statements are needed. The code above is all the code you need.

Steps of the type library-method solution

If you are converting solutions to run on 16-bit and 32-bit products, I suggest the following steps—assuming the Windows API Functions type library is registered:

1. Remove all your **Declare** statements, assuming you are using only standard 16-bit API calls.
2. Check the Windows API Functions check box in the References dialog box.

That is all. You have done a complete porting.

Type library issues

Type libraries are a developing technology—there are very few type libraries commercially available. Bruce did an excellent job in designing the type library to work with both 16-bit and 32-bit products using the Windows 3.1 API call names. Nonetheless, type libraries have some issues that should be reviewed:

- The Windows API Functions type library adds more than a thousand new reserved words to Visual Basic for Applications. All the API calls contained in the type library become reserved words in the language. If you have an existing function called **ordShell**, you must change its name so that it does not conflict with the **ordShell** function already defined in the type library.
- The Windows API Functions type library does not include API calls that require a user-defined type (UDT). This may change in future versions.
- The Windows API Functions type library methods are not available on spreadsheets in Microsoft Excel.

This technology is very promising and will simplify the use of API calls in developing Office solutions.

Corporate development (â€œcorp devâ€) is responsible for executing mergers, acquisitions, divestitures, and capital raising in-house for a corporation. Professionals in this job role work alongside investment bankersInvestment Banking Career PathInvestment banking career guide - plan your IB career path. Learn about investment banking salaries, how to get hired, and what to do after a career in IB. The investment banking division (IBD) helps governments, corporations, and institutions raise capital and complete mergers and acquisitions (M&A). to identify acquisition targets and negotiate thei Specifically, what potential developments in customer demand, technology, or the regulatory environment could have enough impact on the industry to change the entire plan? How and why is this plan different from last yearâ€™s? What were your forecasts for market growth, sales, and profitability last year, two years ago, and three years ago?