Integrating Windows APIs and DLLs Using WinWord

Presented by: **Woody Leonhard**

*Woody is president of Pinecliffe International, purveyors of the critically acclaimed **WOPR** — Woody's Office POWER Pack ™ — the #1 enhancement to Word for Windows. He's also a Microsoft Consulting Partner.*

*Woody wrote **Windows 3.1 Programming for Mere Mortals**, the seminal guide to Windows programming in Visual Basic, WordBasic, and other "macro" languages. **Mere Mortals** shows you how to beat Windows into submission without expensive, complex tools: if you can write a macro, you can program Windows.*

*Woody and Vincent Chen teamed up for the ultimate, no-bull reference to things WinWord, the **Hacker's Guide to Word for Windows**. If you prefer your reference books without the sugar coating, check the **Hacker's Guide**.*

Pinecliffe International
Coal Creek Canyon, Golden, Colorado USA 80403-0100

***WOPR** order line: 800-OK-WINWORD (800-659-4696)*
Outside USA: 314-965-5630

Windows 3.1 Programming for Mere Mortals, *Addison-Wesley, 1992, ISBN 0-201-60832-4. "Irreverent ... a painless introduction to such arcane issues as API calls and DDE." — PC Magazine, 10/27/92.*

Hacker's Guide to Word for Windows, *Addison-Wesley, 1993, ISBN 0-201-63273-X. "It's a wonder anybody at Microsoft still talks to me." — Woody Leonhard, 1/15/93.*

Why?

That's the obvious question, eh? Why would anybody in their right mind want to monkey around with the innards of Windows, from inside a word processor?

The answers to that question are as varied as Word for Windows users themselves.

Do you want to check and see if another program is running? Ever need a stopwatch? How about some place to stick WordBasic variable values that won't disappear? Want to force WinWord to "let go," so other Windows programs can get a little work done? If so, the solution lies in Windows API and DLL calls.

And the good news is that most of Windows is available to you, from simple Word for Windows macros.

Little ol', much-maligned WordBasic has almost all of Windows at its beck and call.

Dynamic Link Libraries

Windows, as you may know, is nothing but a collection of intertwined subroutines, not unlike the TSR Terminate-and-Stay-Resident routines of a kinder and gentler age, rigged to bounce around and occasionally cooperate with one another.

Those subroutines just sit around in the WinPrimordialOoze, waiting to be called and used by Windows programs. They have names like USER and KERNEL and GDI.

They're brought into the Windows picture as they're required — dynamically — and returned to their dank cells when no longer needed. Thus, they're "dynamically linked".

At heart a "Dynamic Link Library" is just a subroutine, or bunch of subroutines. Windows is composed of DLLs. So are most Winapps.

You can even make your own DLLs, should you be of sufficiently masochistic bent, but (alas!) not with WordBasic or Visual Basic.

Your WordBasic programs — and, by implication, your WinWord documents, custom WinWord systems and applications — can take advantage of the Windows Dynamic Link Libraries. They can also hook into home-grown DLLs. It's an incredibly powerful capability, one that we're (at least I'm!) only beginning to understand.

API

If you want to call a DLL, you have to know the name of the subroutine, what to feed it, what it will return, right? Pretty hard to call it otherwise.

That whole shtick — names of routines, what they consume, what they regurgitate — is the "Application Programming Interface", or API. Back in the dark ages, some two or three years ago, computer jocks used to refer to this kinda stuff as "subroutine calling conventions", but "API" sounds cooler, like you know what you're talking about, eh?

Don't get intimidated by the TLAs. A DLL is just a bunch of subroutines. An API is just the subroutines' calling conventions. It's really that simple.

I'm often amazed at how the books treat Windows API calls, as if they're really difficult, or shrouded in mystery. That's a crock. In fact, if you find the right subroutine, and can decipher the computerese gobbledygook, using API calls can be much, much simpler than programming from scratch.

The key to it all is the WordBasic Declare statement.

Stop! Watch!

Here's an easy, fun example. I hit this while working on the *Hacker's Guide*. Say you want to find out how long it takes for a WordBasic loop to run, to see if you can speed things up by tweaking stuff. (Or you may want to keep track of how much time it takes the user to answer a question, or pull in a file from a network server. Whatever.)

The Wrong Way

My first temptation was to use the WordBasic Time$() function, with a timer that looks something like this:

```
Sub MAIN
StartTime$ = Time$()
For i = 1 to 1000
Dummy = Dummy + 1
Next i
EndTime$ = Time$()
MsgBox "Start:" + StartTime$ + " End:" + EndTime$
End Sub
```

If you try to run that little bit of WordBasic (and if you have the same settings I have), you'll get the following message box. Real informative, huh?

Okay, hey, I lied. I never *really* thought about using Time$(). It's a terribly inflexible function, bound at the wrists and ankles to the user's Windows Control Panel "International" settings, programmatically inscrutable. But

what the hay — it makes a better story.

A Windows Way

It just so happens that Windows maintains its own clock, ripe for your picking. And it ticks off milliseconds, accurate to within 55 milliseconds. It doesn't stick a colon in the middle of the number, or plant an "AM" at the end. It simply ticks, telling you how many milliseconds Windows has been running. All you need to do is *get at it*.

Ah, no sweat. If you look in *Mere Mortals* — or any of the "C" or Visual Basic Windows reference books — you'll find the Windows DLL ("USER") containing the timer function; you'll find the name of the function ("GetCurrentTime"); and you'll find what parameters it takes (none), and how it returns the time (as something called a Long Integer).

A nip here and a tuck there, and the program is done.

```
Declare Function GetCurrentTime Lib "User" As Long

Sub MAIN
StartTime = GetCurrentTime
For i = 1 to 1000
Dummy = Dummy + 1
Next i
EndTime = GetCurrentTime
MsgBox "Elapsed Milliseconds:" + Str$(EndTime - StartTime)
End Sub
```

Finally, something you can work with:

It's that simple.

Say, Wuh?

Did that go by too fast? Okay. Here's the slow-mo version.

Dissection

The trick to any DLL call — any intrepid subroutine (or function) call from WordBasic that dares to reach into the WinPrimordialOoze — is with the WordBasic Declare statement. That statement has five responsibilities.

First, it must tell WinWord whether you're looking at a subroutine or a

*Appearances to the contrary, this is **not** a portrait of the legendary WordBasic hack Scott Krueger.*

function. That's easy: a function returns a value, a subroutine doesn't. This guy returns a value (else, how would you know the time?), so the Declare statement starts out like this:

```
Declare Function
```

Next, the Declare statement must tell WinWord the name of the subroutine or function. We cheated and looked it up on page 466 of *Mere Mortals*, but you could just as easily have garnered the function's name from any of dozens of reference books for the Windows API.

```
Declare Function GetCurrentTime
```

If the function GetCurrentTime required any parameters, they would go immediately after the function's name. (You'll see an example shortly.) But GetCurrentTime doesn't need any parameters, so we can ignore that part.

Next, we have to tell WinWord where to find GetCurrentTime, the name of GetCurrentTime's DLL. Again, we sneaked a peek at *Mortals*, and found out that it lives in a Windows library called USER. You can usually find the DLL's name in whichever reference book provided you with the function's name. (Surprisingly, though, the Windows SDK documentation — the expensive, "official" documentation — for some reason assumes you *already know* where GetCurrentTime lives. Bizarre!)

```
Declare Function GetCurrentTime Lib "User"
```

Finally, the function returns a variable, so we need to tell WinWord what kind of variable to expect. Windows has all sorts of variable types, from simple integers to structured data types with pointers, and a whole bunch of things in between (there's a list on page 125 of *Mere Mortals*).

Unfortunately, when WordBasic interacts with Windows it's only capable of understanding four data types: character string; integer; long (32 bit) integer; and real (32 bit) numbers. That restriction to four data types causes no end of havoc; in fact, this amounts to the single most frustrating, damning limitation in WordBasic API calls.

Putting it all together:

```
Declare Function GetCurrentTime Lib "User" As Long
```

There are some significant subtleties in the way WordBasic handles the Windows variable types, and conversion to and from WordBasic variables. Even the character string variable type, which would seem at first blush to be pretty straightforward, can get a bit confusing.

Redux

So there you have it. Windows on the half-shell.

WordBasic has a couple of rather severe limitations when it comes to banging against DLLs in general, and the Windows API in particular. We touched on

the variable type discrepancy above. In addition, there's no way I've found to implement a general Windows "call back" capability in WordBasic.

(Don't let the bafflegab flummox you. There's nothing mysterious about "call back"s. In many programming languages you can tell Windows — or any DLL — "Yo! Run this subroutine. When you're done, tap me on the shoulder and tell me you're through." Windows taps your program on the shoulder by running whichever routine you specify. That's the "call back" routine.)

Several of the WordBestAndBrightest — Scott Krueger, Vince Chen and Guy Gallo among them — have tried to get at Windows calls that return structured data, typically with pointers. (The Windows COMMDLG Common Dialog DLL falls into this category.) So far I don't know of any general success. But don't hold your breath. By the time you read this, somebody may have found The Way.

There's only one rule, really, when dealing with Windows API calls: back up well and back up often. *(That's two rules, huh?)* One little slip of the finger, one slight misinterpretation of the C-centric Windows documentation, one teensy collision with an undocumented feature, can send your WordBasic program... WinWord... even Windows itself off into never-never land. You should back up *every single time* you run a Declare statement.

Don't be timid, though. Bang away. You aren't going to break anything. If you can't get a specific DLL call to work the way you think it should, try a few variations. Experiment!

Gladly then The Way receives
Those who choose to walk in it;
Gladly too its power upholds
Those who choose to use it well.
— Lao Tzu, *Tao Tê Ching*, 6th century BC

Private INI Files

Let's dive into the Windows API with both feet, and play around with some of the most important, most useful Windows API calls: the ones that let you build your own .INI files.

.INI files store parameters; you can think of 'em as "initialization values" or "default values" if you like. In fact, an .INI file can hold almost anything. The beauty of .INI files is that Windows will *maintain them for you* — do all the work.

Rationale

WordBasic comes with two commands that let you muck around with WIN.INI, the mother of all Windows .INI files. They're the two commands called GetProfileString$ and SetProfileString.

Those commands work just fine. In fact, they work so well that you can completely lock up Windows — lock 'er up so tight that it can't even be fixed by re-booting — with just one bollixed WordBasic command. True fact.

I'm vehemently opposed to using WIN.INI in all but the most dire circumstances. (Yes, it's true: sometimes you can't avoid it.) Take a look at your WIN.INI some day and you'll see why: every Tom, Dick and Hairless application shovels its garbage into WIN.INI, and once something goes into WIN.INI, it almost never comes out.

I'm fond of saying that your users should tar and feather you, should you use their WIN.INI — and I'll provide the feathers.

Just because WordBasic gives you an AK-47, doesn't mean you have to test it in rush hour traffic.

If you need to store parameters, you should learn how to create and use your own .INI file: a private .INI file. It's quite simple, really.

The INI Thingie

If you look inside almost any .INI file (that is to say, any .INI file except WINWORD.INI, which is a bizarre mess), you'll find entries, grouped into sections, that look like this:

```
[Enh386]
WinTimeSlice=100,50
display=orchidf.386
```

In general, .INI files are composed of one or more sections, each section containing one or more variable/value pairs, thusly:

```
[Section Name]
VariableName=Value
```

(The official terminology for all this stuff is ludicrous. If you find yourself

> *On the other hand, this **is** a portrait of Lee Hudspeth. Lee's sidekick Jim Lee (the other half of PRIME Consulting) pioneered the use of private INI files from within WordBasic.*

wading through the official Windows documentation, trying to sort it all out, you'll hit at least three different naming conventions. Judicious use of hip waders is highly recommended.)

There are two things you normally want to do to an INI file: first, you will want to set up variables, assigning values to them; second, you'll want to retrieve previously assigned values. *Rocket science, eh?* It's amazingly easy to do both, if you use the API calls built into Windows.

WritePrivateProfileString

Let's say you want to set up your own INI file, called, oh, TECHED.INI. Within TECHED.INI, you want to establish a section called [WinWord], and in that section you want to assign MyDummyVariable=Yes, Suh!.

Nuthin' to it.

```
Declare Function WritePrivateProfileString Lib "Kernel" \
        (Section$, VariableName$, VariableValue$, INIFileName$)\
        As Integer
Sub MAIN
n = WritePrivateProfileString("WinWord", "MyDummyVariable", \
        "Yes, Suh!", "TECHED.INI")
End Sub
```

It's really that simple! If you run that little program, then look at the file TECHED.INI in your Windows directory, it'll look like this:

```
[WinWord]
MyDummyVariable=Yes, Suh!
```

Windows takes care of all the details. It knows that the private .INI file should be stored in the Windows directory. If there's no file called TECHED.INI in the Windows directory, one is created for you. If there's no section called [WinWord], one is created for you. If there's no MyDummyVariable, it's created for you, too. Finally, MyDummyVariable is assigned the value Yes, Suh! (yep, Windows is smart enough not to stumble over the comma and the space).

GetPrivateProfileString

Once a variable's value has been established — with a WordBasic call like the one above, or even if you've set the INI file entry by hand — retrieving the value is so easy it's embarrassing.

```
Declare Function GetPrivateProfileString Lib "Kernel" \
        (Section$, VariableName$, DefaultValue$, \
        ReturnedVariableValue$, MaxChars As Integer, \
        INIFileName$) As Integer
Sub MAIN
n = GetPrivateProfileString("WinWord", "MyDummyVariable", \
        "OOPS!", VariableValue$, 255, "TECHED.INI")
MsgBox VariableValue$, "Returned" + Str$(n) + " Characters"
```

```
End Sub
```

For those of us accustomed to the (relative) sanity of Basic, this command has a very confusing syntax: it's all bass-ackward. The function itself returns the number of characters returned by the function, in the indicated variable, if you knowwhatImean. (Try thinking *that* ten times, fast!) But once you succumb to the inscrutable C weirding way, the actual API call is like falling off a log.

True Confessions

No, I don't keep these strange Windows API calls in my head. I look 'em up. You probably will, too. The question is: where?

You can go to the source of all WinWisdom, the *Programmer's Reference* in the Windows Software Development Kit. Three problems: the SDK costs hundreds of bucks; the *Programmer's Reference* is around 2,000 pages and growing almost daily; and, if you manage to find the command you want, it could take days to wade through the C-speak gobbledygook.

Windows 3.1 Programming for Mere Mortals has a hundred or so API calls, all worked through and documented for WordBasic. But it's far from exhaustive.

If you're willing to endure some pain, you may find the Visual Basic 2.0 Professional Developer's Kit of some use. It contains a very complete listing of Windows API calls, translated for use in Visual Basic. While the Visual Basic syntax isn't identical to WordBasic — not by a long shot — it's nonetheless far more scrutable than C-speak, and thus a good starting point.

The usual admonitions apply: back up often, back up well. And batter the living daylights out of the API calls. It's a lot of fun out there on the bleedin' edge!

Advanced Topics

I wanted to let you know about four particularly important Windows API calls. These particular calls solve WordBasic problems I've encountered over and over again. And, for the life of me, I don't know of any way to solve the problems *without* API calls.

Zapping Out an INI Variable

If you maintain private INI files, *pace* the previous section, you will often find yourself in the position of wanting to delete variables from the INI file.

It's easy if you know the trick. (This is from *Windows 3.1 Programming for Mere Mortals*, page 468.) The Windows documentation will tell you that you can delete a variable — delete a line in the INI file — if you feed Windows a "null pointer" for the variable's value.

It ends up that the easiest way to do that is to redefine the standard Windows WritePrivateProfileString, and feed it a long integer zero. Like this:

```
Declare Function WipeOutVar Lib "Kernel" \
        (Section$, VariableName$, Pointer As Long,INIFileName$) \
        As Integer Alias "WritePrivateProfileString"
Sub MAIN
n = WipeOutVar("WinWord", "MyDummyVariable", 0, "TECHED.INI")
End Sub
```

If you run that little program, Windows will delete the line

```
MyDummyVariable=
```

from the [WinWord] section of TECHED.INI.

Retrieving all Variable Names

This is a far more complex series of API calls, a solution that appeared just after the *Hacker's Guide to Word for Windows* went to press. Vince Chen figured out how to do it.

The Windows documentation tells you that retrieving a list of all variable names within a given section is easy: use GetPrivateProfileString, but send a "null pointer" for the variable name.

There's just one teensy-tiny problem. For some reason, WordBasic won't take the string that Windows wants to send back. We tried for weeks and weeks to figure out why WordBasic was so uncooperative; to this day, I don't know why it doesn't work the easy way.

That's when Vince got to hackin' at it. He discovered that if you reached into the bowels of Windows and *allocated your own space* for the answer, then asked politely for the variable names, Windows and WordBasic could get their act together.

Translating that concept into working code is another problem altogether, of course, but this works....

```
Declare Function GetPrivateProfileString Lib "Kernel" \
        (Section$, Pointer As Long, DefaultValue$, \
        ReturnedValueLocation As Long, MaxChars As Integer,\
        INIFileName$) As Integer
Declare Function GlobalAlloc Lib "Kernel"(wFlags As Integer, \
        dwBytes As Long) As Integer
Declare Function GlobalFree Lib "Kernel"(hMem As Integer) \
        As Integer
Declare Function GlobalLock Lib "Kernel"(hMem As Integer) \
        As Long
Declare Function GlobalUnLock Lib "Kernel"(hMem As Integer) \
        As Integer

Declare Function Stripper Lib "Kernel"(InString$, lp As Long) \
        As Long Alias "lstrcpy"
```

```
Sub MAIN
REM Allocate 4096 moveable bytes, zeroed, and lock it
hlpVarNames = GlobalAlloc(2 + 64, 4096)
REM lpVarNames is a long pointer to the string of Var Names
lpVarNames = GlobalLock(hlpVarNames)

REM Retrieve the zero-delimited string of Variable Names
n = GetPrivateProfileString("WinWord", 0, "", lpVarNames, \
        4096,  "TECHED.INI")

REM Bump over the Chr$(0) delimiters
n = Stripper(VarName$, lpVarNames)
While Len(VarName$) > 0
        MsgBox VarName$, "Variable Name"
        lpVarNames = lpVarNames + Len(VarName$) + 1
        n = Stripper(VarName$, lpVarNames)
Wend

n = GlobalUnlock(hlpVarNames)
n = GlobalFree(hlpVarName)
End Sub
```

That little While loop uses the Windows function lstrcpy to translate a pointer into a string. The funny bump-and-jump machinations are necessary because Windows stores the variable name string in zero-delimited format. WordBasic has a hard time grokking zero-delimited strings. (There's a full discussion in *Mere Mortals*, pp 149-151.) Vince's solution here is a much more elegant one than the one I stuck in the book.

Echo — One More Time, With Feeling

We've been trying to get a reliable Echo Off in WordBasic for as long as there's been a WordBasic, it seems. The problem is simple: how to tell Windows, "Hey, stop updating the screen while I go off and do something!" In a complex WordBasic program, the lousy screen updates can slow down your code *by a factor of ten*.

Once again, Vince found an extraordinarily elegant solution.

```
Declare Function GetActiveWindow Lib "User" As Integer
Declare Function LockWindowUpdate Lib "User"(hWnd As Integer) \
        As Integer

Sub Echo (n)
    Select Case n
      Case 0  ' Echo OFF
        currWindow = GetActiveWindow
        r = LockWindowUpdate(currWindow)
      Case Else  ' Echo ON
        r = LockWindowUpdate(0)
    End Select
End Sub
```

Guy Gallo discovered that if you use the Windows API call GetFocus instead of GetActiveWindow, screen updates for all of WinWord will be suspended.

If you write complicated WordBasic stuff for lots of people and can figure out how to make this Echo Off work, this one routine should just about pay for the whole Tech•Ed conference.

Directory Listing

WordBasic has no easy way to retrieve a list of currently valid subdirectories. We jimmied together a couple of alternatives and put them in the Hacker's Guide, but while we weren't looking Hackus Maximus Scott Krueger came up with a Windows end-run.

This is a great bit of Windows API programming from within WordBasic. First time I saw it, I didn't believe it could be done from inside WordBasic, but sure enough....

```
Declare Function GetFocus Lib "User"() As Integer
Declare Function CreateWindow Lib "User"(lpClassname$, \
        lpWindowName$, dwStyle As Long, X As Integer, \
        Y As Integer, nWidth As Integer, nHeight As Integer,\
        hWndParent As Integer, hMenu As Integer, \
        hInstance As Integer, lpParam$) As Integer
Declare Function DestroyWindow Lib "User"(hWnd As Integer) \
        As Integer
Declare Function SendMessageString Lib "user"(hWnd As \
        Integer, wMsg As Integer, wParam As Integer, lParam$) \
        As Integer Alias "SendMessage"
Declare Function SendMessage Lib "user"(hWnd As Integer, \
        wMsg As Integer, wParam As Integer, lParam As Long) \
        As Integer

Sub MAIN
REM ***Get Subdirectory List - 10/12/92 - by Scott Krueger
DirSpec$ = "*.doc"
FileSpec$ = "*.*"
hWnd = GetFocus
ListBoxhWnd = CreateWindow("ListBox", "GetDirTempListBox",\
        1084227715, 10, 10, 100, 100, hWnd, CtrlID, hInstance,\
        lpParam$)
REM*** use (-32752) to get subDirectories, 0 to get Files,
REM*** (-16368) for subdir and drives
nFileSpec = - 16368
results = SendMessageString(ListBoxhWnd, 1038, nFileSpec, \
        FileSpec$)
count = SendMessage(ListBoxhWnd, 1036, 0, 0)
Dim Dir$(count)
For XX = 0 To count - 1
    Dir$(XX) = String$(255, Chr$(0))
    worked = SendMessageString(listBoxhWnd, 1034, XX, \
        Dir$(XX))
Next XX
Killed = DestroyWindow(ListBoxhWnd)
```

```
Begin Dialog UserDialog 320, 122, "Microsoft Word"
     ComboBox 10, 6, 160, 108, Dir$(), .ComboBox1
     OKButton 205, 7, 100, 21
     CancelButton 205, 31, 99, 21
End Dialog
Dim dlg As UserDialog
KeyClick = Dialog(dlg)
End Sub
```

Run that program and you'll get a ComboBox with all available subdirectories. It looks like this:

Once again, Scott has shown the way with another bit of SuperHack. If you haven't seen his Kdlg replacement for WinWord dialogs, check it out: there are copies available on the CompuServe PROGMSA forum, and on the companion disk to the *Hacker's Guide*.

Staying Connected

As you can see, there's an enormous amount of work going on with WordBasic. It's being extended in ways its originators wouldn't've ever dreamed.... not in their worst nightmares.

How to find out about all these new goodies? Easy. Get on line!

The best way to get a straight answer to your questions — from just-getting-started novice to grizzled guru — is in the Microsoft fora on CompuServe. If you aren't on-line, you're missing the greatest resource of WinWord help in existence.

General WinWord questions are handled on the Word forum: just type GO MSWORD. Questions specific to WordBasic and how it interacts with the outside world (including Windows API calls and DDE) run through the "Programming Microsoft Applications" forum. Type GO PROGMSA.

See you on-line!

Woody Leonhard
Coal Creek Canyon, Colorado
January 15, 1992