Work with the Windows API from within Visual Basic 6

The first question I need to answer is "exactly what is the Windows API?".

The Windows API is a name that collectively refers to the procedures and functions that comprise the Windows Operating System. The procedures and functions are shipped with Windows in libraries, called Dynamic Link Libraries, otherwise known as DLL's. Every operation that takes place on a Windows PC makes use of this collection of procedures and functions contained in these DLL's--but for the most part, as a user of the PC, you never know about them (unless you happen to get one of those nasty error messages when a program bombs referencing a DLL).

When you write programs in Visual Basic, the code that you write makes use of these DLL's---but you never know it, because Visual Basic takes care of interacting with these DLL's for you. For instance, when you display a message box in Visual Basic, VB makes a 'call' to a procedure in one of the Windows DLL's to display the message box.

It's been estimated that 80% of the functionality of the procedures and functions in the Windows DLL's has been incorporated into Visual Basic---therefore, when we speak of the Windows API, we are really talking about getting at that other 20%.

For Instance?

So what types of operations are we talking about in that remaining 20%?

For the most part, really nerdy stuff that most of us would never want to do anyway. But once in a while, a requirement comes along that you'd love to incorporate into your VB program, and there's simply no way to do it within VB.

For instance?

How about this---you want to display a splash screen (a form that appears when your program first starts up), and you want to display it for exactly five seconds before unloading it, and displaying your program's main form.

No big deal you might say---you can just write some code that executes a loop about a zillion times, and there's your five second delay.

That's true---you could code a loop to iterate in such a way that it takes five seconds or so to complete--the problem is that loop may take five seconds to complete on a Pentium III Gigahertz processor---and five minutes on a 486 60 Mhz machine. It seems like there should be an easier way to do this, but unfortunately, VB doesn't come shipped with a statement that I found so useful in my Paradox programming days---Sleep.

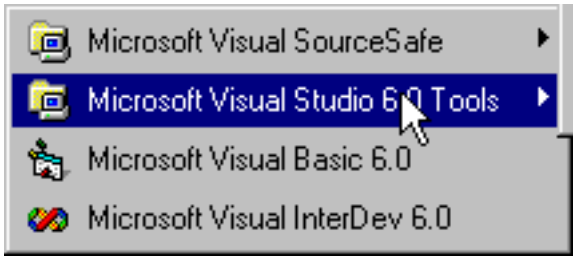
Funny I should mention Sleep, because as it turns out, there is a Sleep procedure included in the Windows API---and it can do exactly what I describe---delay processing for a period of time. The problem is that it's not a VB function---it's a Windows API function. The question is: how to get to the Sleep procedure from within Visual Basic.

What's in the Windows API?

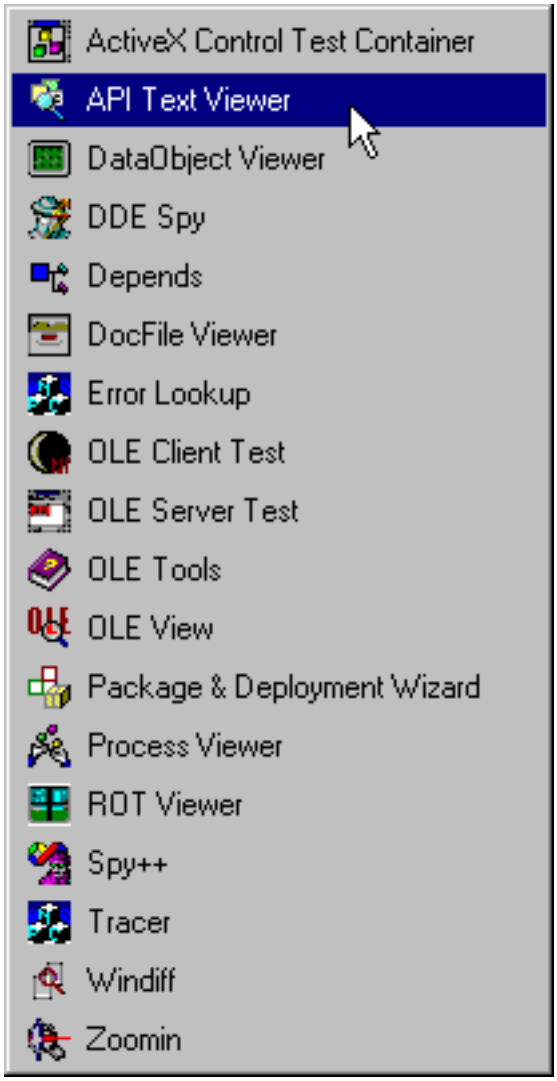
You'll see in a few moments or so that accessing and executing the Sleep procedure from within a Visual Basic program is really very easy. The hard part is really knowing what functions and procedures are available to us in the Windows API---and I'd like to take a few minutes to show you that using the Windows API Text Viewer.

The API Text Viewer

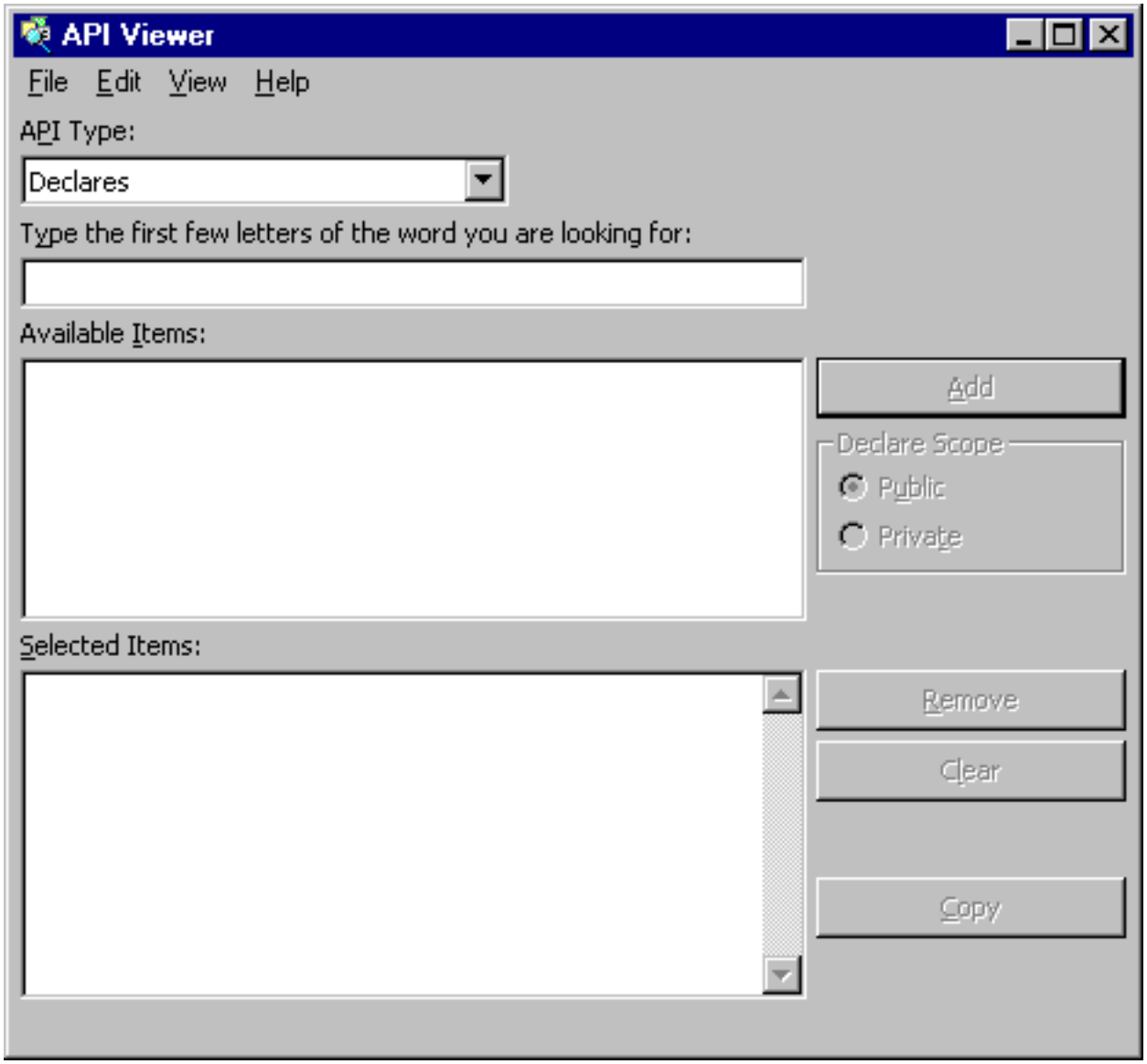
You can access the Windows API Text Viewer by selecting Microsoft Visual Studio 6.0 Tools from the Visual Basic or Visual Studio menu of the Start Menu…



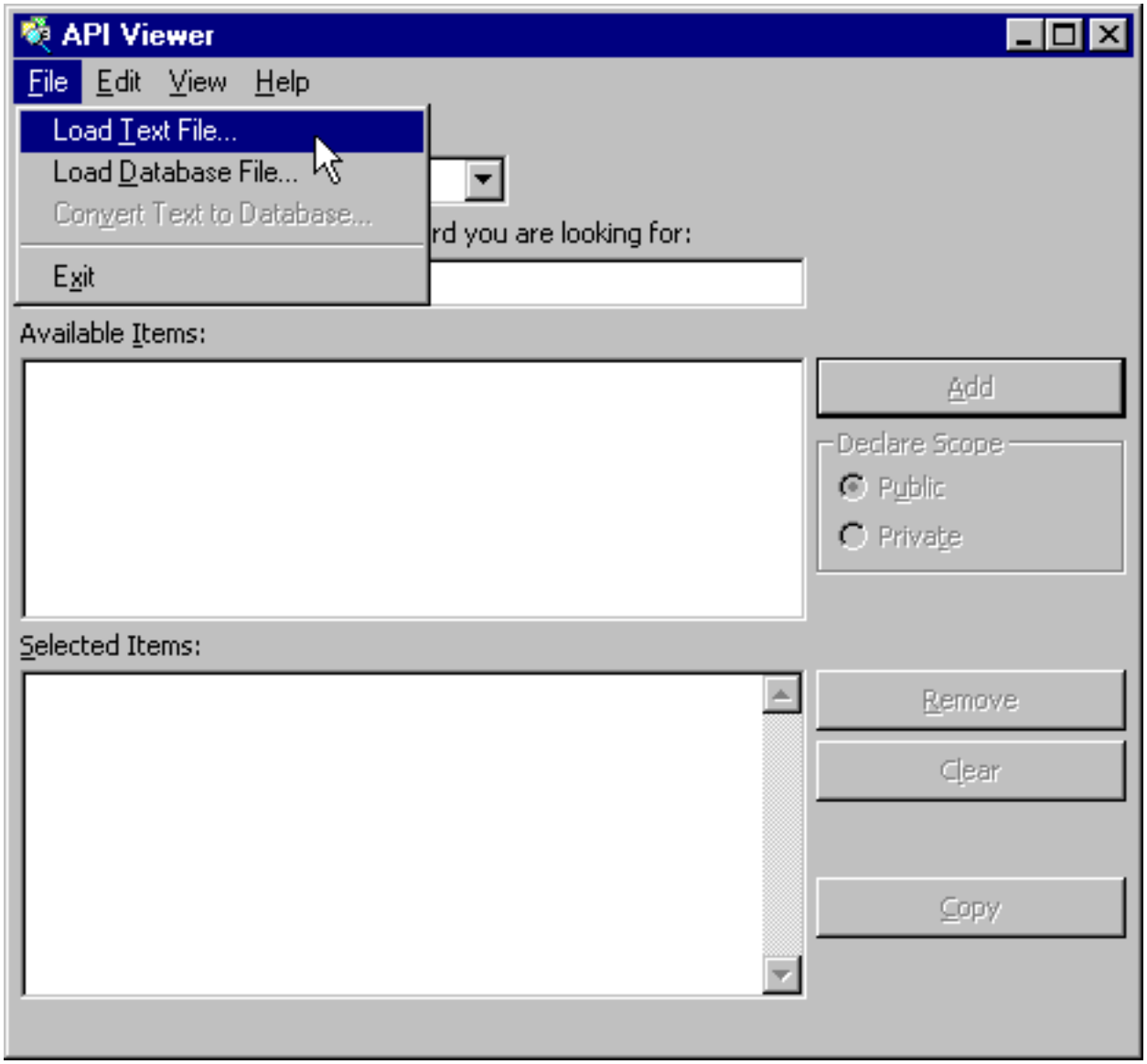
…then select the API Text Viewer…



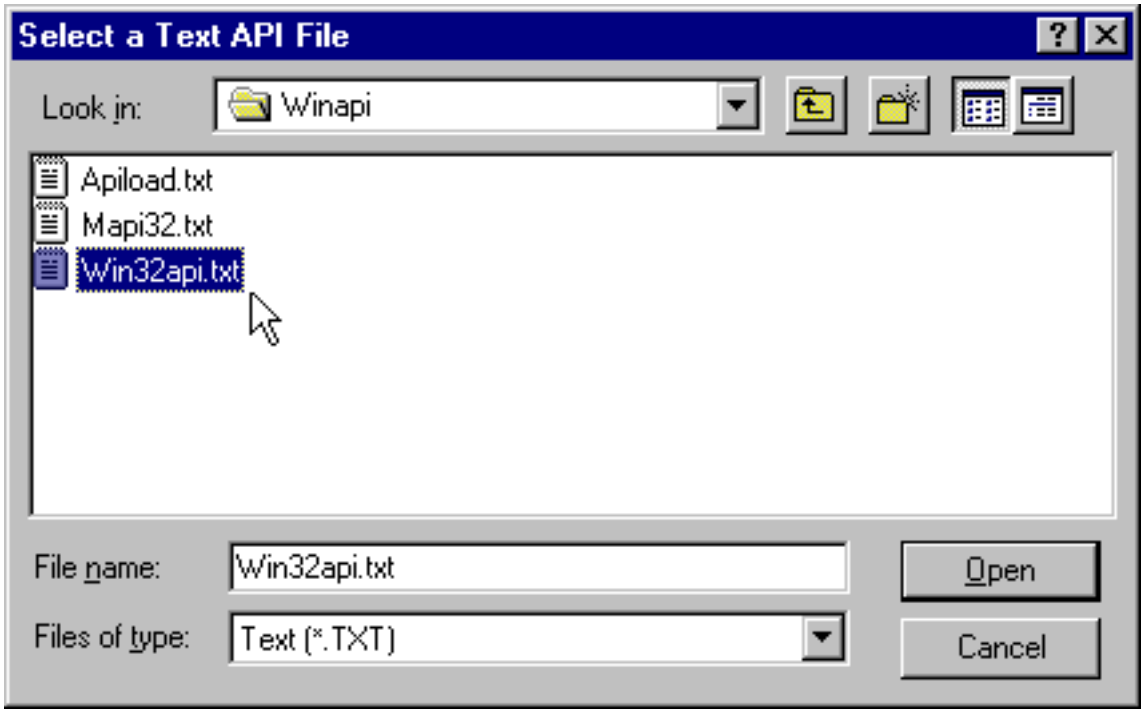
Selecting that menu item opens up the API Text Viewer…



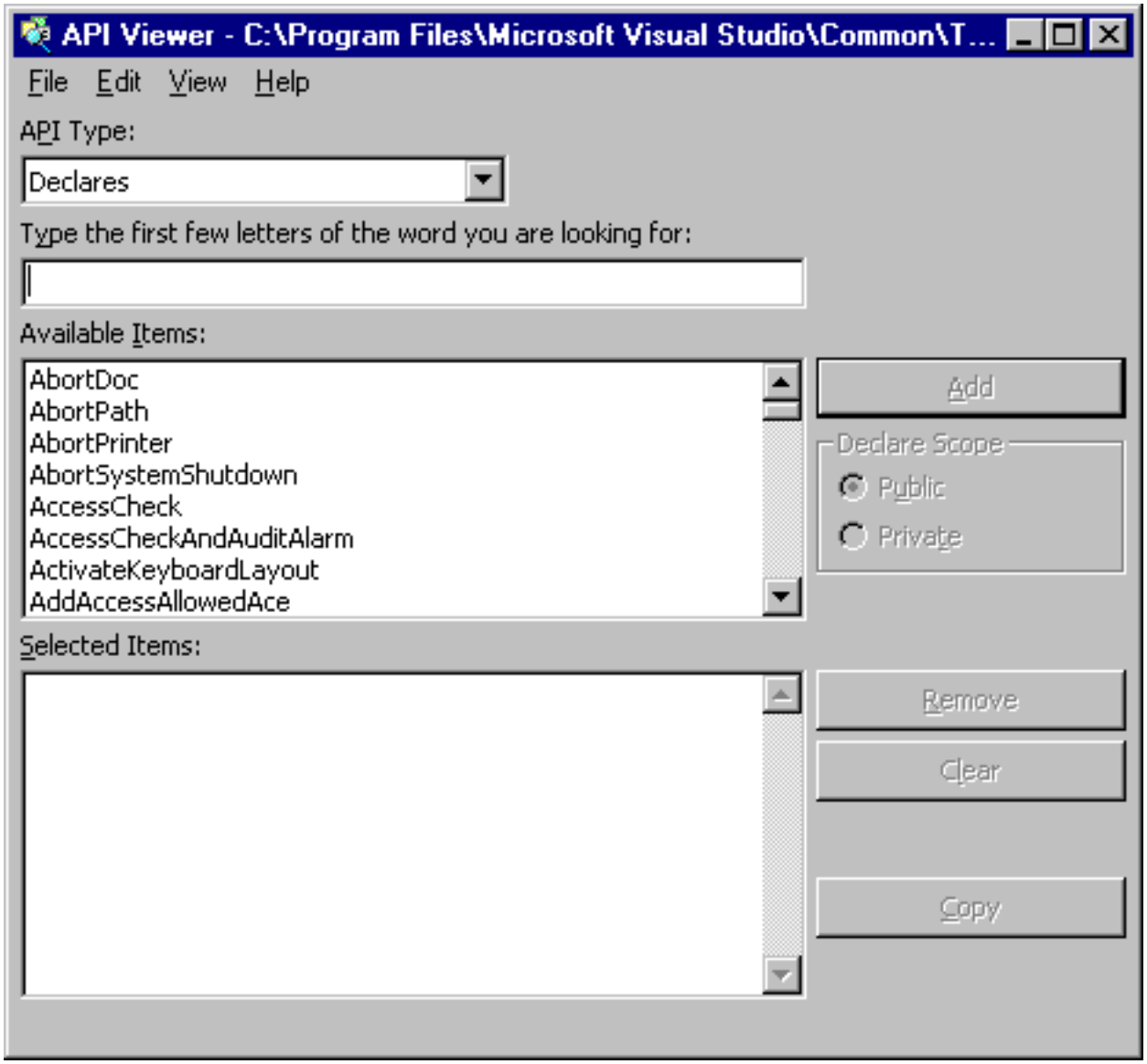
which at first glance can be a little imposing. The API Text Viewer is just an application that allows you to view the contents of one of three text files installed with VB, that provide the barest form of documentation on the functions and procedures contained in the DLL's supplied with Windows. Before we can do anything with the API Text Viewer, we need to make a selection of one of the text files. We can do that by selecting File-Load Text File from the API Text Viewer menu bar…



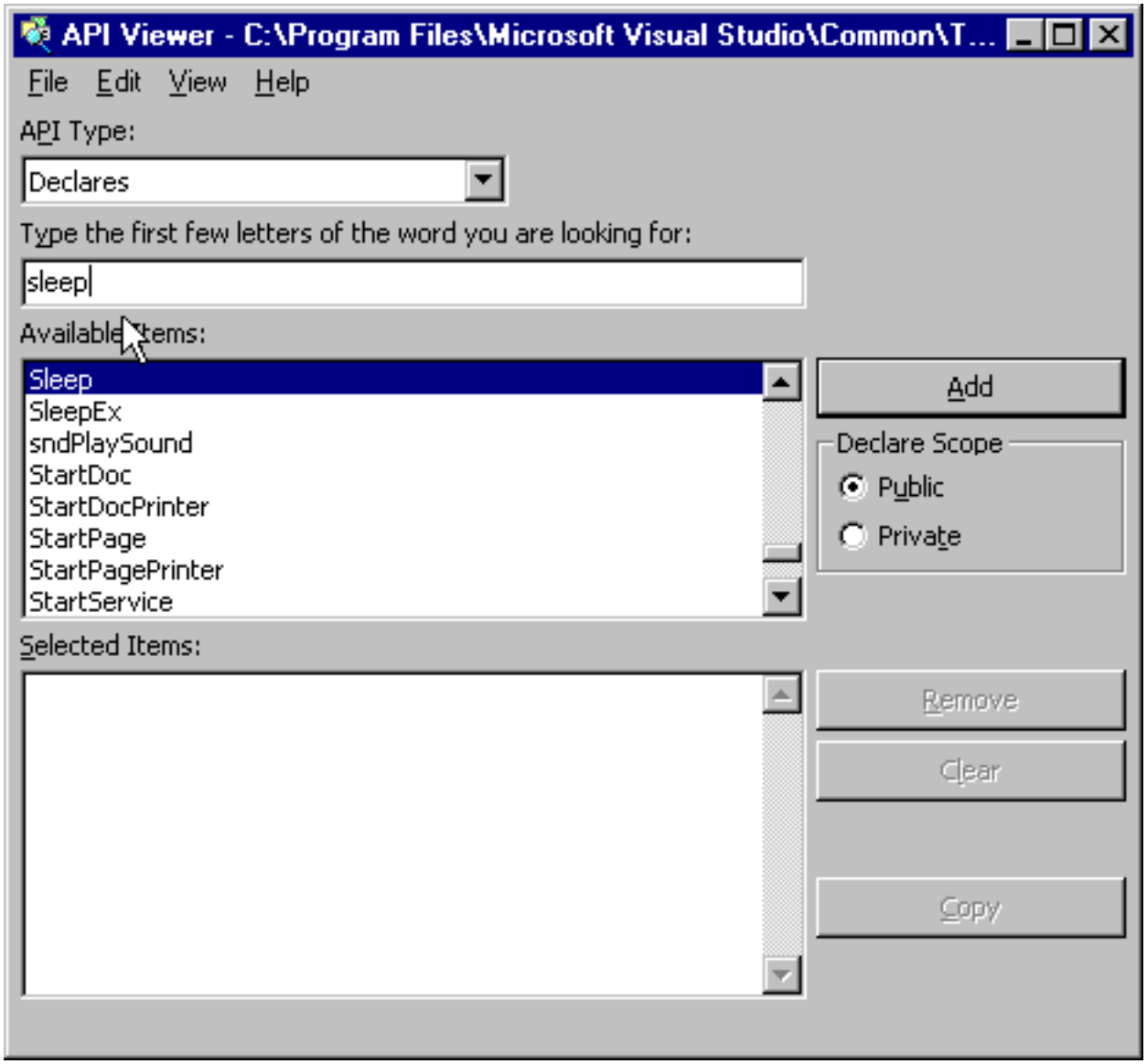
The three available text files supplied for use with the API Text Viewer appear. For this article, we'll be dealing with the Win32api.txt file, which contains documentation on the Windows API. If we select that file, and then click on the Open button…



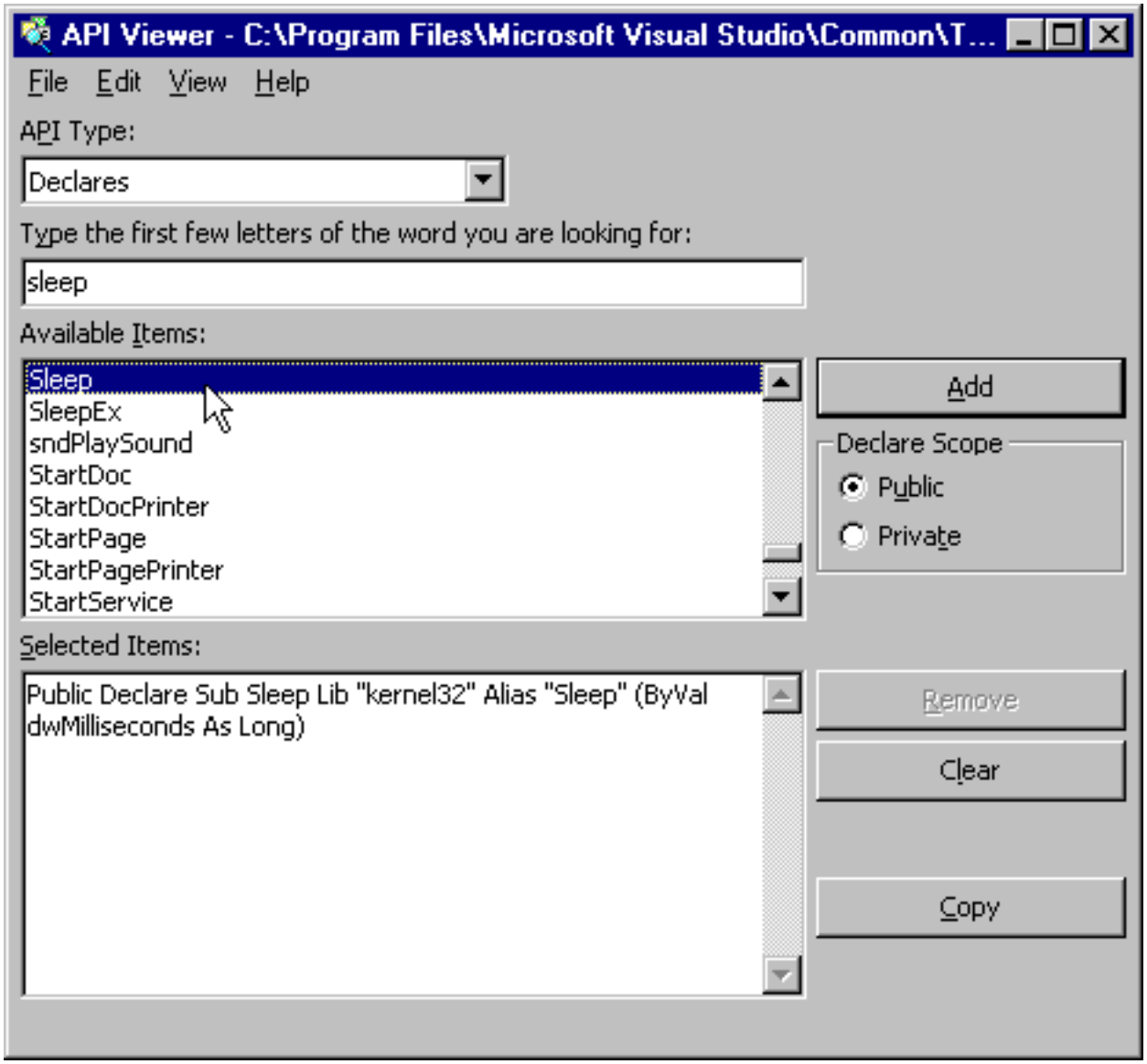
…this window will appear. What you see here are the available procedures and functions (contained in the Available Items ListBox) that comprise the Windows API, and there are a bunch. As you can see, they are listed in alphabetical order, and the first one listed is called AbortDoc.



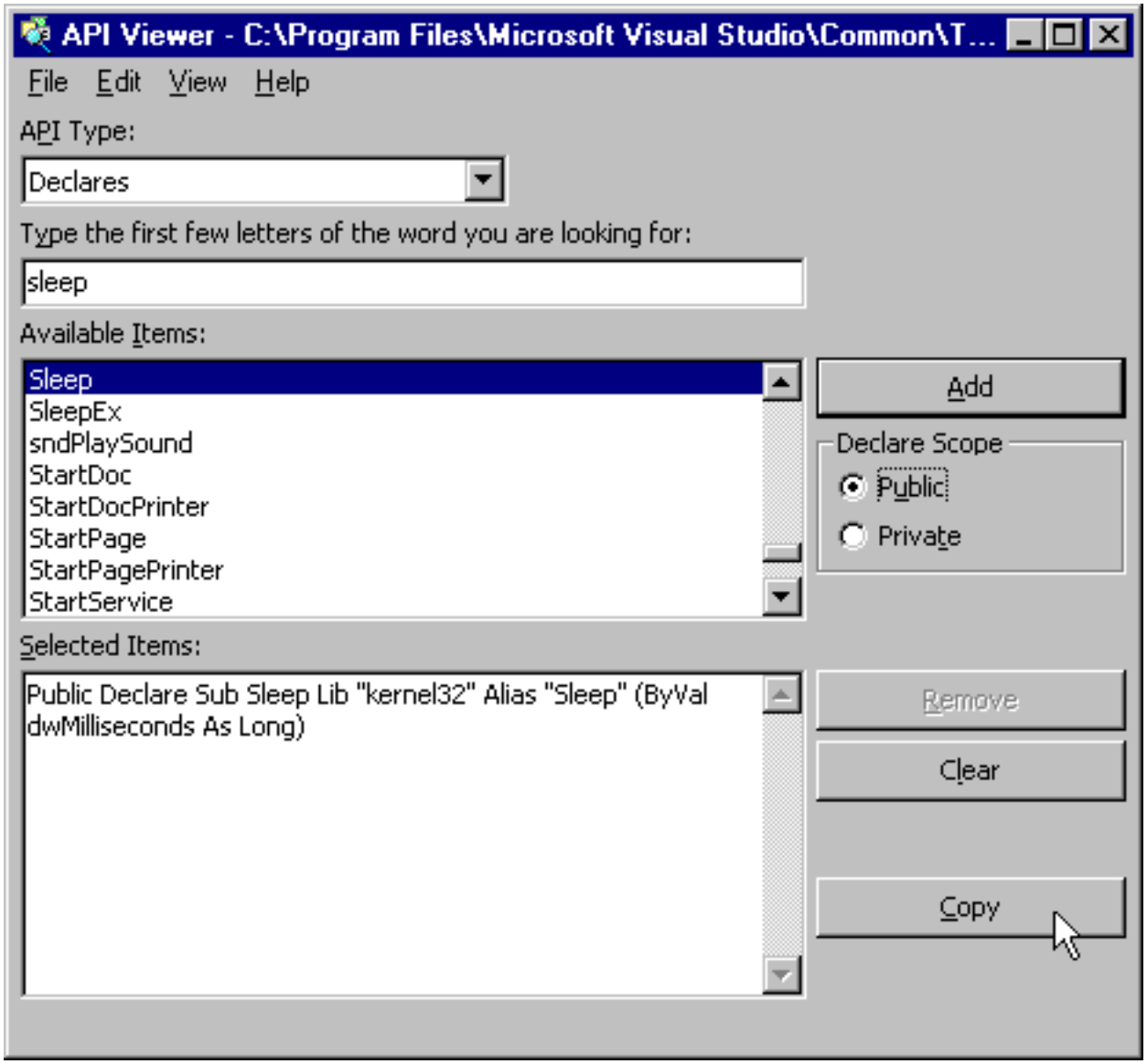
I just happen to know about the Sleep procedure (more on how I know about it later), and the next step I need to take at this point is to find the correct syntax to call the Sleep procedure within my VB program. To do that, I just type the name of the function I am looking for in the textbox above the ListBox labeled 'Available Items:". As I do that, the items in the Available Items ListBox scrolls, and there's the Sleep procedure…



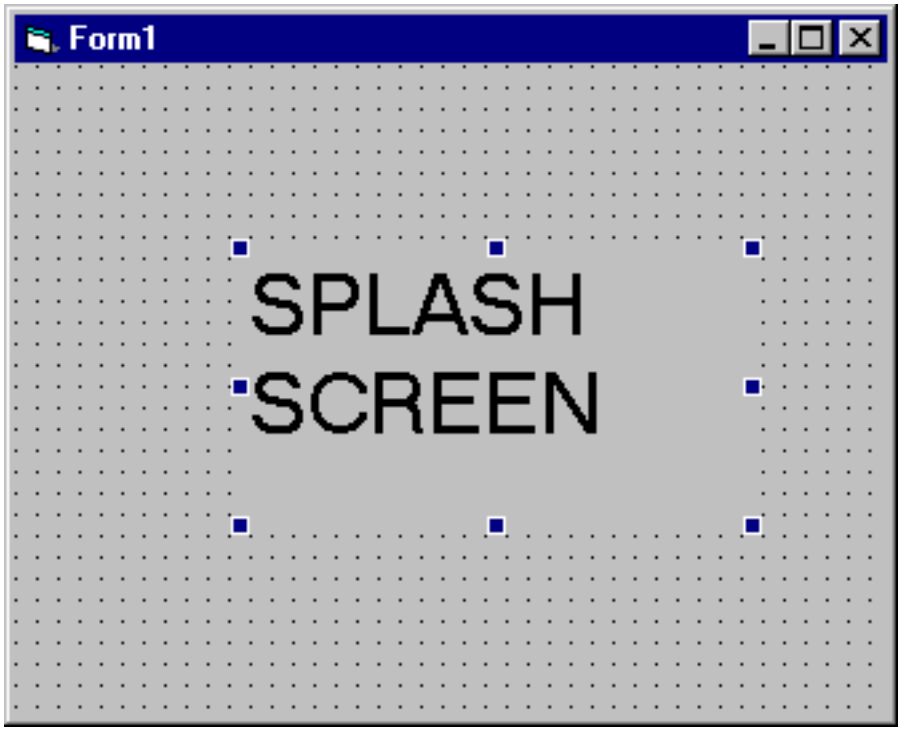
My next step is to click on the Sleep procedure in the Available Items ListBox, and when I do, the correct declaration syntax for the function will appear in the ListBox labeled "Selected Items:"…



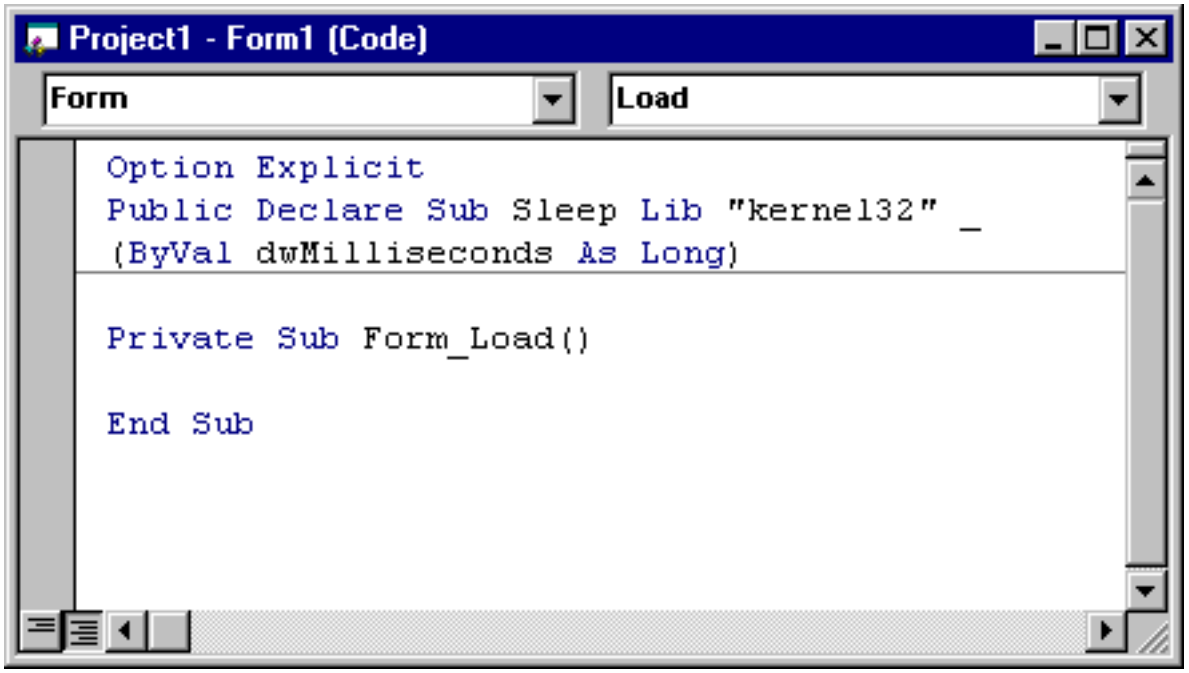
At this point, I need to copy the declaration from the API Text Viewer into my Visual Basic program. The API Text Viewer makes that quite easy---all we need to do is click on the Copy button, and the contents of the ListBox will be copied to the Windows clipboard…



I'll explain the code in a moment or so, but for now I'll start a new Visual Basic project, with two forms named Form1 and Form2. On Form1, I'll place a label control captioned "SplashScreen", like this…



and in the General Declarations Section of the form, I'll paste the declaration for the Sleep procedure I just copied from the API Text Viewer (I placed a line continuation character in the middle of the declaration so that you can see the code better)



The Declare Statement

Let's take a look at the declaration for the Sleep procedure now.

Public Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

I don't know whether you noticed or not, but when I copied the declaration from the API Text Viewer, the reference to an 'alias' was dropped. The 'alias' statement of the Declare statement allows you to refer to a function or procedure in a Windows DLL with a different name within your Visual Basic program. The reason for this is that there are certain Windows API functions and procedures that have the same name of Visual Basic reserved words---and therefore would cause a compile error in your program. If you find yourself in the position of using such a function or a procedure, use the Alias statement--and I must warn you the syntax is a bit confusing. For instance, suppose we wanted to refer to the Sleep procedure within our program as WakeUp---this would be the syntax.

Public Declare Sub WakeUp Lib "kernel32" Alias "Sleep" (ByVal dwMilliseconds As Long)

What's confusing is that the actual name of the function or procedure in the DLL follows the Alias keyword. The true Alias follows the keyword Declare! Just tuck this into your mental database somewhere---it's not likely to come up in your work, but of course, something like this might be asked on one of the VB Certification tests.

Let's look at the syntax of the Declare statement now.

Public Declare Sub Sleep Lib "kernel32" (ByVal dwMilliseconds As Long)

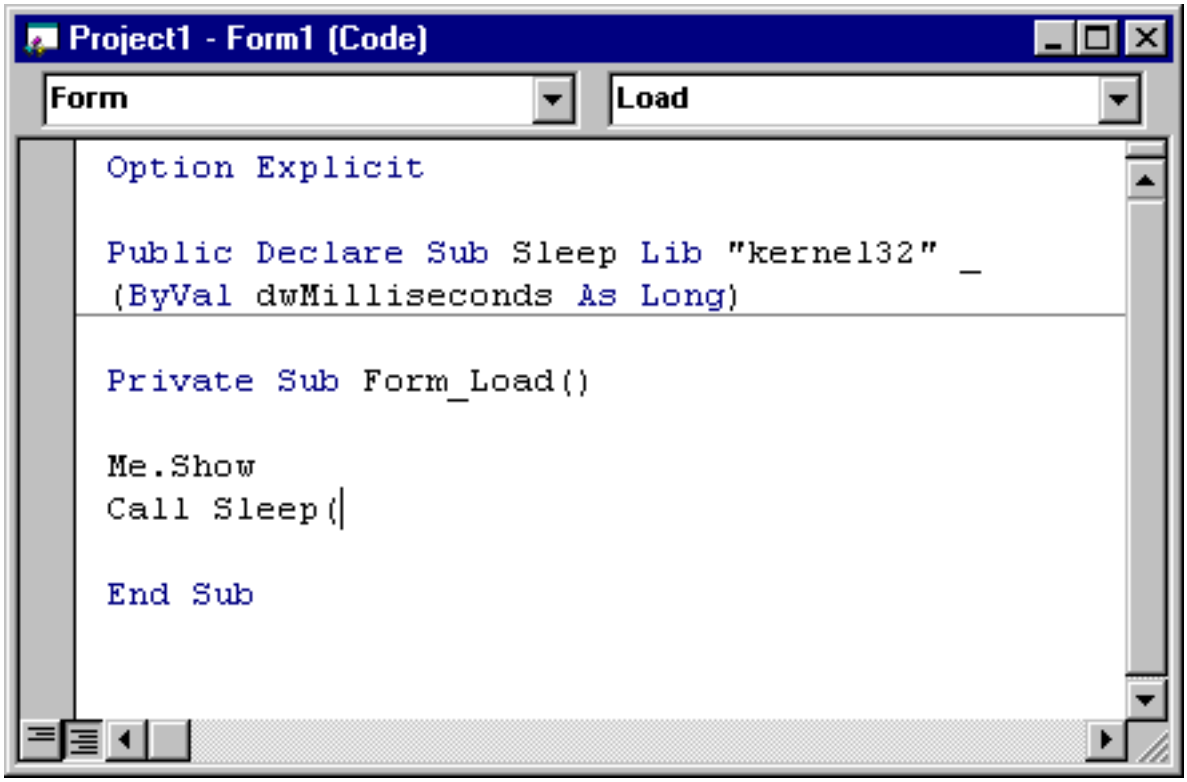
Declarations are either Public or Private, the same as any Visual Basic procedure. The Declare keyword follows that, followed by the word Sub (for subprocedure) or Function (for a function). After that comes the name of the procedure, followed by the keyword "Lib" which stands for library, and then the name of the Windows DLL that contains the Sleep subprocedure. It so happens that Sleep is found in "kernel32" which is located in the Windows\System32 directory. (There's no need to specify the full path of the DLL---neither should you specify the .dll extension---just the name).

Any arguments that the function or procedure is expecting are specified next---in this case, the Sleep subprocedure is expecting a single argument of the long data type called dwMilliseconds---which represents the length of time in milliseconds that the procedure should 'sleep').

At this point I need to warn you about the importance of passing the correct type of data to any procedure you call via the Windows API. Most functions and subprocedures in DLL's are written in C---and C is very unforgiving (unlike VB) if you pass it a data type it is not expecting. So be very careful. I should also warn you that when you call a procedure in a DLL, you are crossing the safety of your VB program's boundaries---and if you call a procedure in a DLL incorrectly, you could 'freeze' your PC--therefore, always be careful to save any work you're doing on your PC before executing a program that calls a Windows API procedure---that includes work you're doing in other applications such as Word or Excel.

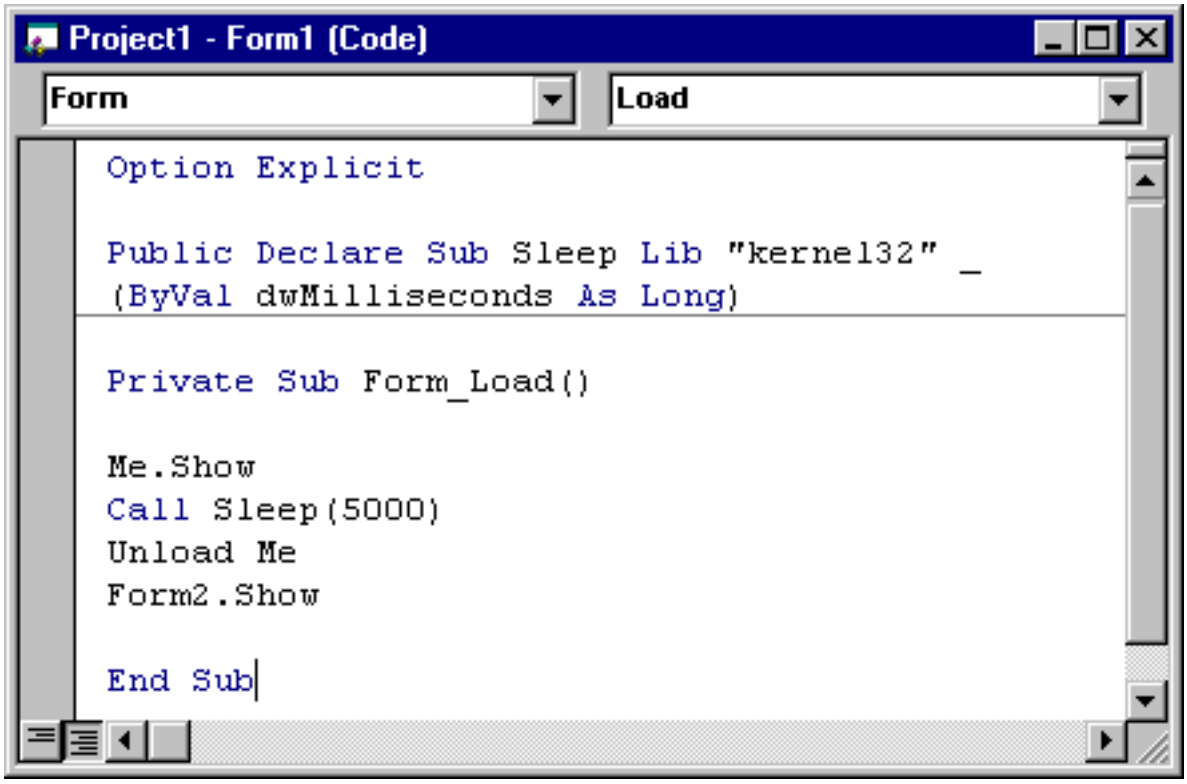
Calling the API Procedure in your code

Now let's write the code to call the Sleep procedure. Since this is a Splash Screen, we'll place code in the Load Event of Form1 to call 'Sleep' using the Call statement. The first thing we do is execute the Show Method of Form1. Since the startup form (in this case Form1) is not actually made visible until the code in the Load Event Procedure wraps up, we need to explicitly make the form visible before calling the Sleep procedure, like this…

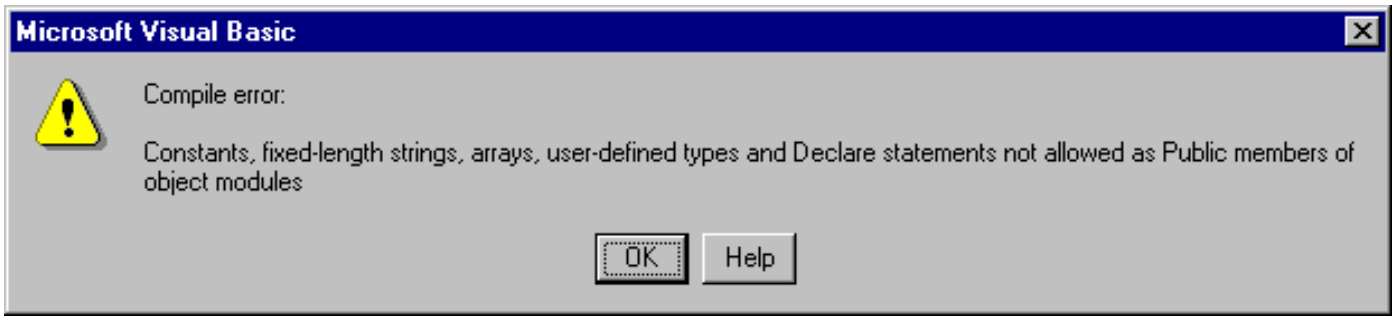


Notice that as soon as I type the Left Parenthesis, VB provides me with help, telling me

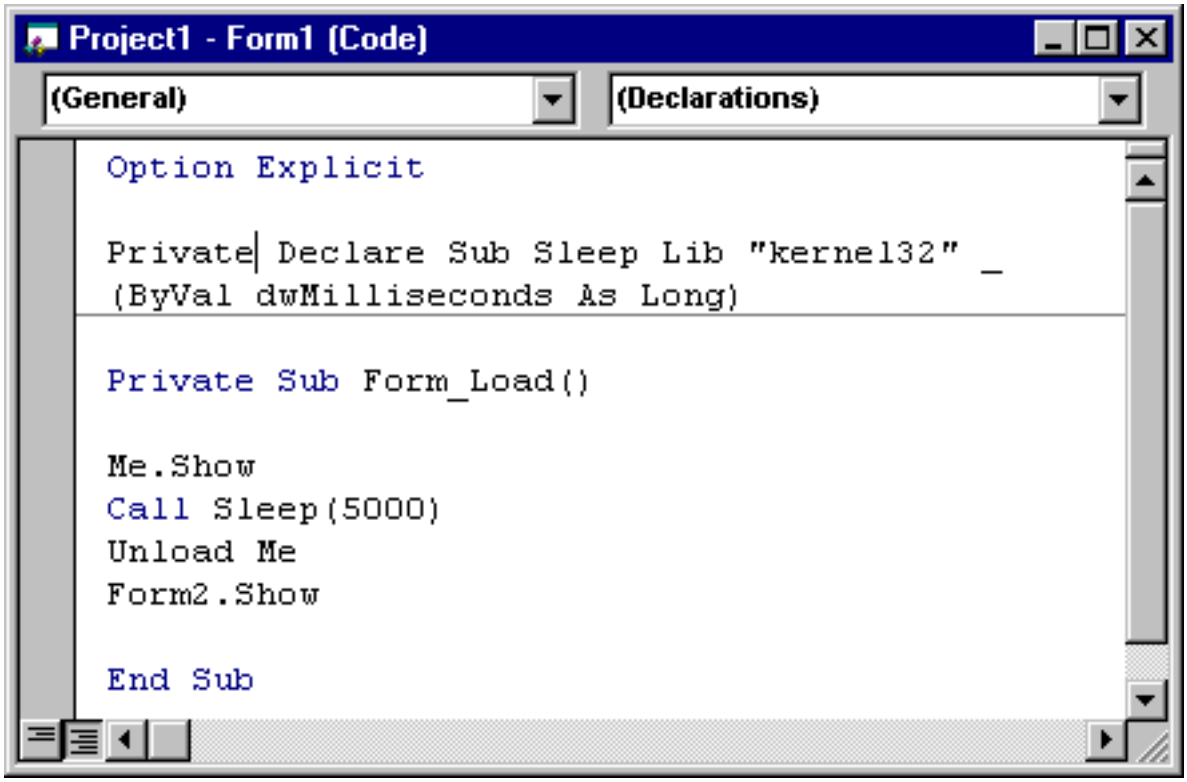
that Sleep requires a single argument of the Long data type. How does VB know this? From the Declare statement we coded. Here's the rest of the code…



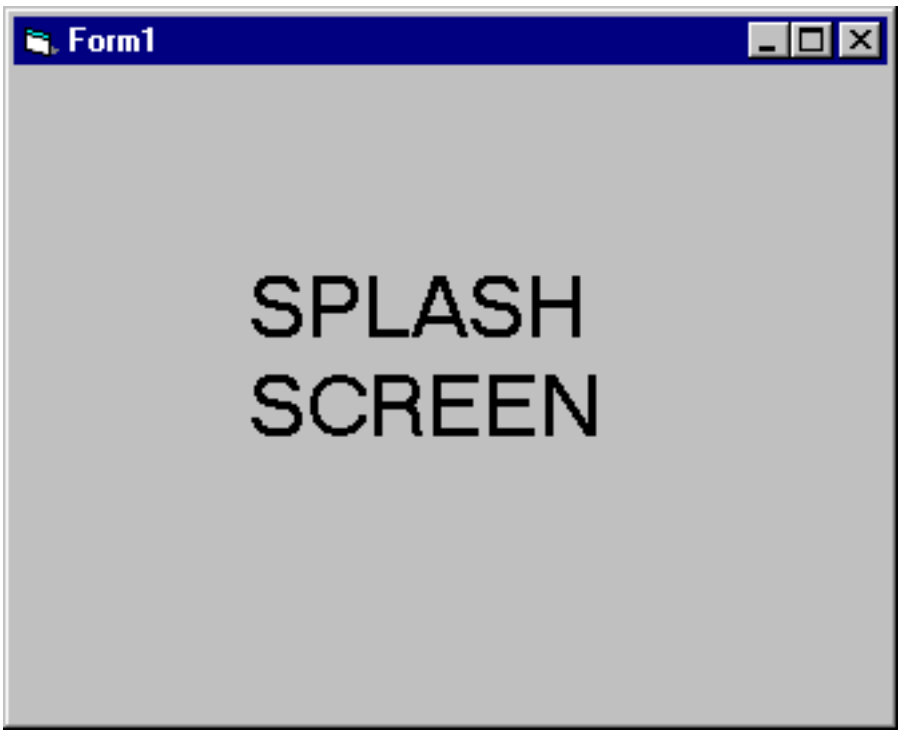
We're calling the Sleep procedure, and passing it an argument of 5000 milliseconds---or 5 seconds. After Sleep wraps up, we'll unload Form1 using the Unload Me statement, and then load Form2 and make it visible using the Show method of Form2. If we now run the program, we get…



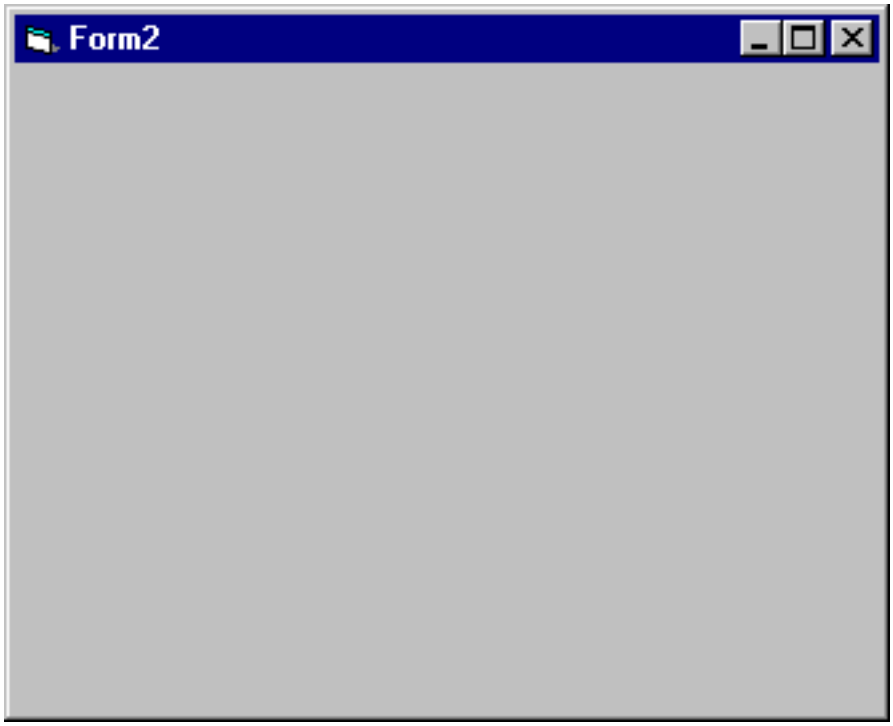
a syntax error! Basically, what VB is telling us is that we can't declare the Sleep procedure in the General Declarations Section of the form using the Public keyword. To fix that, all we need to do is change the keyword Public to Private…



…and then run the program. This time, Form1 appears…

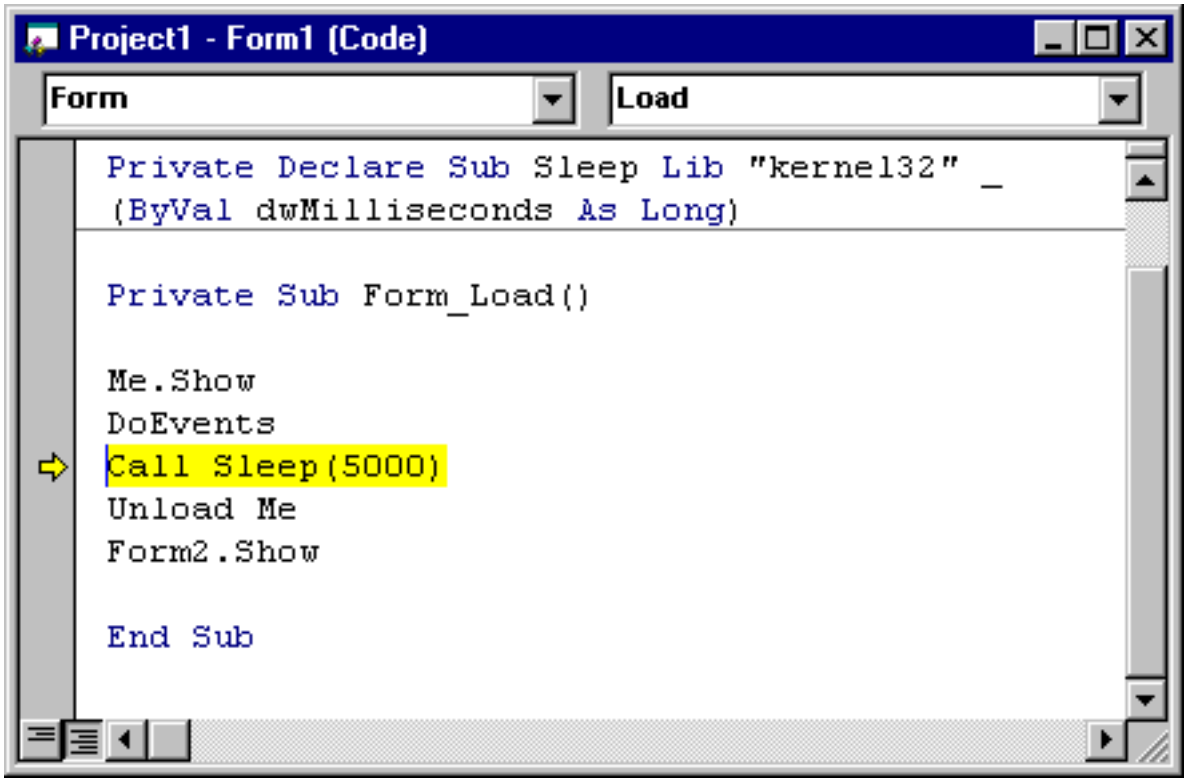


and then five seconds late, Form1 disappears and Form2 appears.



For good measure, you might want to code a 'DoEvents' statement following the Me.Show statement in the Load Event Procedure of Form1.

It's interesting to note that if you run your program in 'Step Mode' (run it using F8), when the program executes the line of code to call the Sleep procedure…



you won't actually see any of the code from the DLL run. DLL's are compiled code, and you can't see the code run in Step Mode.

One more example

I mentioned earlier that calling procedures in the Windows API is a good deal easier than determining that they exist in the first place. Aside from the API Text Viewer, there's really no way to know that a particular function or subprocedure exists, unless you do one of two things: buy the Windows Software Developers Kit from Microsoft, or purchase a book on the Windows API. The book I recommend is by Dan Appleman, and it's entitled

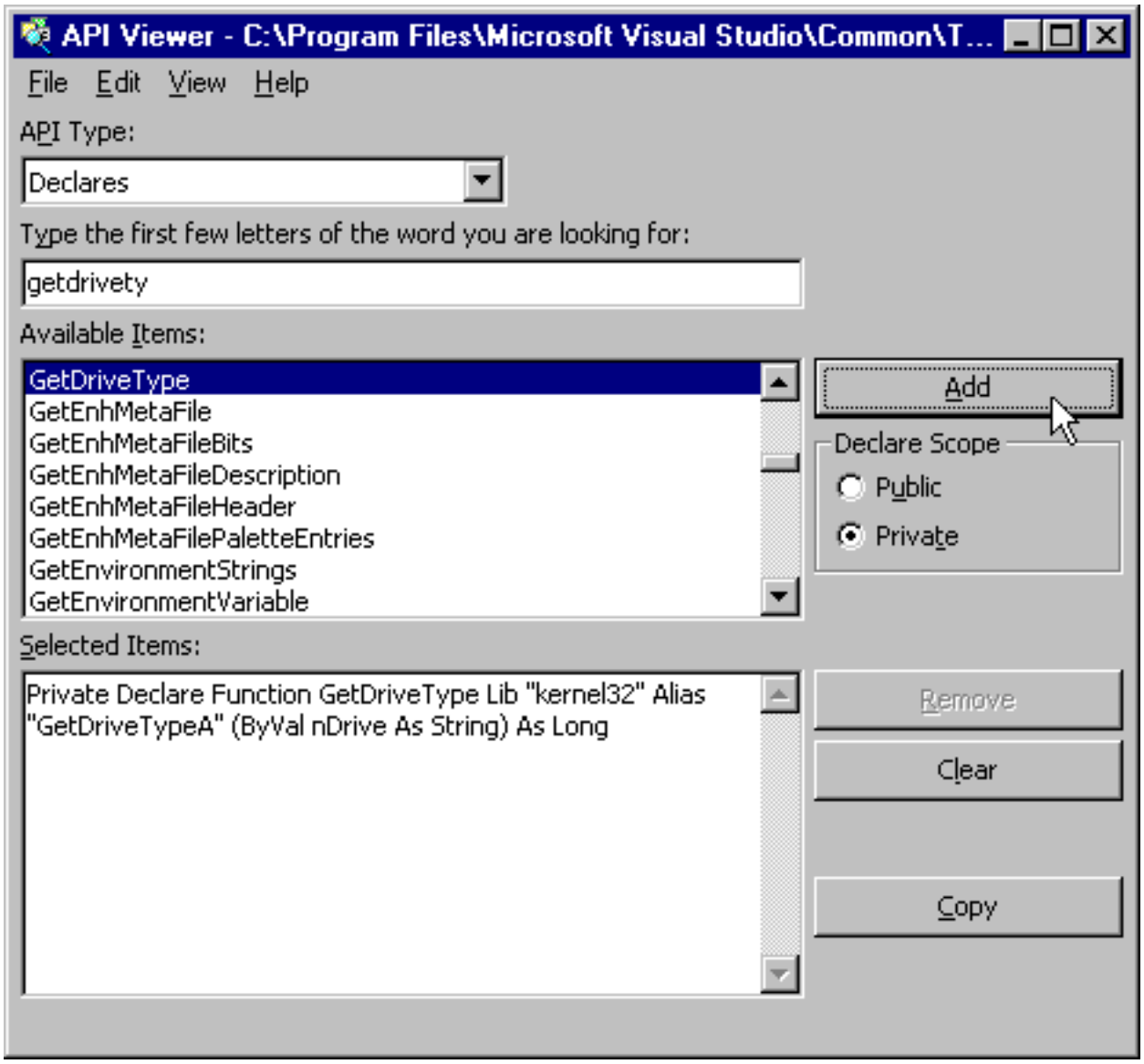
Dan Appleman's Visual Basic Programmer's Guide to the Win32 API
ISBN: 0672315904

It's a great book. It lists a thousand or so Windows API procedures, along with a brief explanation as to how they work and behave. The book is nicely organized according to procedure type---so that all of the procedures pertaining to Disk Operations appear together, all of the procedures pertaining to Printer Operations appear together, etc. There's also a section in the book that describes the caution you need to take when calling C functions and subprocedures within your Visual Basic program.

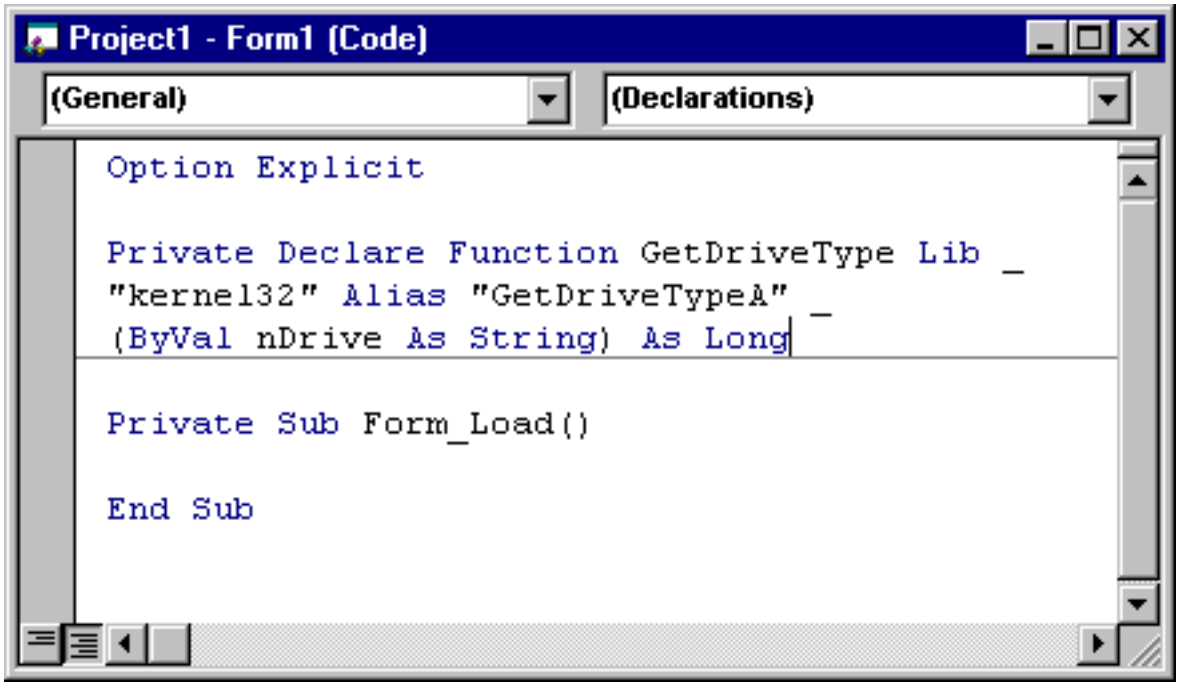
As I said, without either the Software Developers Kit or a book such as Appleman's, you're basically on your own. You can do what I did several years ago, and just start browsing through the Declaration statements that you find in the API Text Viewer. You may find one that sounds interesting and decide to experiment with it. I'm going to highlight a procedure

called GetDriveType which I found while browsing through Text Viewer, and use it to illustrate working with an API function versus the sub procedure Sleep we just examined. Remember, a function returns a value to the program that calls it, and this function will return a value to your program telling you the type of drive associated with a particular drive letter (bear in mind that without some form of documentation, knowing that this procedure exists and knowing how to use it can be extremely difficult…)

We'll start by using the API Text Viewer to get the Declare Statement for GetDriveType. Notice that I selected Private for the Declare Scope this time…



At this point, I'll create a new Visual Basic project with a single form containing a textbox and a command button, and copy and paste the Declare statement into the General Declarations Section of the form.

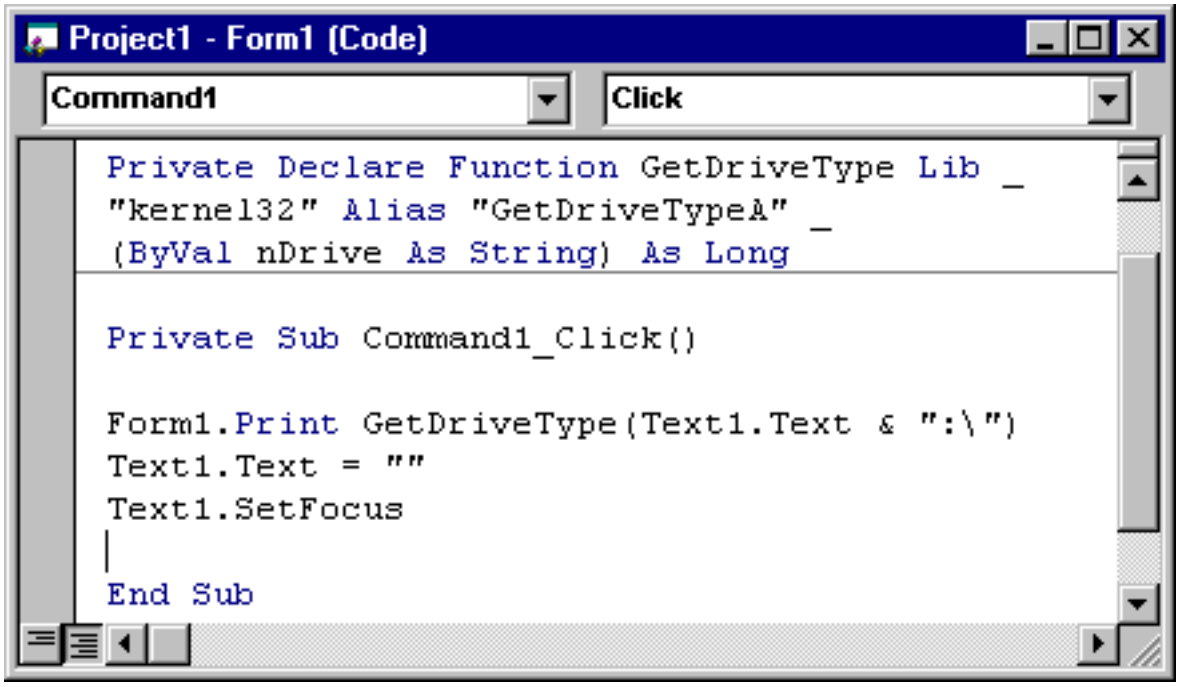


A couple of words about the Declare Statement. GetDriveType is a function (thus the keyword Function in the Declare Statement). It expects a single argument of type String, and because it's a function, it will be returning a value to our program when we call it---in this case a value of type Long, which is a number.

Let's write the code to call this function now. What we'll be doing is taking a Drive Letter that the user enters into a Textbox, and call the GetDriveType to determine the type of Drive associated with that Drive letter (this can be quite useful if you want to determine whether drive letter E on a user's PC is a CD-ROM or a Hard Disk Drive).

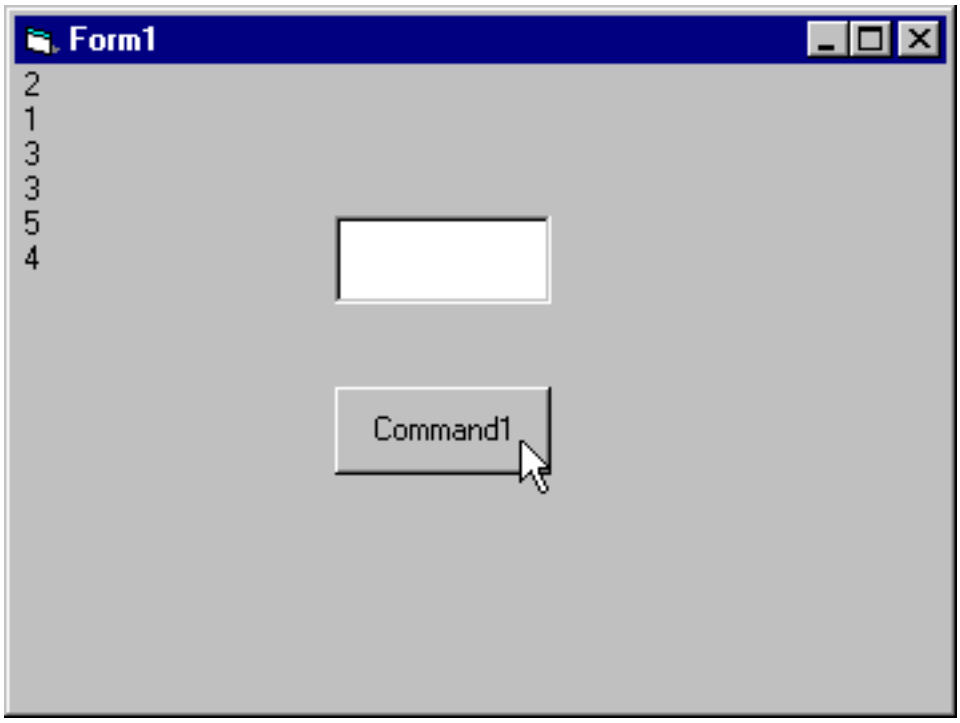
GetDriveType is a little tricky in that it requires a full path name to be passed to it, not just the drive letter (i.e. C:\ not C). How did I discover that? Without documentation, it was hit or miss for a while, and then a lucky guess. Dan Appleman's book will point this out for you though.

We won't ask the user to type in the full path name--we'll take care of that ourselves by concatenating the colon and the backslash to whatever the user enters into the textbox. If this were an actual production program we would also add some validation code to ensure that the user enters only a single drive letter--but I'll leave that for you to do later. Here's the code for the Click Event Procedure of the Command Button.



This code is pretty simple. All I'm doing is taking the drive letter that the user enters into the textbox, concatenating a color and backslash to the end of it, and then passing it to the GetDriveType function in the kernel32 DLL. Since GetDriveType is a function, we have to be prepared to accept a return value, and we'll use the Print Method of the form to display the return value on the form, after which I clear the Text property of the Textbox and set focus to it for the next entry,

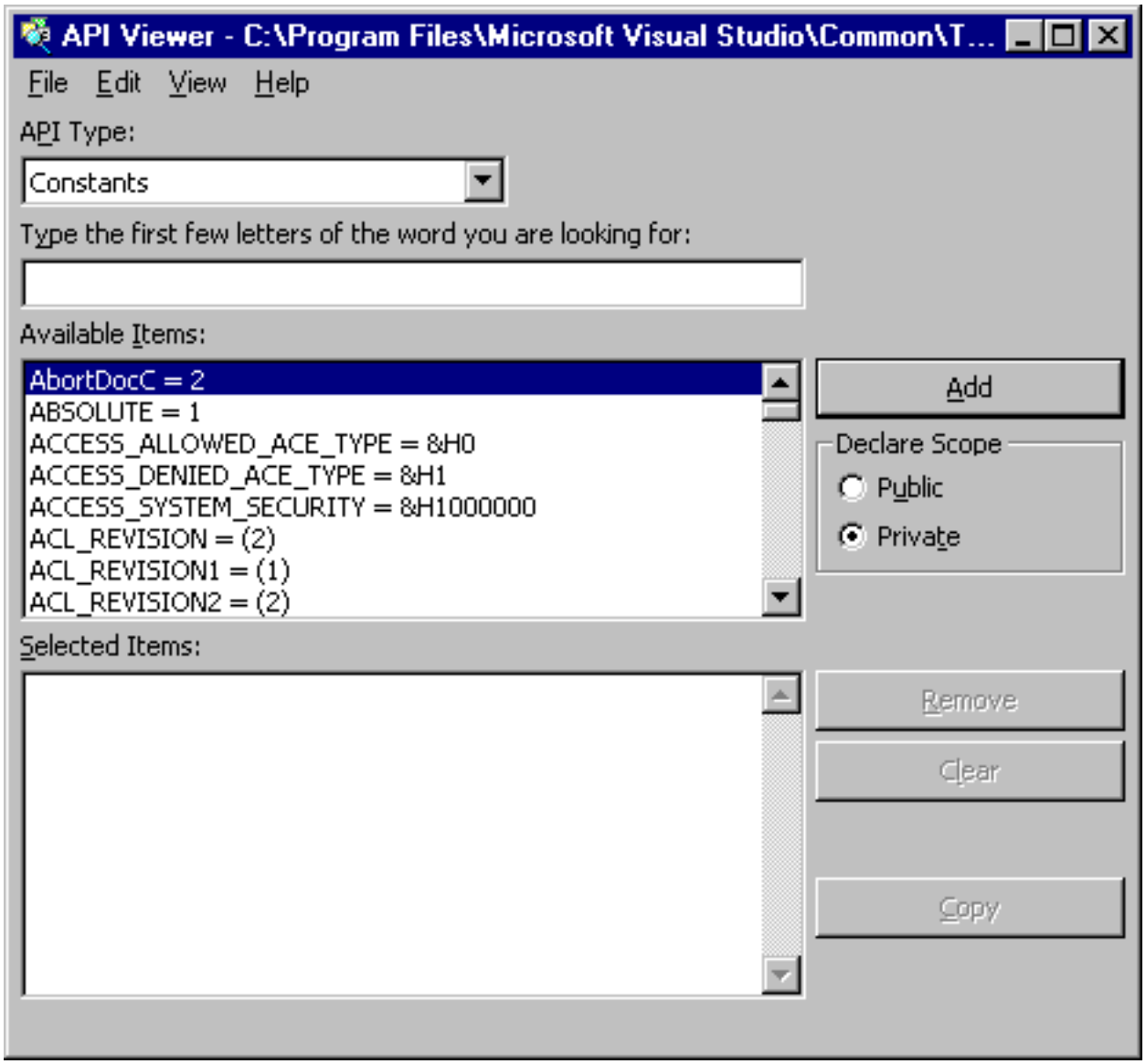
If we now execute this program, and enter values of A, B, C, D, E and H into the Textbox and press the Command button after each entry, we'll see this screenshot…



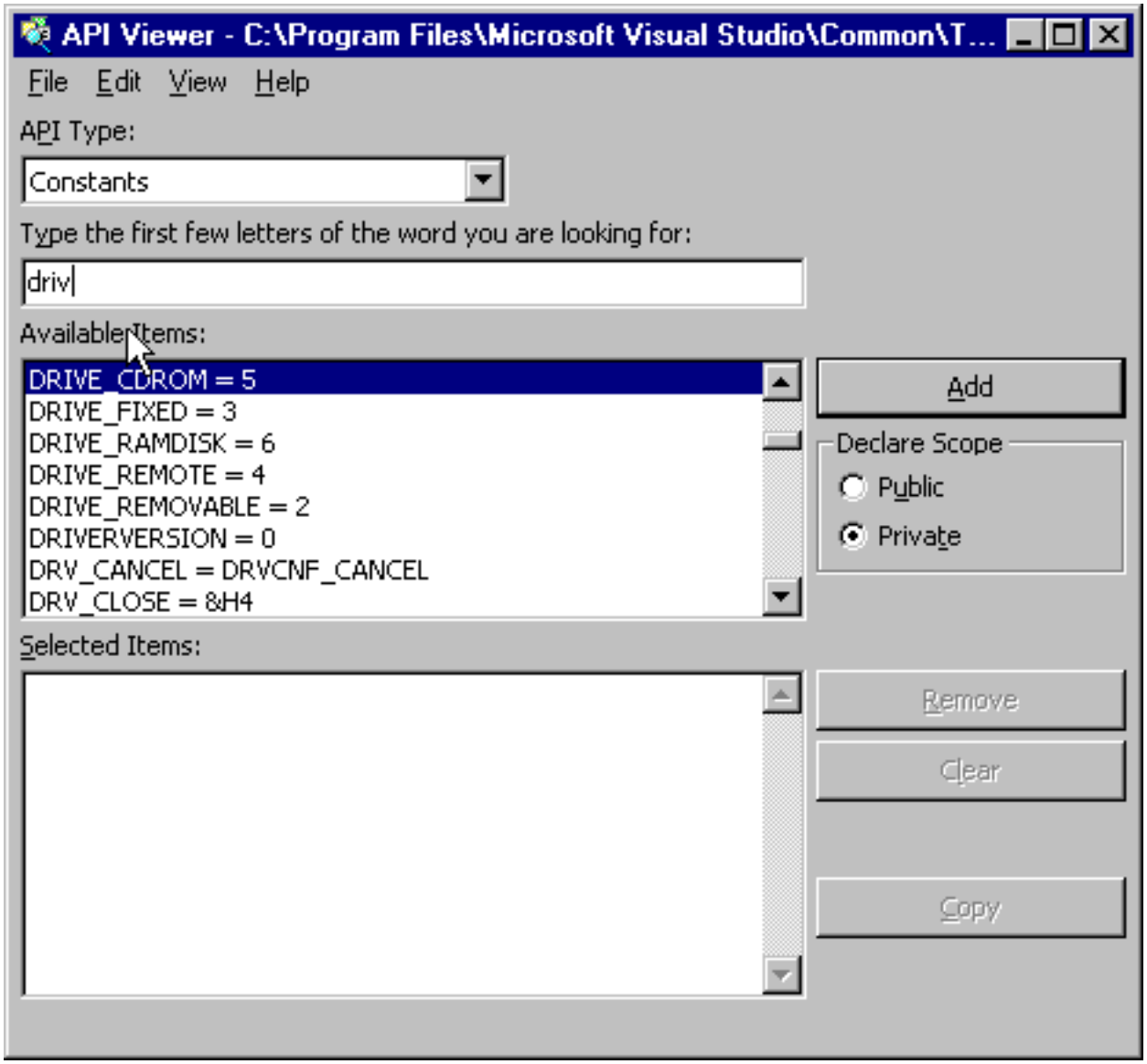
You're probably saying that this is nice, but what do the numbers mean?

Remember, I told you that working with functions and subprocedures in the API can be a guessing game---there's no documentation provided. I'll tell you ahead of time that Dan Appleman's book will tell you what these numbers mean. Without it, you can guess (for instance, the number 2 seems to equate to a floppy drive, the number 3 to a Hard Disk Drive (my PC's C and D Drives are both Hard Drive). 1 is a non-existent drive (I have no Drive B). 5 is a CD-ROM (My E Drive), and 4 would appear to be a Network Drive (Drive Letter F on my PC is a network Drive).

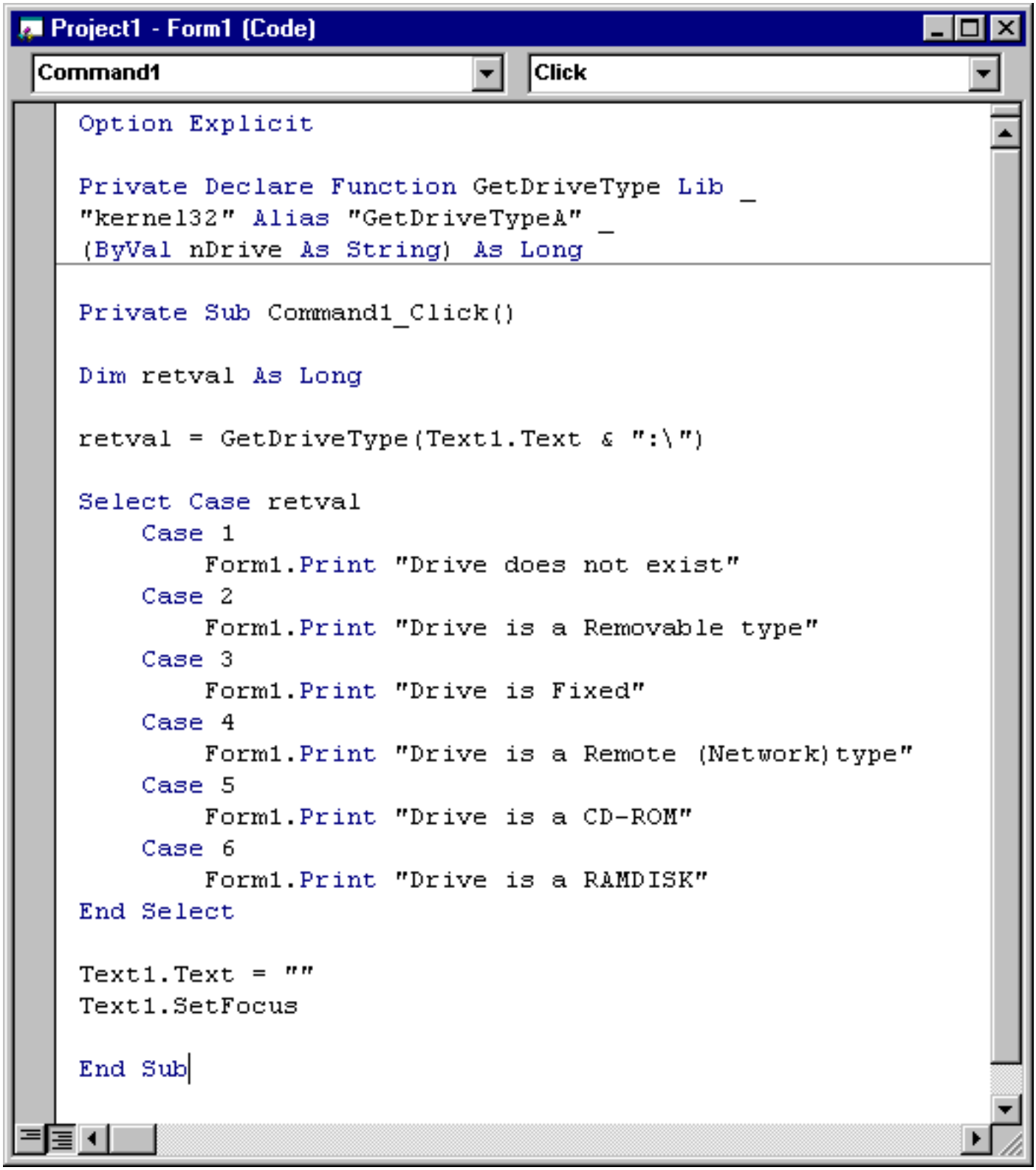
Guessing is hardly ideal, and if you don't want to purchase Dan Appleman's book, you can try to use the API Text Viewer. The API Text Viewer has a list of Constants that are used within the API…



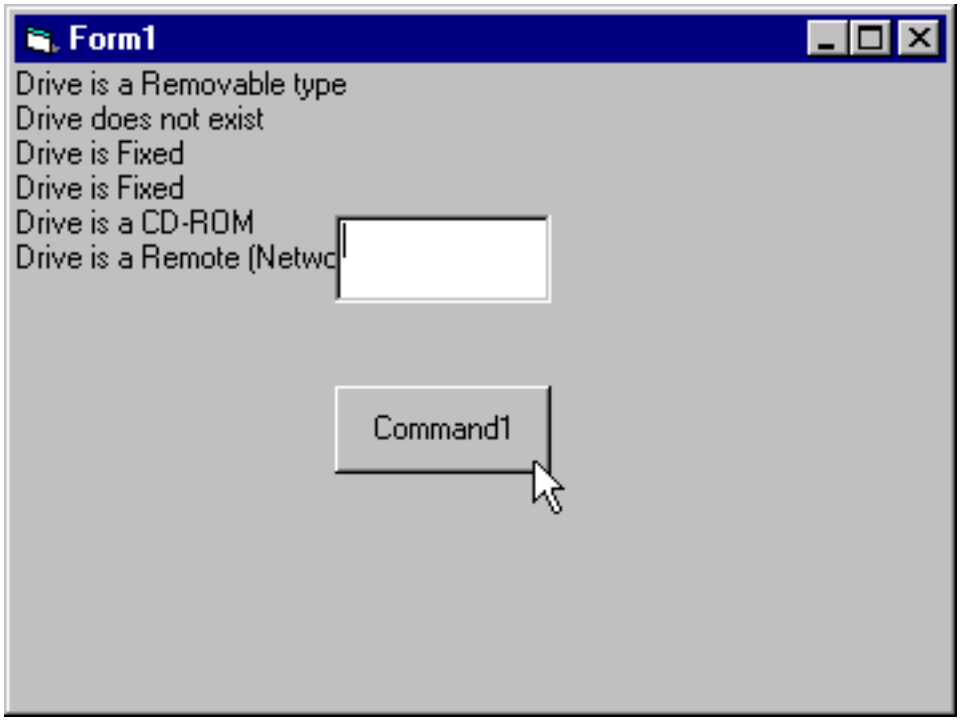
It so happens that the return values associated with the GetDriveType function are represented by Constants beginning with 'drive'



As you can see, there are the values (with the exception of the non-existent drive) that we guessed. If we wanted, we could include these values in a Select…Case structure and make the display of the drive type more user friendly, like this…



Now if we run the program again, once again typing A, B, C, D, E and H into the Textbox, this will be the result…



Summary

I hope you've enjoyed this foray into the world of the Windows API. Using the Windows API can make an extraordinary amount of powerful functions and procedures available to you.

Feel free to use the API Text Viewer to experiment with functions and subprocedures in the Windows API---but remember, save all of your work first!