Session E-WAPI

Win32API for VFP Developers

Doug Hennig
Stonefield Software Inc.
Email: dhennig@stonefield.com
Corporate Web sites: www.stonefieldquery.com and www.stonefieldsoftware.com
Personal Web site: www.DougHennig.com
Blog: DougHennig.BlogSpot.com
Twitter: DougHennig

Overview

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 session discusses how API functions are called in VFP, where to find information on the API, and presents lots of useful API functions you can call in your VFP applications today.

Introduction

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; this is the case even in 64-bit versions of Windows. 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.

Win32API functions were written with C programmers in mind. The parameters they expect and values they return use C data types. Although most C data types map to VFP data types, as we'll see, many Win32API functions use data types not supported by VFP, so those functions either can't be used in VFP or require some special tricks. Also, the documentation for these functions often uses terms and concepts that may be foreign to VFP developers, especially those with limited experience with other languages.

This document discusses the Win32API from a VFP developer's perspective. We'll start by looking at how API functions are called from VFP, and some of the challenges their C orientation can cause us. Then, rather than looking at individual functions, we'll take a solution-oriented approach: we'll look at how Win32API functions can help us solve certain problems in VFP. Along the way, we'll learn techniques for dealing with different types of functions. Also, we'll look at wrapper programs for these functions so once we've written them, we don't have to remember how to call the functions in the future.

I don't claim to be a Windows API expert. Most of the functions I use came from ideas (and even code) generously shared by others, including Christof Wollenhaupt, Rick Strahl, Ed Rauh, George Tasker, and Erik Moore.

Using the Win32API

To use a Win32API function in VFP, you have to figure out how to tell VFP about it, how to call it, and what to do with the values it returns. Let's take a look at how we do that.

The DECLARE command

Before you can use a Win32API function, you have to register it to VFP using the DECLARE command. This overloaded command (a variation of it can be used as a synonym for DIMENSION) registers a function in a DLL for use in VFP. The DLL doesn't have to be part of the Windows operating system, but it's those DLLs we'll be focusing on in the document.

The syntax for the DECLARE command is as follows (clauses between square brackets are optional):

```
declare [FunctionType] FunctionName in LibraryName [as AliasName] ;
    [ParameterType1 [@][ParameterName1][, ;
    ParameterType2 [@][ParameterName2][, ...]]]
```

FunctionType is the data type for the return value of the function; we'll discuss data types in a moment. This clause is optional because a function may not return a value. FunctionName is the name of the function to define. Although VFP is a case-insensitive language, the function name is case-sensitive, so be sure to use the exact case specified in the documentation for the function or you'll get a "cannot find entry point" error (see BadDeclare.prg in the sample files accompanying this document for an example that triggers this error).

LibraryName is the name of the DLL containing the function. However, if you specify "Win32API", VFP automatically searches for the function in the following Windows DLLs: Kernel32.dll, Gdi32.dll, User32.dll, Mpr.dll, and Advapi32.dll.

Similar to how a table can be opened with an alias so it can be accessed with a different name, AliasName specifies an alternate name for the function. This may be used if the function has the same name as a native VFP function (for example, the Windows MessageBox function). For obvious reasons, don't specify a VFP reserved word or a DLL function name you've already registered.

Parameters passed to the function are specified in a comma-delimited list as the data type, followed by "@" if the parameter should be passed by reference rather than by value, and optionally a name that indicates the purpose of the parameter (VFP ignores the name you specify). Passing a variable by reference really means passing a pointer, which is the memory address for the variable's contents.

Data types, both for the function return value and parameters, are shown in **Table 1**. If the data type you specify doesn't match the data type expected by the function, you'll get an error. VFP will automatically convert numeric values to the appropriate size based on the declaration.

Table 1. Data types for Win32API functions.

Data Type	Description
SHORT	16-bit integer (return value only)
INTEGER	32-bit integer
LONG	32-bit integer
SINGLE	32-bit floating point
DOUBLE	64-bit floating point
STRING	Character string
OBJECT	COM object

A couple of issues to be aware of: You may sometimes encounter what appears to be other data types in the Win32API documentation. These are actually just synonyms for the data types in the table above. For example, DWORD ("double word") is an integer, as is HWND (and other "data types" beginning with "H"). Also, VFP integers are always signed, so you may have to convert the value returned by a function to an unsigned value.

Here's an example of a DECLARE statement:

```
declare integer WNetGetUser in Win32API ;
    string @cName, string @cUser, integer @nBufferSize
```

This could also be specified as:

```
declare long WNetGetUser in mpr.dll ;
    string @, string @, long @
```

WNetGetUser (with that exact case) is located in mpr.dll, but specifying "Win32API" is easier. It returns a 32-bit integer (so either "long" or "integer" can be used) and expects two strings passed by reference and a 32-bit integer by reference. The first example specifies parameter names, which makes it clearer what each parameter represents, but the second example shows that these names aren't actually required.

Calling an API function

Once you've registered the function, you call it just like a native function. Parameters passed by value can either be a variable, expression, or constant, but parameters passed by reference must be a variable preceded by "@".

Here's an example. The GetSystemDirectory function gives you the location of the Windows system directory. However, instead of providing it as the return value for the function, it expects you to pass the address of a string buffer (also known as a pointer to the buffer) where the value is placed and the length of that buffer. What does "the address of a string buffer" mean in VFP? Simple: pass a pre-initialized string (one filled with spaces or nulls) by reference. The length of the string in this case is defined by the Windows constant MAX_PATH, which is 260 bytes. Win32API functions frequently terminate strings with a null value, so upon return we'll take everything in the "buffer" up to the first CHR(0) as the value we're interested in.

```
local lcBuffer
#define MAX_PATH 260
```

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

This technique, initializing a string variable to a certain number of spaces or nulls, passing it and its length to a function, then upon return grabbing all the characters up to CHR(0), is one we'll see over and over in our examination of Win32API functions.

While many Win32API functions can easily be called from VFP, others cannot or require some tricky code. Some functions require more complex data types than the simple ones VFP supports. Structures, a user-defined data type that combines native data types in a certain sequence, are used in many functions. Here's an example: the declaration for the GetVersionEx function, which gets information about the operating system, is as follows (this was copied from MSDN, which we'll discuss later, so it isn't in VFP syntax but is readable nonetheless):

```
Public Declare Function GetVersionEx Lib "kernel32" ;
    Alias "GetVersionExA" ;
    (lpVersionInformation As OSVERSIONINFO) As Long
```

Most of that is straightforward, but what the heck is OSVERSIONINFO? It's a structure defined as follows.

```
Public Type OSVERSIONINFO
    dwOSVersionInfoSize As Long
    dwMajorVersion As Long
    dwMinorVersion As Long
    dwBuildNumber As Long
    dwPlatformId As Long
    szCSDVersion As String * 128
End Type
```

VFP doesn't support structures, but we can fake it. All variables are just bytes in memory, so we can use a string for a structure as long as we realize that we'll have to do some data conversion to put information into and get information out of the structure. So, instead of a parameter of type OSVERSIONINFO, we'll declare the function as accepting a string. In this case, the first four bytes contain the size of the entire structure, so we'll put the hex equivalent of 148 (5 longs x 4 bytes/long + 128 byte string = 148) into the first four bytes and initialize the rest of the string to nulls. The VFP BINTOC() function handles the conversion of a numeric value to its hex equivalent in a string. We'll then call the function and pass the string.

```
declare integer GetVersionEx in Win32API ;
    string @lpVersionInformation
lcStruct = bintoc(148, '4RS') + replicate(chr(0), 144)
lnResult = GetVersionEx(@lcStruct)
```

To get information back out of the structure, we have to convert the hex string to a numeric value using CTOBIN(). For example, the major and minor version numbers start at bytes 5 and 9, respectively (count bytes from the beginning of the string and match them up with the structure), so this code gets those values:

```
lnMajorVersion = ctobin(substr(lcStruct, 5, 4), '4RS')
lnMinorVersion = ctobin(substr(lcStruct, 9, 4), '4RS')
```

Instead of doing all this byte counting and conversion, you can use a public domain utility created by Christof Wollenhaupt called Struct (see STRUCT.ZIP in the included files). I won't go into details here about how to use this utility; it comes with documentation and several examples.

Other issues

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, 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 have wrapper programs in the source code files included with this document.

One trick some developers use is to create a program with the same name as an API function. For example, this code is in GetSystemMetrics.prg:

```
lparameters tnIndex
declare integer GetSystemMetrics in Win32API integer nIndex
return GetSystemMetrics(tnIndex)
```

The benefit of this approach is that there's no need to declare functions before they're used. For example, when the Windows API GetSystemMetrics function is needed, call GetSystemMetrics. The first time it's called, the PRG is executed, which declares GetSystemMetrics and then executes it. The second time GetSystemMetrics is used, the Windows API function is called instead of the PRG because of the order in which VFP looks for names.

The DISPLAY/LIST STATUS commands display the names of registered functions at the end of the status information. CLEAR ALL or CLEAR DLLS removes all registered functions from memory. CLEAR DLLS can accept a list of function aliases to clear from memory and the ADLLS() function fills an array with all registered functions.

Many of the more useful functions in the Win32API are 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. For more information on the WSH, see http://tinyurl.com/3z7a9wk.

Resources

So, where can we find information on what Win32API functions are available and how to use them? Here are a couple of 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.
- A great Web site for VFP developers is www.news2news.com/vfp/index.php. This site documents tons of API calls, and shows example code in VFP!

With all that behind us, let's get started looking at Win32API functions.

File and folder functions

Getting file names in the correct case

Many VFP developers use the FULLPATH() and SYS(2014) functions to obtain the full and relative path for a file name. Unfortunately, those functions return the name in upper-case regardless of the actual case of the name. That isn't a problem if the name is used for read access and isn't displayed to the user, but it is if the file is created or the name displayed to the user because all upper-case names is a sign of an antique application.

Instead, use the GetFullPathName and PathRelativePathTo API functions. These functions don't return the case of the name as it exists on disk, but as you specify it in the name passed to them.

GetFullPath.PRG is a wrapper for GetFullPathName. Pass it the file name to get the full path for, relative to the current directory.

```
#define cnMAX_PATH 260

lparameters tcName
local lcBuffer1, ;
    lcBuffer2, ;
    lnLen
declare long GetFullPathName in Win32API ;
    string lpFileName, long nBufferLength, string @lpBuffer, string @lpFilePart
store space(cnMAX_PATH) to lcBuffer1, lcBuffer2
lnLen = GetFullPathName(tcName, cnMAX_PATH, @lcBuffer1, @lcBuffer2)
return left(lcBuffer1, lnLen)
```

GetRelativePath.PRG is a wrapper for PathRelativePathTo. Pass it the file or directory name to get the relative path for and optionally the file or directory name to use as the relative path; if the latter isn't specified, the current directory is used. Note that this uses a slightly different technique for getting the return value from the buffer: converting all nulls to spaces and then using ALLTRIM() to remove extra spaces.

```
#define cnMAX_PATH                    260
#define cnFILE_ATTRIBUTE_DIRECTORY    0x10
#define cnFILE_ATTRIBUTE_NORMAL       0x80

lparameters tcTo, ;
    tcFrom
declare integer PathRelativePathTo in shlwapi.dll ;
    string @out, string @from, integer fromattrib, string @to, integer toattrib
lcPath       = space(cnMAX_PATH)
lcFrom       = iif(vartype(tcFrom) = 'C', tcFrom, sys(5) + curdir()) + chr(0)
lnFromAttrib = iif(directory(lcFrom), cnFILE_ATTRIBUTE_DIRECTORY, ;
    cnFILE_ATTRIBUTE_NORMAL)
lcTo         = iif(vartype(tcTo) = 'C', tcTo, sys(5) + curdir()) + chr(0)
lnToAttrib   = iif(directory(lcTo), cnFILE_ATTRIBUTE_DIRECTORY, ;
    cnFILE_ATTRIBUTE_NORMAL)
PathRelativePathTo(@lcPath, @lcFrom, lnFromAttrib, @lcTo, lnToAttrib)
lcPath = alltrim(strtran(lcPath, chr(0), ' '))
do case
    case empty(lcPath)
        lcPath = tcTo
    case left(lcPath, 2) = '.\'
        lcPath = substr(lcPath, 3)
endcase
return lcPath
```

To see how these functions work, run TestGetFullPath.PRG.

Getting system folders

It's not often that VFP applications need access to the Windows and System folders anymore, but if you do need to do this, you can use the GetWindowsDirectory and GetSystemDirectory API functions. You shouldn't assume that Windows is installed in C:\Windows and that the System folder is C:\Windows\System32 but instead use these functions to find the correct locations.

GetWindowsDir.PRG is a wrapper for GetWindowsDirectory:

```
#define MAX_PATH 260

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

GetSystemDir.PRG is a wrapper for GetSystemDirectory:

```
#define MAX_PATH 260

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

One complication with GetSystemDirectory: on a 64-bit system, it returns the 64-bit System directory rather than the 32-bit version (the oddly-named SysWOW64). Since VFP applications are 32-bit and can't access 64-bit applications, it's more likely you'll need to use SysWOW64. For that, use the GetSystemWow64Directory function. GetSystemWow64Dir.PRG is a wrapper for that function:

```
#define cnMAX_PATH 260

local lcBuffer
lcBuffer = space(cnMAX_PATH)
declare integer GetSystemWow64Directory in Win32API ;
    string @szBuffer, integer nLength
```

```
GetSystemWow64Directory(@lcBuffer, cnMAX_PATH)
return addbs(alltrim(left(lcBuffer, at(chr(0), lcBuffer) - 1)))
```

For example, one of my applications has a button that launches the ODBC Administrator so the user can add DSNs to their system if they wish. VFP applications can only use 32-bit DSNs, so you can't use the default ODBC Administrator in the System folder, ODBCAd32.EXE (which is misnamed since it's a 64-bit application) but instead the one in SysWow64. The following code finds the correct one:

```
if Is64Bit()
    lcSystem = GetSystemWow64Dir()
else
    lcSystem = GetSystemDir()
endif Is64Bit()
lcAdmin = lcSystem + 'odbcad32.exe'
```

We'll look at the Is64Bit function later.

Run TestSystemDirs.PRG to test these three functions.

Setting a file date/time

There may be times (no pun intended) you wish to change the date/time of a file. For example, if you install a file on a user's system, it'll have the original date/time when you created it on your system. You may wish to change that to the date/time it was installed on the user's system. For that, use the SetFileTime API function.

SetFileTime isn't as straight-forward to use as you might think. First, it requires a file handle, so the file must already be open and must be closed after using SetFileTime. We'll use the Windows API CreateFile (despite its name, it can open an existing file) and CloseHandle functions for those tasks. Second, SetFileTime expects date/time values to be passed in a structure. The GetDateTimeStructure function below takes care of that.

SetFileDateTime.PRG is a wrapper for SetFileTime that handles these tasks. Pass it the name of the file to alter and the date/time values for the creation, last accessed, and last modified attributes of the file. All date/time values are optional; don't pass one to leave that attribute untouched.

```
lparameters tcFileName, ;
    ttCreated, ;
    ttAccessed, ;
    ttModified
local lnFile, ;
    lcCreated, ;
    lcAccessed, ;
    lcModified, ;
    lnResult

* Declare the Windows API functions we need.

declare integer CloseHandle in Kernel32.dll ;
    integer HANDLE_hObject
declare integer CreateFile in Kernel32.dll as apiCreateFile ;
    string @ LPCTSTR_lpFileName, ;
    long DWORD_dwDesiredAccess, ;
    long DWORD_dwShareMode, ;
    string @ LPSECURITY_ATTRIBUTES_lpSecurityAttributes, ;
    long DWORD_dwCreationDisposition, ;
    long DWORD_dwFlagsAndAttributes, ;
    integer HANDLE_hTemplateFile
declare integer SetFileTime in Kernel32.dll ;
    long HANDLE_hFile, ;
    string @ const_FILETIME_ptr_lpCreationTime,   ;
    string @ const_FILETIME_ptr_lpLastAccessTime,  ;
    string @ const_FILETIME_ptr_lpLastWriteTime
declare integer SystemTimeToFileTime in Kernel32.dll ;
    string @lpSystemTime, string @lpFileTime
declare integer LocalFileTimeToFileTime in Kernel32.dll ;
    string @lpLocalTime, string @lpFileTime

* Define some constants.

#define GENERIC_READ              0x80000000
#define FILE_WRITE_ATTRIBUTES     0x0100
#define FILE_SHARE_READ           0x00000001
#define FILE_SHARE_DELETE         0x00000004
```

```
#define OPEN_EXISTING                3
#define FILE_FLAG_BACKUP_SEMANTICS   0x02000000

* Open the file. If we can't, return .F.

lnFile = apiCreateFile(tcFileName, bitor(GENERIC_READ, FILE_WRITE_ATTRIBUTES), ;
    bitor(FILE_SHARE_READ, FILE_SHARE_DELETE), 0, OPEN_EXISTING, ;
    FILE_FLAG_BACKUP_SEMANTICS, 0)
if lnFile < 0
    return .F.
endif lnFile < 0

* Put the specified dates into structures.

lcCreated  = iif(empty(ttCreated)  or vartype(ttCreated)  <> 'T', .NULL., ;
    GetDateTimeStructure(ttCreated))
lcAccessed = iif(empty(ttAccessed) or vartype(ttAccessed) <> 'T', .NULL., ;
    GetDateTimeStructure(ttAccessed))
lcModified = iif(empty(ttModified) or vartype(ttModified) <> 'T', .NULL., ;
    GetDateTimeStructure(ttModified))

* Set the file datetime and return if we succeeded or not.

lnResult = SetFileTime(lnFile, @lcCreated, @lcAccessed, @lcModified)
CloseHandle(lnFile)
return lnResult <> 0


function GetDateTimeStructure(ttDateTime)
local lcSystemTime, ;
    lcLocalFileTime, ;
    lcFileTime
lcSystemTime = bintoc(year(ttDateTime), '2RS') + ;
    bintoc(month(ttDateTime), '2RS') + ;
    bintoc(dow(ttDateTime, 1) - 1, '2RS') + ;
    bintoc(day(ttDateTime), '2RS') + ;
    bintoc(hour(ttDateTime), '2RS') + ;
    bintoc(minute(ttDateTime), '2RS') + ;
    bintoc(sec(ttDateTime), '2RS') + ;
    bintoc(0, '2RS')
lcLocalFileTime = replicate(chr(0), 8)
lcFileTime      = replicate(chr(0), 8)
SystemTimeToFileTime(lcSystemTime, @lcLocalFileTime)
LocalFileTimeToFileTime(lcLocalFileTime, @lcFileTime)
return lcFileTime
```

The sample program TestSetFileDateTime.PRG creates a file and sets its creation date one day in the future but its last modified date one month in the past, which is silly but shows how the function works.

Copying, moving, deleting, and renaming files

VFP has commands to copy (but not move), delete, and rename files. However, these operations don't provide any feedback, such as the usual Windows progress dialog. The SHFileOperation API function provides the same functionality but does optionally provide feedback.

SHFileOperation is fairly complex to call as it expects memory to be allocated for a structure. Fortunately, Sergey Berezniker has done all of the work and put it into a wrapper he published at http://tinyurl.com/3vwykxc. I've adapted the code slightly in FileOperation.PRG. This code requires another file called ClsHeap.PRG, written by the late Ed Rauh, to provide memory allocation functionality. Here's the code for FileOperation.PRG:

```
lparameters tcSource, ;
    tcDestination, ;
    tcAction, ;
    tlUserCanceled
local lcLibrary, ;
    loHeap, ;
    lcAction, ;
    laActionList[1], ;
    lnAction, ;
    lcSourceString, ;
    lcDestString, ;
    lnStringBase, ;
    lnFlag, ;
    lcFileOpStruct, ;
    lnRetCode
```

```
#define ccNULL                   chr(0)

#define FO_MOVE                  0x0001
#define FO_COPY                  0x0002
#define FO_DELETE                0x0003
#define FO_RENAME                0x0004

#define FOF_MULTIDESTFILES       0x0001
#define FOF_CONFIRMMOUSE         0x0002
#define FOF_SILENT               0x0004  && don't create progress/report
#define FOF_RENAMEONCOLLISION    0x0008
#define FOF_NOCONFIRMATION       0x0010  && don't prompt the user.
#define FOF_WANTMAPPINGHANDLE    0x0020  && fill in SHFILEOPSTRUCT.hNameMappings
                                         && must be freed using SHFreeNameMappings
#define FOF_ALLOWUNDO            0x0040  && DELETE - sends the file to the Recycle Bin
#define FOF_FILESONLY            0x0080  && on *.*, do only files
#define FOF_SIMPLEPROGRESS       0x0100  && means don't show names of files
#define FOF_NOCONFIRMMKDIR       0x0200  && don't confirm making any needed dirs
#define FOF_NOERRORUI            0x0400  && don't put up error UI
#define FOF_NOCOPYSECURITYATTRIBS 0x0800  && don't copy NT file Security Attributes

* Declare the Windows API function we need and create a Heap object.

declare integer SHFileOperation in SHELL32.DLL string @ LPSHFILEOPSTRUCT
lcLibrary = 'ClsHeap.prg'
loHeap    = newobject('Heap', lcLibrary)

* Ensure we have a valid action.

lcAction = upper(iif(empty(tcAction) or vartype(tcAction) <> 'C', 'COPY', ;
    tcAction))
alines(laActionList, 'MOVE,COPY,DELETE,RENAME', ',')
lnAction = ascan(laActionList, lcAction)
if lnAction = 0
    return .NULL.
endif lnAction = 0

* Perform the operation.

lcSourceString = tcSource + ccNULL + ccNULL
lcDestString   = tcDestination + ccNULL + ccNULL
lnStringBase   = loHeap.AllocBlob(lcSourceString + lcDestString)
lnFlag         = FOF_NOCONFIRMATION + FOF_NOCONFIRMMKDIR + FOF_NOERRORUI
lcFileOpStruct = loHeap.NumToLong(_screen.hWnd) + ;
    loHeap.NumToLong(lnAction) + ;
    loHeap.NumToLong(lnStringBase) + ;
    loHeap.NumToLong(lnStringBase + len(lcSourceString)) + ;
    loHeap.NumToLong(lnFlag) + ;
    loHeap.NumToLong(0) + loHeap.NumToLong(0) + loHeap.NumToLong(0)
lnRetCode = SHFileOperation(@lcFileOpStruct)

* Flag if the user cancel the operation and return success or failure.

tlUserCanceled = substr(lcFileOpStruct, 19, 4) <> loHeap.NumToLong(0)
return lnRetCode = 0
```

To use it, pass the name of the source file (which can use wildcards, such as "*.DBF"), the destination folder or file name, the action ("move", "copy", "rename", or "delete"), and a logical variable passed by reference; that variable is set to .T. if the user canceled the operation. FileOperation.PRG returns .T. if the operation succeeded. If the operation performs on a large enough group of files, SHFileOperation displays a progress dialog; otherwise, it works silently. As you can see from some of the constants defined in FileOperation.PRG, you can configure how SHFileOperation works by setting the lnFlag variable to the desired value.

TestFileOperation.PRG demonstrates how FileOperation works: it copies the entire VFP directory, including all subdirectories, to a folder named FileOperation in your temporary files folder. As it runs, it displays the usual Windows file progress dialog, and displays whether the operation succeeded when it's done. It then deletes the files in the temporary folder to clean up after itself, again displaying the progress dialog.

Getting information about a volume

The GetVolumeInformation API function can provide useful information about a specific volume, including the volume name, serial number, and what features it supports. Most of this information is stored as bit-wise flags in

a Long value, so you have to use one of VFP's BIT functions (such as BITTEST) to determine what the setting for a certain flag is.

GetVolInfo.prg uses GetVolumeInformation to fill an array with information about the specified volume. For the first parameter, pass a drive letter, a volume name, or a fully qualified file name (to get the space on the volume that file resides on); it has code to specifically handle UNC names. The second parameter should be the array, passed by reference using @, in which the volume information should be placed; upon return, it'll contain the information shown in **Table 2**.

Table 2. The contents of the array set by GetVolInfo.PRG.

Column	Information
1	Volume name
2	Volume serial number
3	File system name (FAT, NTFS, etc.)
4	.T. if the file system supports case-sensitive filenames
5	.T. if the file system preserves case of filenames
6	.T. if the file system supports Unicode filenames
7	.T. if the file system preserves and enforces ACLs (NTFS only)
8	.T. if the file system supports file-based compression
9	.T. if the volume is compressed
10	Maximum filename length

```
lparameters tcVolume, ;
    taArray
local lcVolume, ;
    lcVolumeName, ;
    lnVolumeNameLen, ;
    lnVolumeSerialNumber, ;
    lnMaxFileNameLen, ;
    lnFileSystemFlags, ;
    lcFileSystemName, ;
    lnFileSystemNameLen, ;
    lcFileInfo, ;
    lcFileName, ;
    laFiles[1], ;
    lnFiles, ;
    lnI, ;
    lcFile, ;
    lnHandle

* Declare the API function and constants.

declare GetVolumeInformation in Win32API ;
    string lpRootPathName, string @lpVolumeNameBuffer, ;
    integer nVolumeNameSize, integer @lpVolumeSerialNumber, ;
    integer @lpMaxFilenameLength, integer @lpFileSystemFlags, ;
    string @lpFileSystemNameBuffer, integer nFileSystemNameSize
#define cnFS_CASE_SENSITIVE          0
#define cnFS_CASE_IS_PRESERVED       1
#define cnFS_UNICODE_STORED_ON_DISK  2
#define cnFS_PERSISTENT_ACLS         3
#define cnFS_FILE_COMPRESSION        4
#define cnFS_VOL_IS_COMPRESSED      15

* If the path wasn't specified, use the current drive. Otherwise,
* get the drive for the specified path, handling UNC paths
* specially.

do case
    case vartype(tcVolume) <> 'C' or empty(tcVolume)
        lcVolume = addbs(sys(5))
    case left(tcVolume, 2) = '\\'
        lcVolume = addbs(tcVolume)
        lcVolume = left(lcVolume, at('\', lcVolume, 4))
    case len(tcVolume) = 1
        lcVolume = tcVolume + ':\'
    otherwise
```

```
        lcVolume = addbs(justdrive(tcVolume))
endcase

* Create the parameters for the API function, then call it.

lcVolumeName         = space(255)
lnVolumeNameLen      = len(lcVolumeName)
lnVolumeSerialNumber = 0
lnMaxFileNameLen     = 0
lnFileSystemFlags    = 0
lcFileSystemName     = space(255)
lnFileSystemNameLen  = len(lcFileSystemName)
GetVolumeInformation(lcVolume, @lcVolumeName, lnVolumeNameLen, ;
    @lnVolumeSerialNumber, @lnMaxFileNameLen, @lnFileSystemFlags, ;
    @lcFileSystemName, lnFileSystemNameLen)

* Put the information into the array.

dimension taArray[10]
taArray[ 1] = left(lcVolumeName, at(chr(0), lcVolumeName) - 1)
taArray[ 2] = lnVolumeSerialNumber
taArray[ 3] = left(lcFileSystemName, at(chr(0), lcFileSystemName) - 1)
taArray[ 4] = bittest(lnFileSystemFlags, cnFS_CASE_SENSITIVE)
taArray[ 5] = bittest(lnFileSystemFlags, cnFS_CASE_IS_PRESERVED)
taArray[ 6] = bittest(lnFileSystemFlags, cnFS_UNICODE_STORED_ON_DISK)
taArray[ 7] = bittest(lnFileSystemFlags, cnFS_PERSISTENT_ACLS)
taArray[ 8] = bittest(lnFileSystemFlags, cnFS_FILE_COMPRESSION)
taArray[ 9] = bittest(lnFileSystemFlags, cnFS_VOL_IS_COMPRESSED)
taArray[10] = lnMaxFileNameLen
return .T.
```

I use this function for software licensing; the license is tied to a particular system by getting the serial number of the hard drive and hashing into a license code.

To see GetVolInfo.prg in use, run TestVolInfo.prg.

Getting free and total space on a drive

The VFP DISKSPACE() function returns either the free or total space on a specified drive, depending on what parameters you pass. However, DISKSPACE() fails when either value is more than 2 GB, which given the size of today's drive makes it useless. Instead, use the GetDiskFreeSpaceEx API function. GetDriveSpace.prg is a wrapper for this function. Pass it a drive letter or directory name (UNC paths are supported) followed by .T. to return the total space or .F. to return the free space.Run TestGetDriveSpace.prg to test it.

```
lparameters tcDirectory, ;
    tlTotal
#define ccNULL chr(0)
local lcDir, ;
    lnResult, ;
    lcDLL, ;
    lnFunction, ;
    lnHandle, ;
    lcCaller, ;
    lcTotal, ;
    lcFree

* Declare Win32API functions to determine if the GetDiskFreeSpaceEx function is
* available.

declare integer GetModuleHandle in Win32API ;
    string @lpModuleName
declare integer GetProcAddress in Win32API ;
    integer hModule, string @lpProcName

* If the path wasn't specified, use the current drive. Otherwise, get the drive
* for the specified path, handling UNC paths specially.

do case
    case vartype(tcDirectory) <> 'C' or empty(tcDirectory)
        lcDir = addbs(sys(5))
    case left(tcDirectory, 2) = '\\'
        lcDir = addbs(tcDirectory)
        lcDir = left(lcDir, at('\', lcDir, 4))
    case len(tcDirectory) = 1
        lcDir = tcDirectory + ':'
```

```
    otherwise
        lcDir = addbs(justdrive(tcDirectory))
endcase
lnResult = -1

* See if the GetDiskFreeSpaceEx Win32API function is available; if so, call it
* to get the drive space.

lcDLL      = 'kernel32.dll'
lcFunction = 'GetDiskFreeSpaceExA'
lnHandle   = GetModuleHandle(@lcDLL)
if GetProcAddress(lnHandle, @lcFunction) > 0
    declare short GetDiskFreeSpaceEx in Win32API ;
        string @lpDirectoryName, string @lpFreeBytesAvailableToCaller, ;
        string @lpTotalNumberOfBytes, string @lpTotalNumberOfFreeBytes
    store replicate(ccNULL, 8) to lcCaller, lcTotal, lcFree
    if GetDiskFreeSpaceEx(@lcDir, @lcCaller, @lcTotal, @lcFree) <> 0
        lnResult = Hex2Decimal(iif(tlTotal, lcTotal, lcFree))
    endif GetDiskFreeSpaceEx(@lcDir, ...

* Since we can't use GetDiskFreeSpaceEx, just use DISKSPACE.

else
    lnResult = diskspace(lcDir)
endif GetProcAddress(lnHandle, @lcFunction) > 0
return lnResult
```

There are a couple of interesting things about this code:

- Since the GetDiskFreeSpaceEx function was added in Windows XP, this code checks to see whether the function is available by checking whether it exists in kernel32.dll. If not, DISKSPACE() is used instead.
- Normally, you'd use CTOBIN() to convert the output of an API function to a value VFP can use. However, CTOBIN() doesn't support the "large integer" data type output by GetDiskFreeSpaceEx. So, the wrapper code calls Hex2Decimal.prg to convert the value into a VFP numeric.

Network functions

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 calls this function and returns the user's name. Run TestGetUserName.prg to test it.

```
local lcName, ;
    lnSize, ;
    lcUser, ;
    lnStatus

* Initialize the variables we need and declare the Windows function.

lcName = chr(0)
lnSize = 64
lcUser = replicate(lcName, lnSize)
declare integer WNetGetUser in Win32API ;
    string @cName, string @cUser, integer @nBufferSize

* Call the function and return the result.

lnStatus = WNetGetUser(@lcName, @lcUser, @lnSize)
lcUser   = iif(lnStatus = 0, ;
    upper(left(lcUser, at(chr(0), lcUser) - 1)), '')
return lcUser
```

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 http://tinyurl.com/3uwhs4h for error codes); for example, 2250 means the device specified wasn't mapped to anything.

GetNetConnection.prg is a wrapper for this function.

```
lparameters tcDevice
local lcName, ;
    lnLength, ;
    lcBuffer

* If the path wasn't specified, use the current drive.

if vartype(tcDevice) <> 'C' or empty(tcDevice)
    lcName = sys(5)
else
    lcName = tcDevice
endif vartype(tcDevice) <> 'C' ...

* Set up the buffer and declare the WinAPI function we'll need.

lnLength = 261
lcBuffer = replicate(chr(0), lnLength)
declare integer WNetGetConnection in Win32API ;
    string @lpLocalName, string @lpRemoteName, integer @lpnLength

* Call the API function and return the result.

if WNetGetConnection(lcName, @lcBuffer, @lnLength) = 0
    lcReturn = left(lcBuffer, at(chr(0), lcBuffer) - 1)
else
    lcReturn = ''
endif WNetGetConnection(lcName, ...
return lcReturn
```

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). It returns 0 if it was successful or an error code if not.

AddNetConnection.prg is a wrapper for this function.

```
lparameters tcResource, ;
    tcDevice, ;
    tcPassword
local lcPassword, ;
    llReturn

* Ensure valid parameters were passed.

assert vartype(tcResource) = 'C' and not empty(tcResource) ;
    message 'Invalid network resource name'
assert vartype(tcDevice) = 'C' and not empty(tcDevice) ;
    message 'Invalid local device name'

* If the password wasn't specified, use a blank.

lcPassword = iif(vartype(tcPassword) <> 'C', '', tcPassword)

* Declare and call the API function and return the result.

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

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 (for example, a WAN connection for a laptop), 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.

```
lparameters tcDevice
local lcName, ;
    llReturn

* If the path wasn't specified, use the current drive.

if vartype(tcDevice) <> 'C' or empty(tcDevice)
    lcName = sys(5)
else
    lcName = tcDevice
endif vartype(tcDevice) <> 'C' ...

* Declare and call the API function and return the result.

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

Other functions

Detecting a 64-bit operating system

Earlier, I mentioned using Is64Bit to identify whether you're running a 64-bit operating system or not. Is64Bit.PRG is a wrapper adapted from code written by Sergey and published at http://tinyurl.com/3z2a4uw. This code uses several API functions, especially IsWow64Process. It first detects whether that function exists on this version of Windows to avoid getting an error on an older operating system.

```
local llIsWow64ProcessExists, ;
    llIs64BitOS, ;
    lnIsWow64Process
declare long GetModuleHandle in Win32API string lpModuleName
declare long GetProcAddress  in Win32API long hModule, string lpProcName
llIsWow64ProcessExists = GetProcAddress(GetModuleHandle('kernel32'), ;
    'IsWow64Process') <> 0
if llIsWow64ProcessExists
    declare long GetCurrentProcess in Win32API
    declare long IsWow64Process    in Win32API long hProcess, long @Wow64Process
    lnIsWow64Process = 0
* IsWow64Process return value is non-zero if it succeeds
* The second output parameter value is non-zero if we're running under 64-bit OS
    if IsWow64Process(GetCurrentProcess(), @lnIsWow64Process) <> 0
        llIs64BitOS = lnIsWow64Process <> 0
    endif IsWow64Process(GetCurrentProcess(), @lnIsWow64Process) <> 0
endif llIsWow64ProcessExists
return llIs64BitOS
```

Determining if an application is running as administrator

As you may know, logging in as an administrator doesn't necessarily mean you have administrative rights on Windows Vista and later. Instead, you're assigned a split security token, one part of which is a normal user and the other is an administrator. Windows then launches the desktop using the normal user token. That means that you can't execute tasks that require administrative rights unless you "elevate" to the administrator token. In

Windows Vista, this means using the User Access Control (UAC) dialog. In Windows 7, you can configure it to automatically elevate you under certain conditions.

If an application requires administrative rights for certain tasks, there are two ways to handle it:

- Make the user run the application as an administrator by right-clicking its icon and choosing Run as Administrator. This isn't a good idea because the entire application runs with administrator rights, so if the application becomes infected with malware, it can do anything it wants to your system.
- A better approach is to run the application as a normal user and ask to elevate when a task requiring administrator rights is supposed to execute.

We'll see later how to ask the user to elevate but of course you don't have to do that if the user ran the application as administrator. The IsUserAnAdmin API function tells you whether that's the case. IsAdministrator.PRG is a wrapper for that function. Since that function doesn't exist on older operating systems, we need to check whether it's available or not. The code in IsAdministrator.PRG does just that.

```
local llIsUserAnAdminExists, ;
    llReturn
try
    declare long GetModuleHandle in Win32API string lpModuleName
    declare long GetProcAddress  in Win32API long hModule, string lpProcName
    llIsUserAnAdminExists = GetProcAddress(GetModuleHandle('shell32'), ;
        'IsUserAnAdmin') <> 0
    if llIsUserAnAdminExists
        declare integer IsUserAnAdmin in shell32
        llReturn = IsUserAnAdmin() = 1
    else
        llReturn = .T.
    endif llIsUserAnAdminExists
catch
endtry
return llReturn
```

To see this function in action, run VFP and type MESSAGEBOX(IsAdministrator()); it should display .F. Quit VFP, then right-click its icon and choose Run as Administrator. Type MESSAGEBOX(IsAdministrator()) and notice that it now displays .T.

"Executing" a file

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 Microsoft 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. However, I prefer parameters to be ordered so that optional parameters are at the end, so I created ExecuteFile.prg that has parameters in the order I want. Pass this routine the name of the file and optionally the action ("Open" is used if it isn't passed), the working directory, and any parameters to pass to the application. ExecuteFile returns a value over 32 indicating success, -1 if no filename was passed, 2 for a bad association, 29 for failure to load the application, 30 if the application is busy, or 31 if there is no application association for that file extension.

```
lparameters tcFileName, ;
    tcOperation, ;
    tcWorkDir, ;
    tcParameters
local lcFileName, ;
    lcWorkDir, ;
    lcOperation, ;
    lcParameters
```

```
if empty(tcFileName)
    return -1
endif empty(tcFileName)
lcFileName   = alltrim(tcFileName)
lcWorkDir    = iif(vartype(tcWorkDir) = 'C', alltrim(tcWorkDir), '')
lcOperation  = iif(vartype(tcOperation) = 'C' and ;
    not empty(tcOperation), alltrim(tcOperation), 'Open')
lcParameters = iif(vartype(tcParameters) = 'C', ;
    alltrim(tcParameters), '')
lnShow       = iif(upper(lcOperation) = 'Print', 0, 1)
declare integer ShellExecute in SHELL32.DLL ;
    integer nWinHandle, ;    && handle of parent window
    string cOperation, ;     && operation to perform
    string cFileName, ;      && filename
    string cParameters, ;    && parameters for the executable
    string cDirectory, ;     && default directory
    integer nShowWindow      && window state
return ShellExecute(0, lcOperation, lcFilename, lcParameters, ;
    lcWorkDir, lnShow)
```

If you need to run an application as administrator, pass "RunAs" for the second parameter to force the user to elevate to administrator. That isn't required if they've already run the application as administrator, so use IsAdministrator.PRG to determine whether to pass that value or not.

Run TestExecuteFile.prg to see an example.

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 actually plays 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 appears; that same sound plays when you pass a value from 16 to 31 to MessageBeep. **Table 3** shows the range of values you can use.

Table 3. The values to pass to MessageBeep for different sounds.

Range	Sound	MESSAGEBOX() Constant
0 - 15	Default beep	
16 - 31	Critical stop	MB_ICONSTOP
32 - 47	Question	MB_ICONQUESTION
48 - 63	Exclamation	MB_ICONEXCLAMATION
64 - 79	Asterisk	MB_ICONINFORMATION

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

```
lparameters tnSound
local lnSound
lnSound = iif(vartype(tnSound) = 'N', tnSound, 16)
declare integer MessageBeep in Win32API ;
    integer wType
MessageBeep(lnSound)
```

Reading from an INI file

Although the Registry is the "in" place for storing configuration information, sometimes it's just handier to store it in an INI file. It's easier to explain to a user over the phone how to open an INI file in Notepad to change a setting than to fire up RegEdit, locate the appropriate key, and change the setting. Also, when an application is removed from a system, it's easier to delete an INI file than track down all the Registry entries to delete.

INI files are just text files consisting of lines of entry/value pairs (separated with an "=") organized by sections; section names are enclosed in square brackets. For example, the following INI file has two sections, Application and Settings. In the Application setting, the Server entry has the value "t:\appname\".

```
[Application]
Server = t:\appname\
Data = data\

[Settings]
Confirm = off
```

The GetPrivateProfileString API function reads the value for the specified entry in a specified section of an INI file and puts the result into a buffer. One gotcha: the INI file name must include a fully-qualified path or this function won't work.

ReadINI.prg accepts an INI file name, the section and entry names, and a default value to return if the specified entry doesn't exist (a blank string is used if that parameter isn't passed), and returns either the value for the entry or the default value. It uses FULLPATH() on the passed file name to ensure it has a fully-qualified path. Use TestReadINI.prg to show how this routine works.

```
lparameters tcINIFile, ;
    tcSection, ;
    tcEntry, ;
    tcDefault
local lnSize, ;
    lcBuffer, ;
    lcDefault
declare integer GetPrivateProfileString in Win32API ;
    string cSection, string cEntry, string cDefault, ;
    string @cBuffer, integer nBufferSize, string cINIFile
lnSize    = 2048
lcBuffer  = replicate(chr(0), lnSize)
lcDefault = iif(vartype(tcDefault) <> 'C', '', tcDefault)
GetPrivateProfileString(tcSection, tcEntry, lcDefault, @lcBuffer, ;
    lnSize, fullpath(tcINIFile))
return strtran(lcBuffer, chr(0), '')
```

Determining if the application is already running

Have you ever gone to a client's office and found four or five copies of your application running at the same time? That's because the user started up the application, minimized it, then later, forgetting it was already open, fired up the application again (and again and again ...). You can prevent that from happening in the future by calling IsAppRunning.prg as one of the first tasks in your application. If it returns .T., another instance of the application is already running and has been normalized and made the active window, so this instance can just quit. The effect is that running the application a second time simply causes the first instance to be activated.

This routine uses a half-dozen Win32API functions. GetActiveWindow returns the Windows handle for the current window (a window handle is a unique ID assigned to a window that API functions can access it by), GetWindow retrieves a handle to a window that has the specified relationship to the specified window, GetClassName gets the name of the window class to which the specified window belongs, GetWindowText returns the caption for the specified window, SetForegroundWindow brings the specified window to the top, and ShowWindow sets the specified window's show state.

```
lparameters tcWindowName
local llIsAppRunning, ;
    lcWindowName, ;
    lnFoxHWnd, ;
    lnHWndNext, ;
    lnBuffer, ;
    lcBuffer, ;
    lcOurClass, ;
    lcClass

* Define some constants and WinAPI functions.

#define GW_OWNER 4
#define GW_HWNDFIRST 0
#define GW_HWNDNEXT 2
#define SW_SHOWNORMAL 1
declare integer SetForegroundWindow in Win32API ;
```

```
    integer lnhWnd
declare integer GetWindowText in Win32API ;
    integer lnhWnd, string lpString, integer cCH
declare integer GetWindow in Win32API ;
    integer lnhWnd, integer wCmd
declare integer GetActiveWindow in Win32API
declare integer GetClassName in Win32API ;
    integer lnhWnd, string @lpClassName, integer lnMaxCount
declare integer ShowWindow in Win32API ;
    integer lnhWnd, integer lnCmdShow

* If a window name wasn't passed, use _screen.Caption.

lcWindowName = iif(vartype(tcWindowName) = 'C' and ;
    not empty(tcWindowName), tcWindowName, _screen.Caption)
lnFoxHWnd    = GetActiveWindow()
lnHWndNext   = GetWindow(lnFoxHWnd, GW_HWNDFIRST)

* Get our window class name.

lnBuffer   = 200
lcBuffer   = space(lnBuffer)
lnBuffer   = GetClassName(lnFoxHWnd, @lcBuffer, lnBuffer)
lcOurClass = left(lcBuffer, lnBuffer - 1)

* Check each window for the specified title. Skip this application's
* window and any child windows. If we find one, bring it to the
* front.

do while lnHWndNext <> 0
    if lnHWndNext <> lnFoxHWnd and ;
        GetWindow(lnHWndNext, GW_OWNER) = 0
        lcTitle = space(64)
        GetWindowText(lnHWndNext, @lcTitle, 64)
        if lcWindowName $ lcTitle
            lnBuffer = 200
            lcBuffer = space(lnBuffer)
            lnBuffer = GetClassName(lnHWndNext, @lcBuffer, lnBuffer)
            lcClass  = left(lcBuffer, lnBuffer - 1)
            if lcClass == lcOurClass
                llIsAppRunning = .T.
                SetForegroundWindow(lnHWndNext)
                ShowWindow(lnHWndNext, SW_SHOWNORMAL)
                exit
            endif lcClass == lcOurClass
        endif lcWindowName $ lcTitle
    endif lnHWndNext <> lnFoxHWnd ...
    lnHWndNext = GetWindow(lnHWndNext, GW_HWNDNEXT)
enddo while lnHWndNext <> 0
return llIsAppRunning
```

The parameter is optional; if it isn't specified, _SCREEN.Caption is used as the window title. Be sure to pass this parameter if the application is running as a top-level form.

To see an example of this routine, run TEST.EXE from Windows Explorer, don't close the form just yet, and run it again. Notice that you only see one instance of the application start up; that's because the main program calls IsAppRunning and QUITs if it returns .T.

Terminating an application

If you ever need to programmatically terminate another application, KillProcess.PRG is for you. This routine uses six Win32API functions: FindWindow, which returns a handle to a window by its caption, SendMessage, which sends a Windows message to a window, GetWindowThreadProcessId, which gets the process ID for a window, OpenProcess, which opens a process for access, TerminateProcess, which terminates a process, and Sleep, which causes the application to turn control back over to the Windows event handler for a specified period (better than going into a hard loop, such as an empty FOR I = 1 TO 100000, which hogs time slices). First, it tries to get a handle to the window for the specified caption. If it succeeds, it sends a "destroy" message to the window, then waits for up to five seconds to see if the window has gone away (using Sleep to wait for a while). If not, it then finds the process ID for the window and terminates the process.

Notice something interesting about FindWindow. Although the first parameter is declared as a string, in fact we're passing a number to it. The reason is because the first parameter is actually a pointer to a string. Since a

pointer is a memory address, which is just a number, we can pass a pointer ourselves instead of passing the string by reference. We actually want to pass a null value for the pointer, so passing 0 does the trick.

OpenProcess and TerminateProcess can also be used with the process ID returned by _VFP.ProcessID. My friend Dan Lauer uses this to terminate VFP COM servers that he suspects may have crashed. After instantiating the server, he stores its process ID (the Init of the server puts its _VFP.ProcessID in a property) and later, if it seems to have stopped responding, uses these API functions to terminate its process.

```
lparameters tcCaption
local lnhWnd, ;
    llReturn, ;
    lnProcessID, ;
    lnHandle

* Declare the Win32API functions we need.

#define WM_DESTROY 0x0002
declare integer FindWindow in Win32API ;
    string @cClassName, string @cWindowName
declare integer SendMessage in Win32API ;
    integer hWnd, integer uMsg, integer wParam, integer lParam
declare Sleep in Win32API ;
    integer nMilliseconds
declare integer GetWindowThreadProcessId in Win32API ;
    integer hWnd, integer @lpdwProcessId
declare integer OpenProcess in Win32API ;
    integer dwDesiredAccess, integer bInheritHandle, ;
    integer dwProcessID
declare integer TerminateProcess in Win32API ;
    integer hProcess, integer uExitCode

* Get a handle to the window by its caption.

lnhWnd   = FindWindow(0, tcCaption)
llReturn = lnhWnd = 0

* If we found the window, send a "destroy" message to it, then wait
* for it to be gone. If it didn't, let's use the big hammer: we'll
* terminate its process.

if not llReturn
    SendMessage(lnhWnd, WM_DESTROY, 0, 0)
    llReturn = WaitForAppTermination(tcCaption)
    if not llReturn
        lnProcessID = 0
        GetWindowThreadProcessId(lnhWnd, @lnProcessID)
        lnHandle = OpenProcess(1, 1, lnProcessID)
        llReturn = TerminateProcess(lnHandle, 0) > 0
    endif not llReturn
endif not llReturn
return llReturn

* For up to five times, wait for a second, then see if the specified
* application is still running. Return .T. if the application has
* terminated.

function WaitForAppTermination
lparameters tcCaption
local lnCounter, ;
    llReturn
lnCounter = 0
llReturn  = .F.
do while not llReturn and lnCounter < 5
    Sleep(1000)
    lnCounter = lnCounter + 1
    llReturn  = FindWindow(0, tcCaption) = 0
enddo while not llReturn ...
return llReturn
```

To see this routine in action, run TEST.EXE from the Windows Explorer, then run TestKillProcess.prg.

Really locking a window

Normally, setting the LockScreen property of _screen or a form to .T. prevents the window from being drawn while it's updated; this is used for faster refreshes or to eliminating "flashing". Unfortunately, there are times

when LockScreen either doesn't work properly or can't be used. For example, because ActiveX controls like the TreeView control live in their own window, they don't respect the LockScreen status of a form. Suppose you have a form with a TreeView on it and a "refresh" button to reload the TreeView so the user can see changes other users on the network have made. When the user clicks the refresh button, the current nodes in the TreeView are removed and a new set added. The problem is that the user sees the TreeView go blank, then the nodes added (maybe even seeing them added one at a time if the process to add nodes is slow enough), even if ultimately the set of nodes they end up seeing is the same.

The solution is to use the LockWindowUpdate API function. This function works like the LockScreen property, but works at the Windows level rather than in VFP. It expects the handle to the window to lock to be passed; pass 0 to unlock updates. LockWindow.prg is a wrapper for this function; pass .T. to lock the current window or .F. to unlock it. You can optionally pass the handle to the window to lock (its hWnd property); if you omit this parameter, LockWindow calls the GetDesktopWindow Win32API function to get the handle to the active window.

```
lparameters tlLock, ;
    tnHWnd
local lnHWnd
declare integer LockWindowUpdate in Win32API ;
    integer nHandle
do case
    case not tlLock
        lnHWnd = 0
    case pcount() = 1
        declare integer GetDesktopWindow in Win32API
        lnHWnd = GetDesktopWindow()
    otherwise
        lnHWnd = tnHWnd
endcase
LockWindowUpdate(lnHWnd)
return
```

To see an example of this function, run LockWindow.scx. Click the Filter and All Records buttons and watch the TreeView clear and then redraw. Turn on the "Lock TreeView" checkbox, then click the buttons again; this time, the changes to the TreeView snap into place. The Lock Filter button does its magic by calling LockWindow, passing it the handle to the TreeView control (stored in its hWnd property) so it's locked while nodes are cleared and re-added.

Getting time zone settings

There are times (no pun intended) when you need to convert between local and GMT time. For example, if your users need to exchange time-sensitive information with users in other locations, it's best to store times in GMT rather than local time. The GetTimeZoneInformation API function can tell you the name and offset from GMT for the current time zone as well as whether daylight saving time is currently in effect. GetTimeZoneInfo.prg is a wrapper for that function. It returns an object with TimeZoneDesc (the time zone name) and TimeZoneOffset (the offset in seconds from GMT) properties. Run TestGetTimeZoneInfo.prg to see an example.

```
local loObject, ;
    lcTimeZone, ;
    lnID, ;
    lnStandardOffset, ;
    lnDaylightOffset

* Create an object to hold the time zone information.

loObject = createobject('Empty')
addproperty(loObject, 'TimeZoneDesc')
addproperty(loObject, 'TimeZoneOffset')

* Declare the time zone information API function and get the time zone
* information.

#define TIME_ZONE_SIZE  172
declare integer GetTimeZoneInformation in kernel32 ;
    string @lpTimeZoneInformation
lcTimeZone = replicate(chr(0), TIME_ZONE_SIZE)
lnID       = GetTimeZoneInformation(@lcTimeZone)

* Determine the standard and daylight time offset.
```

```
lnStandardOffset = ctobin(substr(lcTimeZone,   1, 4), '4RS')
lnDaylightOffset = ctobin(substr(lcTimeZone, 169, 4), '4RS')

* Determine the total offset based on whether the computer is on daylight time
* or not. Get the description for the time zone.

if lnID = 2  && daylight time
    loObject.TimeZoneDesc   = strtran(strconv(substr(lcTimeZone, 89, 64), ;
        6), chr(0), '')
    loObject.TimeZoneOffset = (lnStandardOffset + lnDaylightOffset) * 60
else   && standard time
    loObject.TimeZoneDesc   = strtran(strconv(substr(lcTimeZone,  5, 64), ;
        6), chr(0), '')
    loObject.TimeZoneOffset = lnStandardOffset * 60
endif lnID = 2
return loObject
```

Returning a result code to a calling program

If your application is called from another Windows application (a BAT file, Windows Task Scheduler, etc.), you may want to return a result code to the caller to indicate whether it succeeded or not. To do that, call the ExitProcess API function as the last line of code in your application, passing it a value to return. This sets the error return code for the application. 0 is the expected value if everything worked correctly. You can use any value you wish to indicate that a problem occurred; you can even return a different value for each type of problem that may have occurred. Here's the declaration of ExitProcess:

```
declare ExitProcess in Win32API integer ExitCode
```

To call it, simply use ExitProcess(some value). Make sure there is no other code that has to execute after this statement (that is, you've done all the cleanup work necessary) because executing this immediately terminates the application.

Determining what colors to use

It's not a good idea to select your own colors for certain things in your application. Otherwise, your application won't match other applications that respect the user's carefully selected color scheme. Instead, call the GetSysColor API function to determine which color to use. Pass it a value indicating the Windows object you want the color for (see the MSDN topic on GetSysColor or GetSysColor.H for the values to use) and it returns the RGB value to use.

For example, the default value for Grid.HighlightBackColor may have been correct for Windows XP when VFP 9 was released but isn't the default color used for highlighted text in more modern operating systems, and of course the user could be using a different theme than the default. So, I have code like the following in the Init of my Grid base class (SFGrid):

```
#include GetSysColor.h
local loParent
with This

* Change the font to Segoe UI in Vista and later.

    if os(3) >= '6' and .FontName = 'Tahoma'
        .FontName = 'Segoe UI'
    endif os(3) >= '6' ...

* Use system colors.

    declare integer GetSysColor in User32 integer nIndex
    .HighlightBackColor = GetSysColor(COLOR_HIGHLIGHT)
    .HighlightForeColor = GetSysColor(COLOR_HIGHLIGHTTEXT)
Endwith
```

Figure 1 shows the effect of this code: the top grid is a base class Grid using the default colors for highlighting while the bottom grid is an instance of SFGrid using the correct system colors.

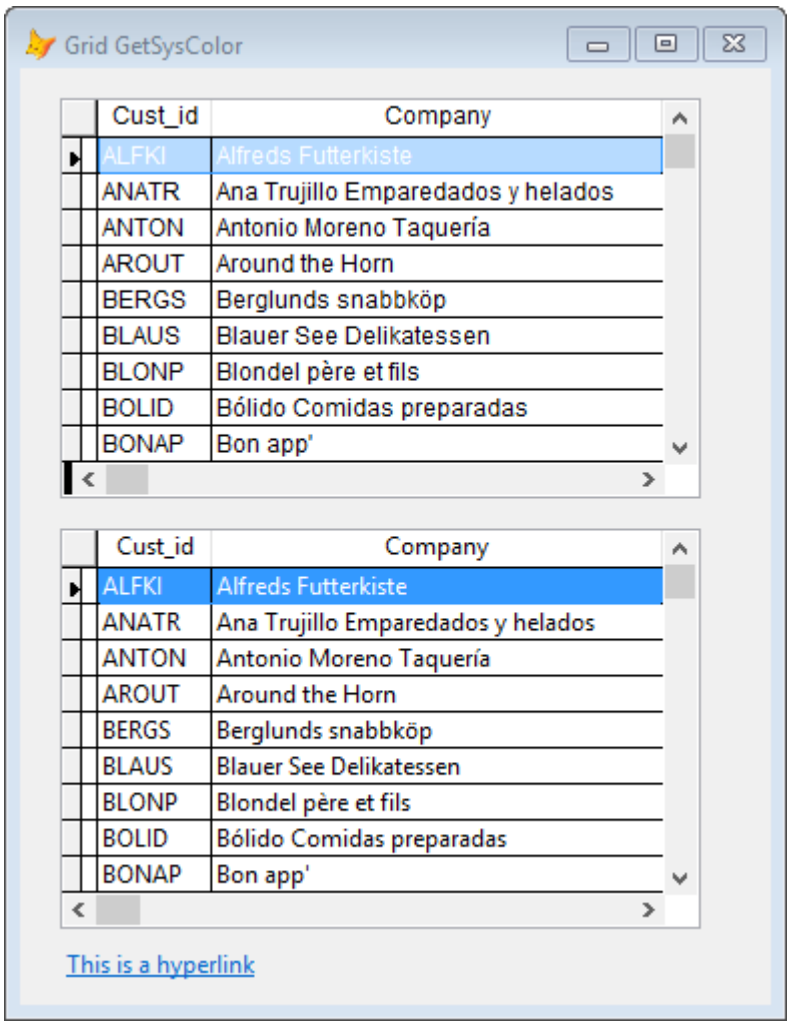


Figure 1. The top grid uses the default colors for highlighting while the bottom grid uses the correct system colors.

Another example of using GetSysColor is SFHyperlinkLabel, a label that provides a hyperlink that works better than the Hyperlink base class (it works with the default browser rather than just Internet Explorer and is much more flexible). The Init and MouseLeave methods set ForeColor to GetSysColor(COLOR_HOTLIGHT) while MouseEnter sets it to GetSysColor(COLOR_MENUHILIGHT).

Handling multiple monitors

One thing that bugs me is applications that can't handle multiple monitors. Typically I have certain applications open on my second monitor. When I travel with my laptop, I don't have a second monitor with me, and when I open some of those applications, they try to be nice and restore themselves to the same size and position as they were the last time, but now they're off-screen because the second monitor isn't there. How hard is it to detect that the location the window should be positioned to doesn't currently exist and to instead position it on the primary monitor? Not that hard, but it does need a few Windows API calls.

The reason you need to use API functions is that the native VFP SYSMETRIC() function only works with the primary monitor. For example, even with two screens with a total of 3520 horizontal pixels (1600 for the primary and 1920 for the secondary), SYSMETRIC(1), which gives the screen width, returns 1600 on my system. So, we'll use the GetSystemMetrics API function to determine the total size available.

SFMonitors.prg defines a couple of classes: SFSize, which just contains properties for size and position settings, and SFMonitors, which has methods to determine the number of monitors, the size of the primary monitor, the size of the total desktop, and the size of the monitor a specific point falls on. Here's the code in the Init and GetPrimaryMonitorSize methods that determines the size of the desktop and how many monitors are available:

```
function Init
```

```
    local loSize

* Declare the Windows API functions we'll need.

    declare integer MonitorFromPoint in Win32API ;
        long x, long y, integer dwFlags
    declare integer GetMonitorInfo in Win32API ;
        integer hMonitor, string @lpmi
    declare integer SystemParametersInfo in Win32API ;
        integer uiAction, integer uiParam, string @pvParam, integer fWinIni
    declare integer GetSystemMetrics in Win32API integer nIndex

* Determine how many monitors there are. If there's only one, get its size. If
* there's more than one, get the size of the virtual desktop.

    with This
        .nMonitors = GetSystemMetrics(SM_CMONITORS)
        if .nMonitors = 1
            loSize   = .GetPrimaryMonitorSize()
            .nRight  = loSize.nRight
            .nBottom = loSize.nBottom
            store 0 to .nLeft, .nTop
        else
            .nLeft   = GetSystemMetrics(SM_XVIRTUALSCREEN)
            .nTop    = GetSystemMetrics(SM_YVIRTUALSCREEN)
            .nRight  = GetSystemMetrics(SM_CXVIRTUALSCREEN) - abs(.nLeft)
            .nBottom = GetSystemMetrics(SM_CYVIRTUALSCREEN) - abs(.nTop)
        endif .nMonitors = 1
    endwith
endfunc

function GetPrimaryMonitorSize
    local lcBuffer, ;
        loSize
    lcBuffer = replicate(chr(0), 16)
    SystemParametersInfo(SPI_GETWORKAREA, 0, @lcBuffer, 0)
    loSize = createobject('SFSize')
    with loSize
        .nLeft   = ctobin(substr(lcBuffer,  1, 4), '4RS')
        .nTop    = ctobin(substr(lcBuffer,  5, 4), '4RS')
        .nRight  = ctobin(substr(lcBuffer,  9, 4), '4RS')
        .nBottom = ctobin(substr(lcBuffer, 13, 4), '4RS')
    endwith
    return loSize
endfunc
```

SFPersistentForm in SFPersist.vcx restores the size and position of a form. It uses SFMonitors to ensure the size and position are consistent with the current desktop. I won't show the code here for brevity; feel free to examine the code in the Restore method yourself. To use SFPersistentForm, drop an instance on a form and set the cKey property to the Registry key in which to store the settings for the form. The first time you run the form, it'll open at its defined size and position, but when you close it, the SFPersistentForm object saves the size and position to the Registry so the next time it's run, those settings are restored, taking into account the fact that the form may have been on a monitor that's no longer available.

To see this in action, run TestGrid.scx. Size and move it somewhere, then close and re-run it. Notice it comes up in the same size and position it was before. Now run Regedit, navigate to HKEY_CURRENT_USER\Software\Win32API\TestGrid and change the setting of Left to some value much higher than the number of horizontal pixels available on your system (for example, on my system, 4000 when the second monitor is attached or 2000 when it isn't). Run the form again and notice that it appears at the right edge of your desktop.

Managing ODBC data sources

If you need to programmatically manage ODBC data sources, there are several API functions that can help. SQLDataSources enumerates data sources, SQLDrivers enumerates drivers, and SQLConfigDataSource adds, modifies, and removes data sources. Because these functions can be tricky to work with, I created SFRegistryODBC in SFRegistry.vcx as a wrapper. The reason the class is called SFRegistryODBC is because some of the functionality requires reading from the Windows Registry (specifically from various subnodes of HKEY_CURRENT_USER\Software\ODBC\ODBC.INI), so this class is a subclass of SFRegistry, itself a wrapper around API functions that manage the Registry.

TestODBC.scx is a sample form that displays the DSNs available on a machine and shows the name of the driver and database the selected DSN specifies. The combobox is filled with data sources by calling the GetDataSources method of SFRegistryODBC (done in the Init of the form). When a DSN is selected from the list, the InteractiveChange method of the combobox calls the GetDataSourceProperty of SFRegistryODBC to display the desired settings.

Getting localization settings

To make your application more widely applicable, you should avoid hard-coding certain settings that change from one country to the next: currency and date formats, default paper size, month and day names, and so on. While you can look for the current values of those things in the Control Panel Regional Settings applet, you can also read them programmatically using the GetLocaleInfoEx API function. In addition to the current settings, you can also read the settings for any "locale" (a combination of language and country, expressed as two-letter ISO codes separated with a hyphen, such as "en-CA" for English in Canada or "de-DE" for German in Germany). GetLocaleInfo.prg is a wrapper for this function; pass it the setting to look up (use the constants in GetLocaleInfo.h for the desired setting) and optionally the locale to use (if it isn't passed, the user's locale is used). Run TestGetLocaleInfo.prg to see an example.

```
lparameters tnSetting, ;
    tcLocale
#define ccNULL chr(0)
local lcLocale, ;
    lnLen, ;
    lcBuffer, ;
    lnReturn, ;
    lcReturn
if vartype(tcLocale) = 'C' and not empty(tcLocale)
    lcLocale = strconv(tcLocale, 5) + ccNULL
else
    lcLocale = .NULL.
endif vartype(tcLocale) = 'C' ...
declare integer GetLocaleInfoEx in Win32API ;
    string locale, long type, string @buffer, integer len
lnLen    = 255
lcBuffer = space(lnLen)
lnReturn = GetLocaleInfoEx(lcLocale, tnSetting, @lcBuffer, lnLen)
lcReturn = strconv(left(lcBuffer, 2 * (lnReturn - 1)), 6)
return lcReturn
```

Creating more flexible message dialogs

The MESSAGEBOX() function is great; it uses a standard Windows dialog to display a message, with title, a variety of button sets, and a pre-determined icon, and returns a value based on which button the user clicked. However, you have to create your own form if your button choices fall outside the set of "OK, Cancel," "Yes, No," and variants. Fortunately, Cesar Chalom created MessageBoxEx.prg, a wrapper for numerous API functions that create a messagebox-like dialog but with the ability to specify your own icon and the text for up to three buttons. **Figure 2** shows an example.

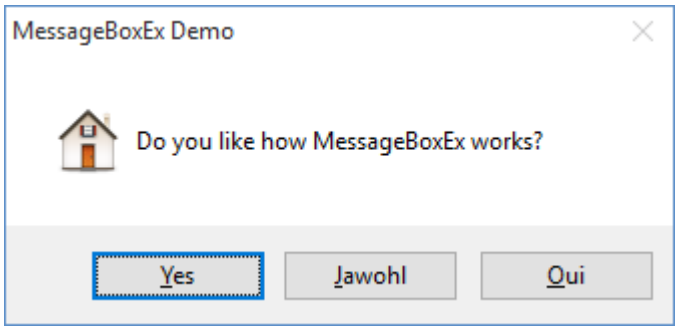


Figure 2. You can use custom buttons and icons with MessageBoxEx.

MesssageBoxEx starts by using an ordinary MESSAGEBOX() call to display the dialog. However, it then grabs the handle of the dialog window and sends messages to the window, via the GetDlgItem, SetDlgItemText, and SendMessage API function, telling it to change the buttons and icon.

To use MessageBoxEx, pass it the text of the message, the icon to use (16 or "x" for a stop sign, 32 or "?" for a question mark, 48 or "!" for a exclamation mark, 64 or "i" for an information icon, or 0 if you're using a custom icon), the title for the dialog, a comma-delimited list of button captions (with "&" to indicate the next character is a hot key), and the path of an ICO file to use for a custom icon. MessageBoxEx returns the number of the button the user clicked.

One additional thing you can do is make the dialog semi-transparent. To do that, add a property to _SCREEN called xmbMessageBoxTransp and set its value to 31 – 255, with 31 being very transparent and 255 being completely opaque (the reason for a minimum of 31 is that anything less than that makes the dialog essentially invisible). I'm not sure why Cesar made this a property of _SCREEN instead of just passing it as a parameter, but that's an easy change to the code if desired.

Other Examples

Here are some other examples I've used:

- 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.
- wwIPStuff, from West Wind Techologies (http://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.

Summary

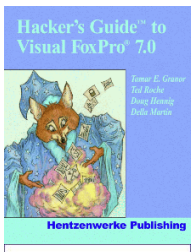
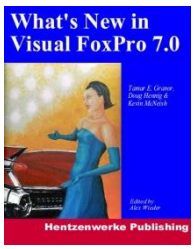
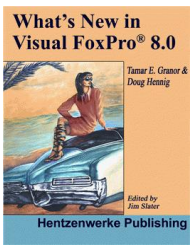
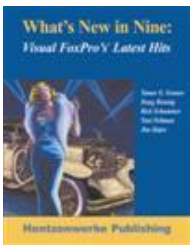
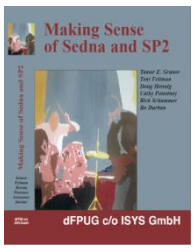
The Win32API is a rich source of services that can extend the functionality of almost any application. It isn't difficult to learn how to use Win32API functions in VFP; the tough part is figuring out what function to use for a given task, how to call that function, and what to do with the return values. Fortunately, this document discusses a number of commonly used functions and presents wrapper programs for them, making them easier to use.

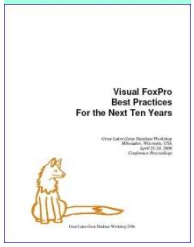
Biography

Doug Hennig is a partner with Stonefield Software Inc. He is the author of the award-winning Stonefield Database Toolkit (SDT); the award-winning Stonefield Query; the MemberData Editor, Anchor Editor, and CursorAdapter and DataEnvironment builders that come with Microsoft Visual FoxPro; and the My namespace and updated Upsizing Wizard in Sedna.

Doug is co-author of *VFPX: Open Source Treasure for the VFP Developer*, *Making Sense of Sedna and SP2*, the *What's New in Visual FoxPro* series, *Visual FoxPro Best Practices For The Next Ten Years*, and *Hacker's Guide to Visual FoxPro 7.0*. He was the technical editor of *Hacker's Guide to Visual FoxPro 6.0* and *The Fundamentals*. All of these books are from Hentzenwerke Publishing (http://www.hentzenwerke.com). He wrote over 100 articles in 10 years for FoxTalk and has written numerous articles in FoxPro Advisor, Advisor Guide to Visual FoxPro, and CoDe. He currently writes for FoxRockX (http://www.foxrockx.com).

Doug spoke at every Microsoft FoxPro Developers Conference (DevCon) starting in 1997 and at user groups and developer conferences all over the world. He is one of the organizers of the annual Southwest Fox and Southwest Xbase++ conferences (http://www.swfox.net). He is one of the administrators for the VFPX VFP community extensions Web site (http://vfpx.codeplex.com). He was a Microsoft Most Valuable Professional (MVP) from 1996 through 2011. Doug was awarded the 2006 FoxPro Community Lifetime Achievement Award (http://tinyurl.com/ygnk73h).


Hacker's Guide™ to
Visual FoxPro® 7.0
Hentzenwerke Publishing

What's New in
Visual FoxPro 7.0
Hentzenwerke Publishing

What's New in
Visual FoxPro® 8.0
Tamar E. Granor &
Doug Hennig
Hentzenwerke Publishing

What's New in Nine:
Visual FoxPro 9 Latest Hits

Making Sense
of Sedna and SP2
dFPUG c/o ISYS GmbH

VFPX
Open Source Treasure
for the VFP Developer
dFPUG c/o ISYS GmbH

Visual FoxPro
Best Practices
For the Next Ten Years

stonefield
Database
TOOLKIT

stonefield
Query

VFPx

MVP
Microsoft
Most Valuable
Professional

FoxRockX