Going Ape Over the Windows API

Doug Hennig

The Windows API (Win32API) contains thousands of useful functions. However, finding which function to use when, and how to call it from a VFP application, can be challenging. This month's article presents several useful API functions you can call in your VFP applications today.

Windows consists of several thousand functions that provide every service imaginable, from graphics drawing functions to network access functions. These functions are often referred to as the Windows 32-bit Application Program Interface, or Win32API for short. Every Windows application calls these functions to interact with the system hardware and present the user interface. In addition, programming languages like VFP often provide commands or functions that are merely wrappers for the appropriate Win32API functions. For example, the VFP MESSAGEBOX() function calls the Win32API MessageBox function, and DISKSPACE() calls GetDiskFreeSpace. However, even a rich language like VFP exposes only a small fraction of the total Win32API to developers. So, accessing Windows services often requires direct access to the Win32API. Examples of the kinds of services not available (or only partially available) natively in VFP but through the Win32API include serial communications, email, communication via the Internet (HTTP, FTP, etc.), network functionality, and file and operating system information.

This month's article discusses several Win32API functions I find useful in the applications and tools I develop. I'm not going to discuss the exact syntax for declaring or calling a Win32API function; see the DECLARE – DLL topic in the VFP help file for details. Also, due to space limitations, I won't show the complete code for the examples accompanying this article; instead, we'll just look at how the specific Win32API functions are declared and typically used.

Although you can declare and call Win32API functions wherever you wish, they're a lot easier to use if you create wrapper routines. These routines, whether in PRGs or methods of class libraries, make it easier to use these functions because, once written, you don't have to remember the exact name and case of the function (unlike VFP, Win32API function names are case-sensitive), what order the parameters are in, which ones are passed by value and which by reference, etc. They can also assign default values to seldom-used parameters and provide more complete documentation about how and when the function should be used. All of the functions we'll look at in this article have wrapper programs included in the Subscriber Downloads for this month.

I should point out that many of the more useful functions in the Win32API are now available in the Windows Script Host (WSH). The WSH is far simpler to work with than the Win32API because it's a set of COM objects; there's no worrying about function declarations, the type of parameters required, or handling structures and other data types VFP doesn't support directly. The only downside of using the WSH is that you can't necessarily be sure it's installed on a user's system, since it didn't come with all versions of Windows. You can install it as part of your application installation, but there are some issues to consider that are beyond the scope of this article. For more information on the WSH, see http://msdn.microsoft.com/scripting or a great set of articles by Ed Rauh and George Tasker at http://www.vfug.org/.

Getting the Directory the Application is Running In

Frequently, an application needs to know what directory it's running in (its "home" directory) because it must access resources such as tables, INI files, graphics files, etc. from that directory. You can't assume the home directory is the current one; the user could have entered a different directory in the "start in" setting for the shortcut to the application, for example. Determining the home directory isn't straightforward; the method depends on what's running: an in-process DLL, an out-of-process EXE server, or a standalone EXE.

GetAppDir.prg can be used to determine this. You can either call it with no parameters (in which case it looks at the top-level program to see where it's running from) or pass it something like SYS(16), which is the name and path of the currently running program. It returns the directory the program is running from with a trailing backslash. GetAppDir.prg uses the GetModuleFileName API function. This function fills a

buffer with the name and path of the currently executing EXE. Here's how the GetAppDir.prg calls this function:

```
#define MAX_PATH 260
declare integer GetModuleFileName in Win32API ;
    integer hModule, string @cFileName, integer @nSize
lnBytes    = MAX_PATH
lcFileName = space(lnBytes)
GetModuleFileName(0, @lcFilename, @lnBytes)
lcFileName = left(lcFileName, at(chr(0), lcFileName) - 1)
```

Getting Windows Directories

GetTempDir.prg is useful when you need to create temporary files and want them to go in the Windows temporary files directory. This might give better performance if the application is running from a server but the temp directory is on a local drive, or might make cleaning up wayward temporary files easier. It uses the GetTempPath API function. This function is simple: it just fills a buffer with the name of the Windows temp directory:

```
#define MAX_PATH 260
lcBuffer = space(MAX_PATH)
declare integer GetTempPath in Win32API ;
    integer nLength, string @szBuffer
GetTempPath(MAX_PATH, @lcBuffer)
return addbs(left(lcBuffer, at(chr(0), lcBuffer) - 1))
```

GetWindowsDir.prg looks almost identical to GetTempDir.prg because the GetWindowsDirectory API function is almost identical to the GetTempPath function, except it fills a buffer with the name of the Windows directory and the order of the parameters is different.

```
#define MAX_PATH 260
lcBuffer = space(MAX_PATH)
declare integer GetWindowsDirectory in Win32API ;
    string @szBuffer, integer nLength
GetWindowsDirectory(@lcBuffer, MAX_PATH)
return addbs(left(lcBuffer, at(chr(0), lcBuffer) - 1))
```

Along these same lines, GetSystemDir.prg uses the GetSystemDirectory API function to get the name of the Windows system directory. This function is useful, for example, to determine if an ActiveX control has been installed on a user's system, since typically they're installed in the Windows system directory.

```
#define MAX_PATH 260
lcBuffer = space(MAX_PATH)
declare integer GetSystemDirectory in Win32API ;
    string @szBuffer, integer nLength
GetSystemDirectory(@lcBuffer, MAX_PATH)
return addbs(left(lcBuffer, at(chr(0), lcBuffer) - 1))
```

Determining the Windows desktop folder, which is useful if you want to create a shortcut, is difficult to do using straight VFP code. To make this process simpler, George Tasker created LNKFILES.DLL (in LNKFILES.ZIP). Its FindDesktopFolder function fills a buffer with the Windows desktop directory. GetDesktopDir.prg is a wrapper for this function. We'll see in a moment how to use this DLL to actually create a shortcut.

```
#define MAX_PATH 260
declare integer FindDesktopFolder in lnkfiles.dll ;
    integer hOwner, string @lpbuffer
lcBuffer = space(MAX_PATH)
lnLength = FindDesktopFolder(0, @lcBuffer)
return addbs(left(lcBuffer, lnLength))
```

TestWinDirectories.prg shows an example of calling all of these functions.

Creating a Shortcut

If you use the VFP Setup Wizard, you're probably aware of one shortcoming it has over other installers (well, it has a lot more than one shortcoming, but we're only concerned with a certain one here <g>): it won't create a desktop shortcut for the application. Fortunately, it does support running a post-setup executable, so you can create an EXE that runs after the setup has completed to do this task.

Unfortunately, there's no built-in VFP function or even an API function to do this; instead, you need to implement the IShellLink COM interface and do it that way, which is very difficult in VFP. The good news is that George Tasker has done all the hard work for us with his LNKFILES.DLL. You just call the CreateShellLink function in the DLL to create the shortcut and optionally call SetLinkWorkDir to set the working directory for the shortcut.

```
declare integer CreateShellLink in lnkfiles.dll ;
    string @lpzLinkFileName, string @lpzExeFileName
declare integer SetLinkWorkDir in lnkfiles.dll ;
    integer hWnd, string @lpszLinkName, string @lpszPath
lcDirectory = justpath(tcFileToLink)
lnResult = CreateShellLink(@tcLinkFile, @tcFileToLink)
lnResult = SetLinkWorkDir(0, @tcLinkFile, @lcDirectory)
```

CreateShortcut.prg is a wrapper for George's DLL; just pass it the name of the shortcut file to create (including the path, which is normally the Windows desktop directory; call GetDesktopDir.prg to get that location) and the fully-qualified name of the file the shortcut links to. CreateShortcut returns .T. if the shortcut was successfully created. Because the most common place to create a shortcut is on the desktop, I created CreateDesktopShortcut.prg so I don't have to remember to add the location of the Windows desktop directory to the name of the link path. Run TestCreateShortcut.prg for an example.

Determining if a Disk is in the Drive

IsDiskIn.prg returns .T. if a formatted disk is in the specified drive. This can be used, for example, just prior to writing to a floppy disk to ensure an error won't occur if there's no disk in the drive. This PRG uses the fact that the VFP DISKSPACE() function returns -1 if there's an error reading the specified drive. However, under some operating systems, an untrappable Windows error occurs in this case. To prevent this from happening, the SetErrorMode API function is used to turn this off. SetErrorMode sets the Windows error mode to the specified value and returns the former value, so simply saving the former value and passing it to SetErrorMode later makes it easy to clean up.

```
declare integer SetErrorMode in Win32API ;
    integer nErrorMode
lnWinErrMode = SetErrorMode(1)
llCanRead    = diskspace(tcDrive) <> -1
SetErrorMode(lnWinErrMode)
```

Determining if a Drive Exists

If you need to determine if a drive exists, DriveExists.prg is the routine for you. It uses the GetLogicalDrives API function, which returns a bitmapped value of available drives. For example, if drive A is available, bit 0 is set; bit 1 is set if drive B is available. DriveExists.prg expects to be passed a drive letter, and returns .T. if the appropriate bit in the value returned by GetLogicalDrives is set. Run TestDriveExists.prg to see how it works.

Here's the applicable code calling GetLogicalDrives:

```
declare integer GetLogicalDrives in Win32API
lnDrives = GetLogicalDrives()
lnDrive  = asc(upper(left(tcDrive, 1))) - asc('A')
return bittest(lnDrives, lnDrive)
```

Determining Drive Space

In my February 2001 column, I presented a routine called GetDriveSpace.prg that overcomes a bug in the DISKSPACE() function in VFP 6 and previous versions (this bug is fixed in VFP 7): it returns incorrect values when there's more than 2 GB of free space. GetDriveSpace.prg uses the GetDiskFreeSpaceEx API

function to return the correct value. However, there's one problem: this function isn't available in early versions of Windows 95. So, I used the GetVersionEx API function to determine which version of the operating system was running so I could know whether to call GetDiskFreeSpaceEx or not.

Since then, George Tasker pointed out to me a better way to determine if an API function is available or not. First, call GetModuleHandle to get a handle to the DLL containing the function (KERNEL32.DLL in this case). Then call GetProcAddress to find the address of the desired function; if it returns 0, the function doesn't exist. Note that in this case, the name of the function passed to GetProcAddress is actually GetDiskFreeSpaceExA, because that's the actual name as it exists in the DLL. When you declare an API function in VFP, if the specified name doesn't exist, VFP automatically adds "A" to the end and searches for that name.

```
declare integer GetModuleHandle in Win32API ;
    string @lpModuleName
declare integer GetProcAddress in Win32API ;
    integer hModule, string @lpProcName
lcDLL      = 'kernel32.dll'
lcFunction = 'GetDiskFreeSpaceExA'
lnHandle   = GetModuleHandle(@lcDLL)
llExists   = GetProcAddress(lnHandle, @lcFunction) > 0
```

The Subscriber Downloads for this month includes a revised GetDriveSpace.prg.

Getting the User's Login Name

The WNetGetUser API function gives you the current user's login name. This can be used, for example, in applications where you need to know who the user is (perhaps each user has their own configuration settings stored in a table indexed by user name) but you don't want to ask them to log in. GetUserName.prg is a wrapper for this function and returns the user's name. Run TestGetUserName.prg to test it.

```
lcName = chr(0)
lnSize = 64
lcUser = replicate(lcName, lnSize)
declare integer WNetGetUser in Win32API ;
    string @cName, string @cUser, integer @nBufferSize
if WNetGetUser(@lcName, @lcUser, @lnSize) = 0
    lcUser = left(lcUser, at(chr(0), lcUser) - 1)
endif WNetGetUser(@lcName, ...
```

Getting the Network Name for a Mapped Device

The WNetGetConnection API function gives you the network name for a mapped device. "Device" could be a drive or printer. For example, if T: is mapped to a network volume, this function can tell you the server and volume names it's mapped to. This function accepts three parameters: the name of the mapped device, a buffer for the remote name, and the length of the buffer. It returns 0 if it was successful or an error code if not (see WIN32API.TXT, described in "Other Functions" at the end of this article, for error codes defined as constants starting with "ERROR"); for example, 2250 means the device specified wasn't mapped to anything. GetNetConnection.prg is a wrapper for this function.

```
lnLength = 261
lcBuffer = replicate(chr(0), lnLength)
declare integer WNetGetConnection in Win32API ;
    string @lpLocalName, string @lpRemoteName, ;
    integer @lpnLength
if WNetGetConnection(tcDevice, @lcBuffer, @lnLength) = 0
    lcReturn = left(lcBuffer, at(chr(0), lcBuffer) - 1)
endif WNetGetConnection(tcDevice, ...
```

Attaching to a Network Resource

The WNetAddConnection API function maps a local device to a network resource. Pass it the name of the network resource, the password for the current user to connect to the resource (an empty string implies the user is already validated), and the name of the local device (which shouldn't be already mapped to anything;

you can use DriveExists.prg, discussed earlier, to ensure this). It returns 0 if it was successful or an error code if not. AddNetConnection.prg is a wrapper for this function.

```
declare integer WNetAddConnection in Win32API ;
    string lpRemoteName, string lpPassword, ;
    string lpLocalName
llReturn = WNetAddConnection(tcResource, tcPassword, ;
    tcDevice) = 0
```

Disconnecting from a Network Resource

The WNetCancelConnection API function disconnects a mapped device from its network resource. Why not just stay permanently connected? The server may not always be available (such as with a WAN connection over dialup), or you may have a limited number of network licenses available so you want to minimize concurrent connections. WNetCancelConnection accepts two parameters: the name of the local device to disconnect and a numeric parameter indicating whether Windows should close any open files or return an error. It returns 0 if it was successful or an error code if not. CancelNetConnection.prg is a wrapper for this function.

```
declare integer WNetCancelConnection in Win32API ;
    string lpLocalName, integer lpnForce
llReturn = WNetCancelConnection(tcDevice, 0) = 0
```

Playing the Windows Error Sound

?? CHR(7) plays the Windows "default beep" sound, but you can SET BELL TO a WAV file to play that sound instead. However, what if you want to play the Windows "critical stop" or some other sound and don't know what the WAV file associated with that sound is? Also, there are some instances where ?? CHR(7) doesn't play any sound.

The solution to these issues is to use the MessageBeep API function. It accepts a numeric parameter for which sound to play. These values match the icon values used in the MESSAGEBOX() function (although a range of values will actually play the appropriate sound). For example, adding 16 to the icon parameter of MESSAGEBOX() (the MB_ICONSTOP constant in FOXPRO.H) plays the Windows "critical stop" sound when the dialog appear; that same sound plays when you pass a value from 16 to 31 to MessageBeep. Table 1 shows the range of values you can use.

Table 1. Values for MessageBeep.

Range	Sound	MESSAGEBOX() Constant
0 - 15	Default beep	none
16 - 31	Critical stop	MB_ICONSTOP
32 - 47	Question	MB_ICONQUESTION
48 - 63	Exclamation	MB_ICONEXCLAMATION
64 - 79	Asterisk	MB_ICONINFORMATION

Here's an example calling this function:

```
declare integer MessageBeep in Win32API ;
    integer wType
MessageBeep(16)
```

ErrorSound.prg is a simple wrapper that accepts the sound value to play (if it isn't passed, 16 is used, which plays the "critical stop" sound).

"Executing" a File

The Windows Explorer allows you to double-click on a file to "open" it. That causes the associated application for that file's extension to be executed and the file loaded in that application. You don't have to worry about what application to use or where it's located; Windows handles the details.

We can do the same thing in our applications using the ShellExecute API function. Pass this function the name of the file; if there's an application associated with that file, that application is started and the file loaded. This makes it easy, for example, to bring up the user's default browser to display an HTML file or

have Word or Excel print a file. In addition to the name of the file, you can also pass an "action" (for example, "Open" or "Print"), the working directory to use, parameters to pass to the application, and the window state to use. The FoxPro Foundation Classes (FFC) has a class called _ShellExecute with a ShellExecute method that calls the ShellExecute function, so you can use it as a wrapper.

Here's an example using this function:

```
declare integer ShellExecute in SHELL32.DLL ;
    integer nWinHandle, string cOperation, ;
    string cFileName, string cParameters, ;
    string cDirectory, integer nShowWindow
ShellExecute(0, 'Open', fullpath('MyFile.HTML'), '', ;
    '', 1)
```

Other Functions

We've just scratched the surface of the Win32API in this article. Here are some other functions I've used:

- My December 1998 column discussed the Registry class in the FFC Registry.vcx. This class uses API functions to read from and write to the Windows Registry.
- If you need to work with ODBC data sources, check out the ODBCReg class, also in the FFC Registry.vcx.
- The Solutions application that comes with VFP (in the SOLUTION subdirectory of the directory pointed to by _SAMPLES) has several examples, including how to play multimedia files from VFP.
- The shareware wwIPStuff, from West Wind Techologies (www.west-wind.com), uses API functions extensively to provide features such as sending email, uploading and downloading FTP files, and HTTP functionality in VFP applications. I've discussed it in a couple of previous articles: the September 2000 article discussed its FTP capabilities and in March 2000, we looked at using it to send email.
- My June 2000 column discussed several API functions useful for obtaining volume information, including the serial number.

Where can you find information on what other functions are available and how to use them? The resources mentioned below are by no means exhaustive but are some of the places I've found valuable information, tips, and even complete source code.

The MSDN Library has documentation on all API functions. It's written for C programmers, but is applicable to VFP developers as well.

Visual Studio comes with a tool called the API Viewer (APILoad.exe in the Common\Tools\WinAPI subdirectory of the Visual Studio directory) that can help with API functions. It doesn't, unfortunately, include any information about the purpose of each function or its parameters, but it does show the syntax in a more readable form than the MSDN Library. It also shows API constants and their values, and the layout of structures. To use it, choose Load from the File menu and select WIN32API.TXT from the same directory as the API Viewer.

WIN32API.TXT actually has more information that the API Viewer displays. In fact, I often find it more useful to open this file in a text editor because I can then read the full text (including comments) and do text searches, which the API Viewer doesn't support.

Karl Peterson, a well-known Visual Basic guru, has a lot of Win32API utilities on his Web site (www.mvps.org). He's tweaked WIN32API.TXT significantly, adding missing functions and more comments. He has a utility that identifies the meaning of Win32API error messages. He also has a lot of example code showing how various Win32API functions are used. The code is all in VB, of course, but can be readily converted to VFP. Another API-dedicated Web site is www.allapi.net.

The API and FAQ sections of the Universal Thread (www.universalthread.com) have a lot of documentation, samples, and utilities for the Win32API, and they're in VFP format too! In fact, most of the information I've gathered on Win32API functions I regularly use came from there.

Conclusion

The Win32API is a rich source of services that can extend the functionality of almost any application. It isn't difficult to learn how to use Win32API functions in VFP; the tough part is figuring out what function

to use for a given task, how to call that function, and what to do with the return values. Fortunately, this article discusses a number of commonly used functions and presents wrapper programs for them, making them easier to use.

Doug Hennig is a partner with Stonefield Systems Group Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT), co-author of "What's New in Visual FoxPro 7.0" from Hentzenwerke Publishing, and author of "The Visual FoxPro Data Dictionary" in Pinnacle Publishing's Pros Talk Visual FoxPro series. Doug has spoken at every Microsoft FoxPro Developers Conference (DevCon) since 1997 and at user groups and developer conferences all over North America. He is a Microsoft Most Valuable Professional (MVP) and Certified Professional (MCP). Web: www.stonefield.com Email: dhennig@stonefield.com