

The introduction of 32-bit Microsoft® Windows® 95 (and Windows NT™) will bring 32-bit Office applications into common use and will result in users switching their Microsoft Office files, macros, and solutions to the 32-bit versions. This section will assist you in porting *solution code* — that is, code written in the Microsoft Excel macro language (XLM), WordBasic, Visual Basic for Applications, or Access Basic — to 32-bit versions of Office running on 32-bit operating systems.

Changes to your existing code *will be required* if — and *only if* — your 16-bit Office solution (including Microsoft Access or Microsoft Project) calls a 16-bit Windows application programming interface (API) or 16-bit Windows dynamic-link library (DLL), *and* you are porting that code to a 32-bit Office application (again, including Microsoft Access 95 or Microsoft Project 95).

Porting your solution code is necessary because 16-bit API calls and 16-bit DLL calls (referred to in this section simply as API calls) will not execute correctly when the solution code containing those calls is run in a 32-bit Office application. This section applies to solution code that uses APIs in the following products: Microsoft Excel, Microsoft Project, Microsoft Access, and Word for Windows.

Note

It is worth repeating that 16-bit solutions ported to 32-bit Windows are not effected. For example, existing 16-bit Office solutions, 16-bit Visual Basic, and 16-bit FoxPro applications, even if they call 16-bit APIs, will run just fine on Windows 95 or Windows NT. It is only when users want to run solutions code that includes 16-bit API calls on a 32-bit application that porting is required.

Abstract

Solution providers and corporate developers face an important task in ensuring that their Microsoft® Office-based solutions run successfully under Windows® 95 and Windows NT™.

The rule they must follow is this: neither a 32-bit compiled application nor solution code called from a 32-bit Office application can make direct 16-bit API or DLL calls. In addition, neither a 16-bit compiled application nor solution code called from a 16-bit application can make direct 32-bit API or DLL calls. This inability to make calls back and forth between 16-bit and 32-bit layers occurs in both the Windows 95 and Windows NT environments because of their advanced flat-memory-model management systems, as well as the way in which they load DLLs.

To prepare for Office 95, you must change your solution code to make Win32 API calls when the solution code is executed from 32-bit Office applications (including Microsoft Access 95 and Microsoft Project 95). If this is not possible (for example, you don't have access to the source code of the DLL), you must change the solution code to *thunk* through an intermediate DLL to make the 16-bit API call. The good news is that updating solution code to support Win32 API calls is a relatively simple mechanical process. A more significant task is to write code that is operating-system-independent (that is, so the solution code will run on both 16-bit and 32-bit Office applications). This document will discuss both of these tasks, as well as other 16-to-32-bit API issues you may need to handle.

Not e Although you must update API calls when porting solution code to 32-bit operating systems, you do not need to change code that uses OLE Automation or dynamic data exchange (DDE). All OLE and DDE code will continue to work regardless of whether the applications are 16-bit or 32-bit. OLE and DDE insulate automation calls, so all combinations of containers (clients) and servers (16/16, 16/32, 32/16, and 32/32) will work under Windows 95 and Windows NT.

How This Document Is Organized

What you need to know depends on your situation; therefore, this section is organized in terms of complexity, from the easier issues to the more complex ones.

1. “Which API Should Your Solution Code Call?” is a quick overview of which API you should be using, according to your application needs.
2. “Calling the Win32 API” describes what an API is and discusses the issues involved in converting existing 16-bit API calls to Win32 API calls, finding Declaration statements, and error codes.
3. “Writing a Single Code Base for 16-Bit and 32-Bit Office Applications” supplies code samples for writing solution code that will run on both a 16-bit and 32-bit Office application.
4. “Determining Whether a 32-Bit Application Is Running” describes how to determine whether your Office application is 16-bit or 32-bit and how to select the appropriate 16-bit or 32-bit API call.
5. “Recompiling DLLs” tells you what you will need to do to make the DLL and solution code work on Windows 95 and Windows NT if your solution code calls a custom DLL.
6. “Thunking” tells you how, if you cannot recompile your DLLs, you can add an intermediate DLL.
7. Advanced Programming Topics: Translating C API Declarations to Visual Basic or Visual Basic for Applications

Which API Should Your Solution Code Call?

When you write solution code for yourself, you write it for the version of the Office application you have and for your operating system. Distributing this solution to others means that you have to make it also work on their computers, which may use different versions of Windows and Office applications than you used when you wrote it. It happens that the operating system is a moot issue. What *is* important is whether the Office application is 16-bit or 32-bit. The following table shows that the application, *not* the operating system, determines which API you use in porting your solution code.

Microsoft Product	Windows versions 3.0, 3.1, 3.11	Win32s	Windows NT	Windows 95
16-bit applications	16-bit API	16-bit API	16-bit API	16-bit API
32-bit applications	N/A	32-bit API	32-bit API	32-bit API

Not e Microsoft Office (including 32-bit Access and 32-bit Project) products do not run on Win32s®, but since Microsoft FoxPro does, the Win32s column was added to show that FoxPro programmers should use the same rules for choosing the API. Also, Win32s, Windows NT, and Windows 95 do not have identical sets of API calls. For more information on this, you should consult the Win32 SDK documentation in the Development Library (in particular, see the Compatibility Tables in the *Win32 Programmer's Reference*, Vol. 5).

Calling the Win32 API

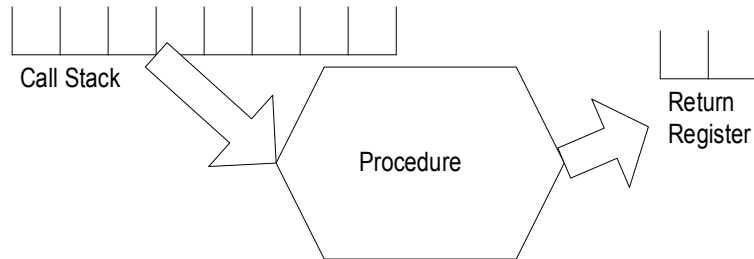
The path to 32-bit Windows API enlightenment consists of four steps.

1. Understanding what Windows API calls are.
2. Understanding the differences between 16-bit and 32-bit Windows APIs.
3. Using WIN32API.TXT to find the correct **Declare** statement.
4. Testing an API **Declare** statement.

What Is an API Call?

An API call in C, Visual Basic, or other languages places a series of values (parameters) at a location in memory (the stack) and then requests the operating system or DLL to execute a function (the procedure call) using the values provided. The function reads the values (call stack) and executes its function code using those values or the data that the values point to. If a result is returned, it is placed at another location (return register) for the calling application to use. This is shown in the following illustration. To ensure accuracy, the number of bytes of

data on the stack is verified before and after the procedure is called. The message “Bad DLL calling convention” appears when the wrong number of bytes are on the stack.



In practical terms, Windows API calls are how applications request services (screen control, printers, memory) from the operating system. There are approximately 300 API calls in Windows 3.0, over 700 API calls in Windows 3.1, and over 1,000 API calls in Windows 95. These API calls are packaged in executables and DLLs found in the Windows directory — USER.EXE, GDI.EXE, and one of the following KERNEL files: KRNL286.EXE, KRNL386.EXE, or KERNEL32.DLL.

To call an API from your solution code, use these four steps:

1. Identify the file containing the API.
2. Determine the parameters required by the API.
3. Create a **Declare** statement for the API.
4. Call the function with valid parameters.

The following is a simple example for the **GetVersion** API call that obtains the version of Windows that is running. The **GetVersion** API call is located in KERNEL under 16-bit Windows and does not use any parameters (so the **Declare** statement has empty parentheses). The following **Declare** statement is written for 16-bit Windows for use by Visual Basic for Applications:

```
Declare Function GetVersion Lib "KERNEL" () As Long
```

By comparison, here is the same function as it would be used by an Office 95 application running on 32-bit Windows:

```
Declare Function GetVersion Lib "KERNEL32" () As Long
```

Although the Windows API name stays the same, note that the location of the API has changed to KERNEL32. After all, you are calling from a 32-bit application, so you must make a 32-bit API call. The parameter data type, on the other hand, did not change (it remained a **Long**). In general, the function parameters will change more and require more attention than the parameters of

the return value. Understanding the differences between 16-bit API calls and 32-bit API calls is essential to porting your solution code to Windows 95.

What Are the Differences Between a 16-Bit API and a 32-Bit Windows API?

As shown above, most 32-bit Windows API calls have the same name or a very similar name to the 16-bit API calls. In fact, the documentation may show the same arguments, with the only apparent difference being the library name change from KERNEL to KERNEL32 as shown above. However, the code must handle changes in addition to the name change:

- Case-sensitivity
- Unicode or ANSI options
- Change of parameter data type (shown above)

These bulleted items can require subtle changes in the **Declare** statements that are not always easy to identify.

Case sensitivity

The first issue in moving to 32-bit Windows API calls is case sensitivity in the name of the function. API calls under 16-bit Windows would work if you entered the function name as GetVERSION, GeTvErSiOn, or getversion. In other words, in 16-bit Windows the following statements are equivalent:

```
Declare Function GetVersion Lib "KERNEL" () As Long
Declare Function GeTvErSiOn Lib "KERNEL" () As Long
```

API calls under 32-bit Windows are case-sensitive for the function call and must be correctly entered in the **Declare** statement. In other words, the following statements are *not* equivalent in 32-bit Windows.

```
Declare Function GetVersion Lib "KERNEL32" () As Long
Declare Function GeTvErSiOn Lib "KERNEL32" () As Long
```

The easiest way to handle this change is *always* use the **Alias** control word. The contents of an **Alias** string map to the actual API call name (which is case sensitive), but the function name used in code, which appears between “Function” and “Lib,” is not case sensitive and will not change if you type it different ways in your code or use the same name for variables or procedures. Using the **Alias** control word, the GetVersion function (32-bit Windows) would be entered as:

```
Declare Function GetVersion Lib "KERNEL32" Alias "GetVersion" () As Long
```

Now you can forget about the case sensitivity of API names when writing code: as long as you spelled and typed the function name correctly in the **Alias** string and you spell the function name in code the same way as in the **Declare**

statement, the function will be mapped by Visual Basic or Visual Basic for Applications back to the correct **Declare** function automatically.

Not e The **Alias** control word is the single most important thing you can use in preparing to switch to 32-bit operating systems because it means you will only have to change the contents of the **Declare** statement and not every instance of the function being called in your code.

Unicode or ANSI options

Both Windows NT and Windows 95 have two API interfaces. One interface is based on the American National Standards Institute (ANSI) character set, where a single byte represents each character. The other interface was created for the Unicode character set, where two bytes represent each character. All 16-bit Windows operating systems and applications use the ANSI character set. All 32-bit versions of Windows added Unicode to allow foreign language characters to be represented because some languages have many more characters than the 26 letters of English. C programmers handle this by setting a flag in their **include** file (*.H). The flag causes hundreds of macros throughout the C **include** files to select the correct Unicode or ANSI functions.

All western language versions of Office products use ANSI for Visual Basic for Applications code. Therefore, programmers using current versions of Visual Basic for Applications or macro languages will *always* use the ANSI version of the API call (when using the WIN32API.TXT file, documented later in this paper, this choice is made for you).

In case you are curious or need to know how to tell the ANSI version from the Unicode version, the ANSI version adds an *A* to the end of the API name, and the Unicode version adds a *W*. (*W* is for wide, as in the width of the bytes provided for characters). Note that the name of an API call adds the characters *A* and *W* at the end of the API name *if* — and *only if* — the API requires parameters with string (character) data types.

The Win32 SDK documentation in the Development Library does not record the permutations of the name of the API call. The documentation gives only the name of the root function and its library name. The actual name of the API in the library may be one of three possibilities:

- **MyAPICall**, which uses no character strings in the call.
- **MyAPICallA**, which uses ANSI character strings in the call.
- **MyAPICallW**, which uses Unicode character strings in the call.

A visual picture of the amount of data the API expects to find on the stack may help illustrate the differences. Possible call stacks for an example function are shown in the following illustration (the 16-bit version is padded because 16-bit Windows always pads the stack to 16 bits).

16-bit API Call	HWND	LPSTR	TCHAR		Integer
32-bit ANSI API Call	HWND	LPSTR	TCHAR	PADDING	Integer
32-bit Unicode API Call	HWND	LPSTR	TCHAR	PADDING	Integer

The three possible declarations for **MyAPICall** are shown below (formatted to make comparison easier). Note that all of the statements use the **Alias** control word so that the function name used in code (MyAPICall) does not have to change even if the name of the function called is appended with an “A” or “W”:

```
'16 bits
Declare function MyAPICall Lib "MYDLL.DLL" Alias "MyAPICall" (
    ByVal hwndForm As Integer,
    ByVal lpstrCaption$,
    ByVal hAccKey As String,
    ByVal iMagicNumber As Integer
) As Integer

'32-bit ANSI
Declare function MyAPICall Lib "MYDLL32.DLL" Alias "MyAPICallA" (
    ByVal hwndForm As Long,
    ByVal lpstrCaption$,
    ByVal hAccKey As String,
    ByVal iMagicNumber As Long
) As Long

'32-bit UNICODE * For illustration only.
Declare function MyAPICall Lib "MYDLL32.DLL" Alias "MyAPICallW" (
    ByVal hwndForm As Long,
    ByVal lpstrCaption$,
    ByVal hAccKey As String,
    ByVal iMagicNumber As Long
) As Long
```

Any one of these declarations would add the function **MyAPICall** to your application; you can only have one **MyAPICall** function.

Not This code sample introduces the **ByVal** keyword, which enables you to pass Visual Basic parameters to a API function "by value". "By Value" is the default for functions written in C and is therefore the default for Windows API calls. The reason you must use **ByVal** is Visual Basic and Visual Basic for applications default to **ByRef** ("By Reference" which passes a pointer to the value rather than the value itself) which is not what API calls expect. **ByVal** can also be used to convert a Visual Basic string to a C string (null terminated). **ByVal** is included in the Declare statements in WIN32API.TXT so you will know when to use it, but for more information on **ByVal**, consult the MSDN article Q110219 "How to call Windows API from VB," or published references such as "The Visual Basic Programmer's Guide to the Windows API" by Dan Appleman.

Change of parameter data type

The easiest way to learn what the new required parameter data types are for 32-bit API functions is to have somebody else give them to you. Therefore Microsoft is providing a Visual Basic declaration file, WIN32API.TXT, for your use with this document that will also ship in MSDN (July CD), Visual Basic 4.0, and in the Access Developer's Toolkit for Access 95. All you will need to do is copy the desired API **Declare** statement from WIN95API.TXT into your source code.

Another source of information is the *Win32 Programmer's Reference* (found on the MSDN Development Library CD), which is discussed in the Advanced Programming Topics section in this document. This reference may occasionally be required to resolve questions about the inclusion or exclusion of **ByVal** in the declaration or the need to put parentheses around the actual value passed.

If you use the *Win32 Programmer's Reference*, however, you must be careful to properly convert C to Visual Basic data types, for example don't mistake a C **int** for a Visual Basic for Applications **Integer**. Many Windows data types and Visual Basic **Integer** data types have ceased being the same size, as shown in the following table. It is critical to remember that the sizes of many API parameters have changed, and you must not assume they are the same.

Visual Basic Data Types	Size of Variable	16-Bit Windows Data Types	32-Bit Windows Data Types
Integer	2 bytes	int, short, WORD, HWND, HANDLE, WCHAR	short, WCHAR
Long	4 bytes	long, LPSTR	int, long, HANDLE, HWND, LPSTR

Finally, whichever resource you choose, judicious use of the **Alias** control word may assist you with changing parameter data types by allowing existing 16-bit

code (italicized to point out that “code” does not include the **Declare** statement, which must change to point to a 32-bit API) that calls the API to be left unchanged. This seems surprising but the **ByVal** control word and automatic type conversion in Visual Basic, Access Basic, WordBasic, and Visual Basic for Applications will change the size of parameters for you in many cases (**Integer** to **Long**, for example). Alternatively, type conversion will extend integers with a sign (+/-) which may lead to incorrect long parameters and cause overflows on conversion from **Long** to **Integer**. Again - the best solution is to check the references to get the correct functions.

What Types of Errors Can Occur with an API Declare Statement?

After you create a **Declare** statement, you may find that it fails to work. There are many possible mistakes one can make in a **Declare** statement. I will cover the common errors below.

Error 453: Function is not defined in specified DLL

Either you misspelled the function name or you have a problem with case in the function name. Remember that the functions are case-sensitive in Win32; they were not case-sensitive in 16-bit Windows.

Error 48: Error in loading DLL

Usually, this error is caused by having the wrong size or arguments, but may also occur for some of the reasons described under Error 53 below.

Error 53: File Not Found

This may occur for many reasons. Windows checks the loaded libraries for matches, and if the DLL is not loaded, it will attempt to load the DLL from disk. Many functions available in the 16-bit Windows on Windows (WOW) layer on a Windows NT system are not available directly from Windows NT. Calling the 16-bit Windows and Win32 **GetProfileString** function from a 16-bit and a 32-bit solution will give a confusing set of error messages. The 16-bit application call will find KERNEL and fail to find KERNEL32, while the 32-bit application will find KERNEL32 and fail to find KERNEL. The general cause of this error is a mismatch of calls and environment. The solution is to write code that works in both 16-bit and 32-bit environments.

Assuming you can now code the API call and call it successfully from solution code, you're ready to attack the next problem.

Writing a Single Code Base for 16-Bit and 32-Bit Office Applications

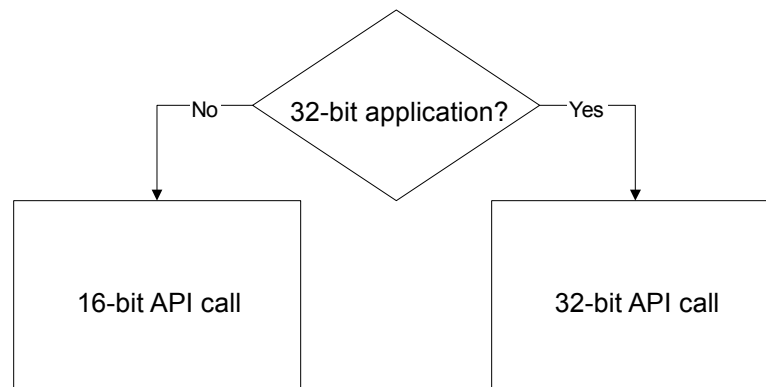
If your users are running both 16-bit and 32-bit versions of Microsoft Excel, should you put the 16-bit API call or the 32-bit API call in your solution?

Microsoft Excel is not like Visual Basic version 4.0, which allows conditional compilation of code into executables, but instead runs solution code in workbook files which may be opened in either the 16- or the 32-bit version. This is actually a trick question — the right answer is that you put *both* the 16-bit API call and the 32-bit API call in your solution inside an **If ...Then...Else** control structure.

With 32-bit applications using the same solution code as 16-bit applications, you do not know which API to call in the solution code. Your code must determine whether the application is a 16-bit one or a 32-bit one. How does the code answer the following questions?

- Microsoft Excel: Is the host application 16-bit Microsoft Excel version 5.0 (make 16-bit API calls) or 32-bit Microsoft Excel version 5.0 for Windows NT or 32-bit Microsoft Excel for Windows 95 (make 32-bit API calls)?
- Microsoft Access: Is the host application 16-bit Microsoft Access version 2.0 (make 16-bit API calls), or 32-bit Microsoft Access for Windows 95 (make 32-bit API calls)?
- Microsoft Project: Is the host application 16-bit Microsoft Project version 3.0 (make 16-bit API calls) or 32-bit Microsoft Project 95 (make 32-bit API calls)?
- Word for Windows: Is the host application Microsoft Word version 2.0 or 16-bit Word version 6.0 (make 16-bit API calls) or 32-bit Word version 6.0 for Windows NT or Word 95 (make 32-bit API calls)?

Remember, if you make the wrong API call (i.e., cross the 16 to 32 bit barrier), an error will occur. The solution code must determine whether the application is a 16-bit application or a 32-bit application, so it can make the appropriate call.



The solution is to put every API call into a *wrapper* — a Visual Basic procedure or a Microsoft Excel version 4.0 macro. This wrapper routine checks the “bitness” of the application and selects the appropriate API call. Place these wrappers in separate modules so that your code may be easily reused. Some API calls (for example, **GetPrinterDriveDirectory** and **GetWinMetaFileBits**) are not available in all 32-bit operating environments, which means that the structure of an API wrapper can become as complex as this:

```
Function MyAPICall$(ByVal Args)
  If Engine32() Then
    'Select is rarely needed
    Select Case OS32() 'Based on GetVersionEx API
      Case 0 'Win32s
        ....
      Case 1 'NT 3.1
        ....
      Case 2 'NT 3.5
        ....
      Case 3 'Windows 95
        ....
    End Select
  Else '16-bit
    ....
  End If
End Function
```

This complexity is the exception and not the rule.

32-bit Executables Are Easy

Compiled languages, such as FoxPro and Visual Basic, build 16-bit or 32-bit application executables. The executable targets either 16-bit API calls or 32-bit API calls. You can determine the appropriate API calls while building the application. You can select the calls either by having all the 16-bit declarations in one file and all the 32-bit declarations in another file and manually switch them in a project. The other option is using the **#IF... #ELSE...** directives and conditional compilation supported by Visual Basic version 4.0. If you must support Visual Basic 3.0 and Visual Basic 4.0 applications concurrently, separate files may reduce code maintenance. If you support FoxPro, you will have no problem using 16-bit API calls from compiled 32-bit FoxPro solutions because the **RegFN** functions will automatically thunk from the 32-bit layer to the 16-bit layer if needed.

Compiled 32-bit languages may require some minor differences in API calls depending on the 32-bit operating system. For example, developers must program context menus differently for Windows 95 than for Windows NT.

Determining Whether a 32-bit Application Is Running

In the previous section the ability to write application independent code was covered by adding code for both 16 and 32-bit scenarios. However, you still need to determine in source code whether the application is a 32-bit application or a 16-bit application without doing any API calls (you cannot do an API call because you do not know if a 16-bit API call or a 32-bit API call will work). The following code will determine if the application is a 32-bit application.

Not The **Application.OperatingSystem** property in Microsoft Excel and **e** Project does not return the version of Windows you have installed, but the layer of Windows that the application is running on, for example, the 16-bit subsystem in Windows NT.

Microsoft Excel 5.0, Microsoft Excel 95, Microsoft Project 4.0, Microsoft Project 95

```
Function Engine32%()  
    If Instr(Application.OperatingSystem,"32") then Engine32%=True  
End Function
```

Word 6.0, Word 95

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

Microsoft Access 1.1 or higher

```
Function Engine32%()  
If SysCmd(7) > 2 Then Engine32% = True  
End Function
```

Here is a simple example to help you understand some issues.

```
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 **GetTickCount** API has the same name for both 16-bit Windows and 32-bit Windows, so you *must* use an **Alias** control word to change the function name in at least one of the **Declare** statements (in the example above, the names in both **Declare** statements were changed - to **GetTickCount32** and **GetTickCount16**). Next, depending on the application's "bitness", **GetTickCount** is mapped to the correct API function name (**GetTickCount32** or **GetTickCount16**) and its associated API call. In this example, **GetTickCount** in your code will get mapped to **GetTickCount32** (in the **GetTickCount** function), which is mapped to **GetTickCount** in **KERNEL32**, when **Engine32%** is **True**.

Recompiling DLLs

This section has so far focused on the issue of updating Windows API calls — but the situation for solution code that calls 16-bit DLLs that you have bought, developed, or simply used is exactly the same. The developer must change all 16-bit DLL **Declare** calls in solution code to 32-bit calls. This requires creating a 32-bit version of the DLL (at least) and possibly changing the **Declare** statement (in Microsoft Excel version 4.0 macros, the **Register** function). This also means a separate DLL must exist for both the 16-bit application and the 32-bit application. For file management, the name of the 32-bit DLL should include "32" at the end. The developer must recompile the DLL as a 32-bit ANSI DLL. The parameters passed to the DLL *must* use the **stdcall**-passing protocol to talk to 32-bit Visual Basic for Applications (instead of the **PASCAL**-passing protocol used with 16-bit Windows). Remember to place the calls for the 16-bit and 32-bit versions of the DLL in a wrapper similar to the API wrapper above. For additional information on recompiling applications, consult Chapter 1, "Porting 16-Bit Code to 32-Bit Windows," in *Programming Techniques* from the Visual C++ 2.1 documentation in the Development Library, or consult your C compiler vendor's documentation.

Thunking

In some cases, you may not have the source code of a DLL. In this case, your solution is to use *thunking* to port your solution to Windows 95. Thunking enables direct 16-bit and 32-bit calls but requires much more work than simply changing the Windows API call. If you cannot change or recompile the 16-bit DLL, you must write a new 32-bit DLL wrapper to access the 16-bit DLL. The 32-bit application calls to this 32-bit wrapper DLL, which then talks to the original 16-bit DLL.

Thunking allows parameters to be pushed correctly on the stack, enables a DLL of a different bitness to load in your process, and converts memory addresses from **offset** (32-bit) to **segment::offset** (16-bit). This means, however, there are some challenges even if you do the thunking work. For example, pointers to pointers to memory locations require additional work in 16-bit and 32-bit scenarios.

Note that there are different ways to think depending on your operating system. Windows 95 and Windows NT think differently. For more information, see “Diving into the Requirements for the Windows 95 Logo” in the Development Library. This section gives an overview of thinking across the Windows platforms and contains pointers to more detailed information on thinking.

Advanced Programming Topics

Most developers writing solution code know the C language, and the following information is provided to assist them in using their knowledge of C to create Declare statements for Visual Basic and Visual Basic for Applications using the tools they already have.

Working from C declarations

Apart from the API location changing (from KERNEL to KERNEL32), the main issue in moving from 16-bit API calls to 32-bit API calls is the change in the size of parameter data types. Some background information may help you understand what has changed and why. Windows 3.0 was designed for the Intel 80286 CPU, where the hardware handles data 2 bytes at a time or in 16-bit words. Windows 95 was designed for later CPUs, where the hardware can handle data 4 bytes at a time or in 32-bit words. A look at how Visual Basic represents an **Integer** versus how Windows represents an **int** reveal these differences:

- **Integer** and **int** are each 2 bytes in the 16-bit Windows operating system and in 16-bit Microsoft Excel, Visual Basic, Microsoft Access, Word for Windows, and Microsoft Project.
- **Integer** is 2 bytes in 32-bit Microsoft Excel, Visual Basic, Microsoft Access, Word for Windows, and Microsoft Project, the same as in the 16-bit versions of these products.
- **int** is 4 bytes in the 32-bit Windows operating systems, Windows 95, and Windows NT.

To illustrate how this change of size can change a call, recall the fictional **MyAPICall** API used earlier. The **MyAPICall** call needs the handle to the application's window (HWND), a string, a character, and an integer to be placed on the stack. In C, the function would be:

```
int MyAPICall (HWND hwndForm, LPSTR lpstrCaption, TCHAR tchAccKey, int iMagicNumber)
```

Each parameter has two parts: the data type (**HWND**, **LPSTR**, **TCHAR**, **int**) and the field name (**hwndForm**, **lpstrCaption**, **tchAccKey**, **iMagicNumber**). Each data type requires a specific number of bytes to represent it. Each field name has some funny characters as a prefix — these characters (known as Hungarian notation) indicate the data type, such as **int** or **lpstr**.

Windows has many data types that API calls use as parameters. The following table shows some of the more significant data types used by Windows 95 API

calls. Many Windows data types use the C data type of **int**. When **int** changed from 16-bits to 32-bits, the related Windows data types also changed.

C Data Type	Windows 3.0, 3.1, & 3.11 and Windows for Workgroups 3.1 & 3.11 (16-Bit)	Win32s, Windows NT, and Windows 95 (32-Bit)
unsigned int, UINT, int	2 bytes	4 bytes
short	2 bytes	2 bytes
long	4 bytes	4 bytes
char, CHAR	1 byte	1 byte
WORD	2 bytes	2 bytes
Handle (hWnd, hDC, hMenu)	2 bytes	4 bytes
LPSTR	4 bytes	4 bytes
WCHAR	2 bytes	2 bytes
TCHAR (ANSI or Unicode)	1 byte	1 or 2 bytes
POINT	4 bytes	8 bytes

Thus, converting our **MyAPICall** API call from C, the declarations for **MyAPICall** using Visual Basic for Applications, Access Basic, or WordBasic would be as follows (formatted to make comparison easier):

'16 bits

```
Declare Function MyAPICall Lib "MYDLL.DLL" Alias "MyAPICall" (
    ByVal hwndForm As Integer,
    ByVal lpstrCaption As String,
    ByVal hAccKey As String,
    ByVal iMagicNumber As Integer
) As Integer
```

'32 bits

```
Declare Function MyAPICall Lib "MYDLL32.DLL" Alias "MyAPICall" (
    ByVal hwndForm As Long,
    ByVal lpstrCaption As String,
    ByVal hAccKey As String,
    ByVal iMagicNumber As Long
) As Long
```

A final tool you may find useful is the following table which maps C languages declaration data types to their Visual Basic equivalents.

C language declaration	Visual Basic equivalent	Call with
Boolean	ByVal B As Boolean	Any Integer or Variant variable
Pointer to a string (LPSTR)	By Val S As String	Any String or Variant variable

Pointer to an integer (LPINT)	I As Integer	Any Integer or Variant variable
Pointer to a long integer (LPDWORD)	L As Long	Any Long or Variant variable
Pointer to a structure (for example, LPRECT)	S As Rect	Any variable of that user-defined type
Integer (INT, UINT, WORD, BOOL)	ByVal I As Integer	Any Integer or Variant variable
Handle (32 bit, HWND)	ByVal H As Long	Any Long or Variant variable
Long (DWORD, LONG)	ByVal L As Long	Any Long or Variant variable
Pointer to an array of integers	I as Integer	The first element of the array, such as I(0)
Pointer to a void (void*)	V As Any	Any variable (use ByVal when passing a string)
Void (function return value)	Sub Procedure	n/a
NULL	As Any	ByVal 0&
Char (TCHAR)	ByVal Ch As String	Any String or Variant variable

Using the Win32 Programmer's Reference

As stated above, the two primary sources for Win32 API information are the *Win32 Programmer's Reference* and a list of Microsoft-supplied Win32 **Declare** statements for Visual Basic, such as WIN95API.TXT. The Development Library CD contains a listing with explanations of the entire Win32 API set in the *Win32 Programmer's Reference*, which is used in the example that follows.

As an example of using the *Win32 Programmer's Reference*, **GetProfileString**:

GetProfileString

[Quick Info](#)

The **GetProfileString** function retrieves the string associated with the specified key in the given section of the WIN.INI file. This function is provided for compatibility with 16-bit Windows-based applications. New applications should store initialization information in the registry.

```
DWORD GetProfileString(  
    LPCTSTR lpzSection,      // address of section name  
    LPCTSTR lpzKey,          // address of key name  
    LPCTSTR lpzDefault,      // address of default string  
    LPTSTR lpzReturnBuffer,  // address of destination buffer  
    DWORD cchReturnBuffer   // size of destination buffer  
);
```

By contrast, the declaration from the Windows 3.1 SDK *Programmer's Reference, Volume 2*, is:

```
int GetProfileString(  
    LPCSTR lpzSection,      // address of section  
    LPCSTR lpzEntry,        // address of entry  
    LPCSTR lpzDefault,      // address of default string  
    LPSTR lpzReturnBuffer,  // address of destination buffer  
    int cbReturnBuffer      // size of destination buffer  
);
```

Note that the Win32 SDK authors selected **DWORD** instead of the 32-bit **int** for the parameter data type for *chReturnBuffer*.

At the top of almost every function entry in the *Win32 Programmer's Reference* in the Development Library, there is a hot spot, *Quick_Info*, which provides essential information for developers. The following illustration shows the Quick Info hot spot for **GetProfileString**.

GetProfileString

Windows NT	Yes	}	<i>Applicable Operating Systems</i>
Windows 95	Yes		
Win32s	Yes		
Import Library	kernel32.lib	—	<i>Lib "Kernel32"</i>
Platform Notes	—		<i>Yes: Both ANSI and Unicode</i>
Unicode	Yes	—	GetProfileStringA and GetProfileStringW <i>No: Function name is used</i> GetProfileString

First, QuickInfo identifies the Win32 operating systems where the function is available. Second, it identifies the library containing the function. Third, it indicates whether this function has separate ANSI and Unicode versions.

The declaration in Visual Basic for Applications is:

```
Declare Function GetProfileString Lib "KERNEL32" _
    Alias "GetProfileStringA"(ByVal lpAppName As String, _
    ByVal lpKeyName As String, ByVal lpDefault As String, _
    ByVal lpReturnedString As String, ByVal nSize As Long) As Long
```

As noted above, the use of the **Alias** control word allows any existing 16-bit code calling the API to be left unchanged. The following 16-bit Windows API code pasted into Microsoft Excel, Microsoft Project, Microsoft Access, or Visual Basic 4.0 will work correctly with the 32-bit **Declare** statement above.

```
rc% = GetProfileString(App$,Key$,Def$,RString$,RLen%)
```

This is because although both rc% and RLen% (the % denotes an integer data type) are the wrong size (2 bytes instead of 4 bytes), the **ByVal** control word and automatic type conversion in Visual Basic, Access Basic, WordBasic, and Visual Basic for Applications will change the size for you. (When any of these languages passes the value to the DLL and gets the results, it will typecast them from **Integer** to **Long** or from **Long** to **Integer** automatically.) This feature removes many porting issues between 16-bit Windows calls and Win32 API calls but also, as mentioned earlier, may cause an overflow on conversion.