Issue Date: FoxTalk February 1996

Calling Win32 and 32-bit DLL Functions from FoxPro 2.X

Rick Strahl
rstrahl@west-wind.com

Gain Visual FoxPro-like access to Windows internals through this set of library functions that you can use in FoxPro 2.x.

With Windows 95 and Windows NT moving into the mainstream, the new 32-bit implementation of the Windows API -- Win32 -- is becoming increasingly important to developers. This version of the API provides many enhanced operating system features that were previously unavailable. Visual FoxPro, by using the new DECLARE-DLL syntax, has no problem taking advantage of the features and functionality provided within Win32, but FoxPro 2.x doesn't support access to this new 32-bit API since it can't call 32-bit DLLs directly.

Although this lack of support has little effect on FoxPro's internal operation, it does make a difference for some system functionality you might need to provide in your applications. For example, access to the system registry requires use of the Win32 API. Several of the system information functions have 16-bit API counterparts that return incorrect values. You have to call the Win32 versions of those functions if you want reliable results. Finally, Win32 provides a host of useful functionality that previously required add on libraries or DLLs to accomplish. But don't feel left out. In this article I'll describe a DLL that bridges the gap between your FoxPro programs and 32-bit DLLs, using an intermediate translation interface called Call32.

What's with Win32?

Both Windows 95 and Windows NT implement their base operating systems service APIs based on a 32-bit version of the Windows API called Win32. Win32 provides many features that were previously left out of the Windows API and many new features that weren't available or were implemented differently in Windows 3.1. The main reason for the enhanced functionality is that Win32 essentially is *the* programmer's interface to the operating system, replacing the reliance on a separate MS-DOS and BIOS layer. For this reason Win32 provides a wider variety of services that previously were provided only through DOS and BIOS function calls, which meant you had to have low-level access to the hardware and system interrupts using a language such as C.

Win32 has many functions that give you much more control and information about the operating system. For example: Have you ever wanted to set file attributes? Try SetFileAttributes. How about retrieving the name of the current computer under Windows 95 or NT? You can't do it without GetComputerName in the Win32 API. In fact, many network-specific information keys that used to be stored in SYSTEM.INI are no longer stored there under NT and Windows 95 and now need to be retrieved via API calls. How about finding the current, *correct* operating system version? FoxPro's OS() and the 16-bit GetVersion API function return wrong results for both Windows NT and Windows 95, while the Win32 version returns the correct value. The list could go on and on.

The single most important feature that you are likely to need from the Win32 API though is the registry. The registry is Microsoft's replacement solution for the slow .INI file interface for configuration files that store information about system services in a registration database. The registry's database approach allows faster access to the system configuration information making it possible to store a larger number of entries without suffering a performance hit. The registry is supported by 16-bit Windows, but is extremely limited to OLE and file extension registration via the HKEY_CLASSES_ROOT key, which pretty much makes it useless. Only by using Win32 can you retrieve any registry information about the computer and the applications installed on it. The CALL32.DLL I'll present here (available in the accompanying **Download file**), along with the examples, provides read and write access to the registry from your FoxPro 2.x applications.

Using CALL32.DLL with FoxPro

A while back I ran into several problems that required the use of Win32 API calls in order to solve a particular problem. One of the things I needed to do was to figure out the Windows version number correctly for all Windows platforms. The other was reading several values from the system registry. I couldn't find a way to do either using Win16 API calls or any other method I knew of at the time. While searching for a solution, I ran into some public domain C DLL code specific to Visual Basic that allowed VB to call Win32 API functions. With some tweaking of the C code and by creating a pair of FoxPro front end routines, I was able to get the DLL to work under FoxPro 2.x, allowing me to access many Win32 API functions from my programs.

The Call32 DLL interface consists of two DLL functions that work similar to the way Foxtools uses RegFn() and CallFn() by registering the DLL function and then calling it with the function handle that is retrieved. A function called Declare32 registers functions much like the Foxtools RegFn() function does, registering the DLL function by describing the function name and the parameters it uses. The other function named Call32 acts like a router that takes the original arguments and passes them on to the actual 32-bit DLL, translating the parameter types from 16 bit to 32 bit in the process.

When using Call32 from FoxPro you end up having to register the W32 function twice: once for using the Call32 interface and its registration rules, which are slightly different from Foxtools, and once for the actual function that you end up calling with CallFn(). Because of this double registration of each 32-bit function, it's easy to get tangled up in the registration and calling logic, which all together requires five separate lines of code for a single 32-bit DLL call. For this reason I created a pair of front-end routines that simplify the job, leaving the user with only three simple function calls instead of five that require intimate knowledge of the process.

Let's start with an example. The following code calls the SetFileAttributes function in the Win32 API:

```
* Call WIN32 SetFileAttributes
* BOOL SetFileAttributes(lpFileName,dwFileAttributes)
#DEFINE FILE_READONLY 1
#DEFINE FILE_HIDDEN   2
#DEFINE FILE_SYSTEM 4
#DEFINE FILE_NORMAL 128
*** Register 32-bit function with Call32
lhcall32=Reg32("SetFileAttributes",;
               "Kernel32.dll","pi")
*** get a handle for use with Foxtools
lhsetattr=RegFP("CL","L")
*** Now actually call the 32-bit function
*** Note the final parm: Handle from Reg32 call
=callfn(lhsetattr,"Test.txt",file_readonly,lhcall32)
```

The calls the Reg32 and RegFP functions in the previous code are the FoxPro front-end routines that simplify the interface to the actual Call32 DLL functions. In a nutshell, these two functions are responsible for registering the Win32 function, once for the Call32 DLL (Reg32) and once in normal Foxtools fashion using RegFn (RegFP). The final call to the Win32 DLL function is accomplished by using the familiar Foxtools CallFn function with one important addition: The final parameter must be the handle returned from the Reg32 function. As you can see, two function handles are passed to this final call of CallFn() -- the first parameter is to satisfy Foxtools, the last for the Call32 handle.

Reg32 registers the Win32 function with the Call32 interface. It takes the name of the 32-bit DLL function, the DLL it's contained in, and a list parameter types as parameters. As with Foxtools and RegFn, the parameter types are passed as individual characters, which are similar to, but not the same as, those used by RegFn. Reg32 returns a handle to the 32-bit function, which must be used as the *last* parameter of the final function call with CallFn(). Here's the full syntax for Reg32():

```
lh32Handle=reg32(,,)
```

Table 1 contains a list of the parameter types.

Table 1. *Parameter types.*

Parameter	Description
I	32-bit Integer. FoxPro must pass a Long when actually using or returning parameters of this type unless the function explicitly returns a short integer type such as SHORT or BYTE.
P	Pointers. Use this type for *all* string values (even string constants!) and any values that need to be passed by reference.
W	Window handles. Use this type whenever you need to pass a window handle. It automatically translates the 16-bit handle to a 32-bit handle. You can also use this for passing DWORD type parameters and other unsigned integers if a plain integer type fails.

Setting up the parameter types in this step is separate from setting up the parameters used by Foxtools and RegFn in the next step. While Reg32 registers the function with the Call32 DLL, RegFP registers the function with FoxPro using the standard Foxtools interface. RegFP expects the function parameter and return types in typical Foxtools fashion. Here's the syntax:

```
lhHandle=RegFP(,)
```

These are the types you pass and receive from RegFP map to standard Foxtools types, so you can use Long and Character both by value or by reference by pre-pending the variable name with '@'. Keep in mind that in the Win32 API all integers are 32 bit, so usually you must pass them as Longs, unless the API call explicitly calls for a SHORT or BYTE value. RegFP automatically adds a final parameter of type Long to support the required function handle that must be passed as the final parameter when using CallFn().

Once the function is registered, you can now call it using a standard CallFn() call. The syntax for the call looks like this:

```
lvResult=CallFn(lhHandle,Parm1,Parm2,ParmN,lh32Handle)
```

It's very important that the last parameter in the CallFn() statement is the 32-bit function handle that was retrieved with Reg32 in order for the function to work correctly.

Let's take a look at another example. The following retrieves the correct Windows version under Windows 95 or Windows NT. This code uses a set of additional bit shifting functions I added to CALL32.DLL to make sense of the result returned from the GetVersion API call:

```
*** Win32 API call -  INTEGER GetVersion(Void)
lhcall32=reg32("GetVersion","Kernel32.dll","")
lhwinversion=RegFP("","L")
lnversion=callfn(lhwinversion,lhcall32)

*** Large Number shown in Scientific Expression
? "Win32 Getversion result:",lnversion

*** Now decode the version number with
*** bit shifting function provided in CALL32.DLL

*** Result is returned in a single LONG
*** LoWord contains version
*** low byte=Major - HiByte=Minor
lhLoword=regfn("LoWord","L","I")
lhLoByte=regfn("LoByte","I","I")
lhHiByte=regfn("HiByte","I","I")
lnversion=callfn(lhLoWord,lnversion)
lnPlatForm=callfn()
lnmajor=callfn(lhLoByte,lnversion)
lnminor=callfn(lhHiByte,lnversion)
? "Win32 GetVersion (Converted Version):  Major ",;
      lnmajor," - Minor ",lnminor
```

The code starts by registering the GetVersion API call using Reg32 and specifying the parameters types to pass and return. In this case there's no parameter, so the parameter type is passed as a null string. Next the call is registered with Foxtools using the RegFP function, which again shows no parameters and a return type of Long. The actual API call returns an Integer, but remember that 32-bit integers are Longs to FoxPro and Foxtools. Finally, you make the actual call to the API function passing the Foxtools handle and the handle returned from Reg32 using the CallFn function.

GetVersion returns a large Long that is encoded to contain a Windows platform flag and version information. The low WORD (a word is a 16-bit half of a Long or DWORD value) of the returned Long contains the version number, of which the low byte (or half a WORD) contains the major version number with the high byte containing the revision number. In order to decode the version numbers, you need to do some bit shifting in order to get at the individual version numbers. HiWord and LoWord, which take a Long as a parameter, and HiByte and LoByte, which takes an Integer as a parameter, are all contained in the CALL32.DLL file as individual functions that you can use for retrieving individual WORDs or BYTEs from a Long or Integer value. This a common operation for API calls that encode multiple values in a single return value to conserve memory and keep functions compact.

The previous example is provided for demonstration of Call32's functionality only. If you need to get the Windows version number, it would be much easier to use W32Version provided in CALL32.DLL instead. I created this abstracted custom function so that it returns the Windows version number as an integer where the major version is multiplied by 100, adding the revision number to it. To call it use the following code:

```
PROCEDURE WinVersion
lhw32ver=regfn("W32Version","","I","call32.dll")
RETURN callFn(lhw32ver)
```

It returns 400 for Windows 95 and 351 for Windows NT on my machine for example.

How it works

The hard work for the Call32 interface is handled by the code in the C functions contained within CALL32.DLL. The Call32 function performs the thunking and function aliasing that make it possible to call 32-bit functions. If you're interested in the source code for Call32, it's included in the accompanying **Download file**. I can't take credit for the actual Call32 code; Peter Golde created the thunking interface and put the code into the public domain.

Calling the Win32 functions from C is pretty messy, and if you're interested in this, take a look at the abstracted functions that I added in the C program file. W32Version and Read/WriteRegistry both use the Call32 function to provide their functionality.

On the FoxPro end I created the Reg32 and RegFP functions to reduce the number of lines required to make a 32-bit DLL call from five to three and hide the implementation details. The user doesn't need to know how it works, but simply pass the parameters. The only rule to remember is that the final CallFn() call must include the 32-bit function handle as the last parameter.

Here's the code to the Reg32 and RegFP functions:

```
************************************************
PROCEDURE Reg32
*****************
***    Author: Rick Strahl
***  Function: Registers a 32-bit DLL function using
***            CALL32.DLL. Thunk interface
***    Assume: Foxtools is loaded. Uses CALL32.DLL
***      Pass: pcDLLFunction - Name of 32-bit funct
***            pcDLLName     - DLL container
***            pcParmTypes   - Parameter types
***                            I - Integer (FP Long)
***                            P - Pointer
***                                Strings,Reference
***                            W - Handles
***    Return: Function handle that must be used to
***            CALL32 function
************************************************
PARAMETERS pcdllfunction,pcdllname,pcparmtypes
PRIVATE lhcall32,lncall32
lhcall32=regfn("Declare32","CCC","L","CALL32.DLL")
lncall32=callfn(lhcall32,pcdllfunction,;
                pcdllname,pcparmtypes)
RETURN lncall32
************************************************
PROCEDURE RegFP
*****************
***  Function: Registers 32-bit DLL function
***            with Foxtools!
***    Assume: Foxtools loaded, uses CALL32.DLL
***      Pass: phFunction - Function handle provided
***                         via Reg32
***            pcParms    - Parameter types
***    Return: Function Return value
************************************************
PARAMETER pcparms,pcrettype
RETURN regfn("Call32",pcparms+"L",;
             pcrettype,"CALL32.DLL")
```

Accessing the registry

With 32-bit DLL access in place, my next problem was to access the system registry. For those not familiar with the system registry, it is accessed by providing a registry root key (HKEY values if you bring up the registry editor), a key name (which looks like a path "\SOFTWARE\Microsoft\Windows\CurrentVersion") and an entry ("Version") to work with. Registry paths take on a hierarchical structure very similar to DOS file paths, where the files are represented as the actual values stored in an entry. The registry API consists of a set of more than 10 functions, which allow reading and writing of both keys and node values. I don't expect my FoxPro 2.x programs to do much writing to the registry, so I created only basic read and write functions that are described below.

This should be easy now that we can call Win32 API functions, right? Unfortunately, the answer didn't turn out to be quite so easy because there appear to be some problems with Foxtool's use of large long integer values. FoxPro actually uses signed Longs while the registry uses unsigned integers, which causes some problems at the extreme end of the number range for these values. Registry access requires use of very large negative values for the root registry keys and several of these simply wouldn't work when passing them as parameters via CallFn. For example, the key value for HKEY_LOCAL_MACHINE is -2147483646 ((HKEY) 0x80000002). This value causes FoxPro to bomb when calling the RegOpenKey API function directly with CallFn.

I had to use a workaround by creating a wrapper DLL function for RegOpenKey and passing the root registry keys as strings to the intermediate function. The function decodes the string and passes the resulting Long value on to the API function, which returns a key handle. Once this routine was in place, I was able to create the individual registry access function wrappers using the Win32 extensions.

Here's the code for basic registry access wrapper functions:

```
************************************************
PROCEDURE OpenKey
```

```
*****************
***    Author: Rick Strahl
***  Function: Opens a registry key before reading
***            writing entries.
***    Assume: Calls 16-bitRegOpen in CALL32.DLL
***             because of limitations in Longs & HKEY
***      Pass: tcHkey   -  "HKEY_..." strings
***            tcSubkey -  Reg path "\Software\version"
***    Return: key handle
************************************************
PARAMETER lchkey,lcsubkey
lnrethandle=0
lhropen=regfn("RegOpen","CC","L","CALL32.DLL")
lnkey=callfn(lhropen,lchkey,lcsubkey)
RETURN lnkey
************************************************
PROCEDURE CloseKey
*****************
***  Function: Close registry key.
***            Calls 16-bit WinAPI
***      Pass: thHandle - Key handle
************************************************
PARAMETER thHandle
lhClose=RegFn("RegCloseKey","L","L")
RETURN callfn(lhClose,thHandle)
************************************************
FUNCTION querystr
*****************
***  Function: Reads a registry string
***     Notes: Also works with Binary types
***            as long as it doesn't contain
***            NULL values.
***            Calls Win32 API - uses CALL32.DLL
***      Pass: tnHandle - Key Handle
***            tcEntry  - Entry to retrieve
***    Return: string or "" if empty or "*ERROR*"
************************************************
PARAMETER tnhandle,tcentry
PRIVATE lhCall32,lhFP,lcDataBuffer
*** Register function with Win32 and Foxtools
lhcall32=;
  reg32("RegQueryValueEx","ADVAPI32.dll","ipippp")
lhfp=regfp("LCL@L@C@L","L")
*** Return buffer to receive value
lcdatabuffer=SPACE(1024)
lnsize=LEN(lcdatabuffer)
lntype=1         && REG_SZ

lnresult=callfn(lhfp,tnhandle,tcentry,0,@lntype,;
             @lcdatabuffer,@lnsize,lhcall32)
IF lnresult#0
 RETURN "*ERROR*"
ENDIF
IF lnsize<2
 RETURN ""
ENDIF
*** Return string and strip out NULLs
RETURN SUBSTR(CHRTRAN(lcdatabuffer,CHR(0),""),;
       1,lnsize-1)
************************************************
PROCEDURE WriteStr
*****************
***  Function: Writes a key value to an entry.
***    Assume: Calls Win32 API - Uses CALL32.DLL
***      Pass: tnHandle - Key Handle
***            tcEntry  - The entry off the key
***            tcValue  - Value to set it to
***    Return: .T. or .F.
************************************************
PARAMETERS tnHandle, tcEntry, tcValue
PRIVATE  lnSize, lcDataBuffer, tnType
*** Register function with Win32 and Foxtools
lhReg32=;
  Reg32("RegSetValueEx","advapi32.dll","ipiipi")
lhRegFP=RegFP("LCLLCL","L")
```

```
lnSize=LEN(tcValue)
lnType=1   && REG_SZ
lnResult=CallFn(lhRegFP,tnHandle,tcEntry,0,1,;
                tcValue,lnSize,lhReg32)

IF lnResult#0
   RETURN .F.
ENDIF
RETURN .T.
```

Here's an example of how you use these functions (this accesses the Windows 95 registry; you might have to change the keys to make this work under NT):

```
SET LIBRARY TO home()+"FOXTOOLS.FLL"
SET PROCEDURE TO call32
lcroot="HKEY_LOCAL_MACHINE"
lcsubkey="SOFTWARE\Microsoft\Windows\CurrentVersion"
*** Must open the key first
lhreg=openkey(lcroot,lcsubkey)
lcOldSetting=querystr(lhreg,"RegisteredOwner")
? "Old Value: " + lcoldsetting
? "Setting Value: ", writestr(lhreg,"RegisteredOwner","New Owner")
? "Showing New Value: ",querystr(lhreg,"RegisteredOwner")
? "Rewriting old value: ", writestr(lhreg,"RegisteredOwner",lcOldSetting)
? "Showing Old Value: ",querystr(lhreg,"RegisteredOwner")
*** Don't forget to close the key
=closekey(lhreg)
SET LIBRARY TO
```

OpenKey calls the custom function I created in CALL32.DLL in order to work around the Long limitation I mentioned earlier. Both the RegOpenKey and RegCloseKey API calls are available in the Win16 API, so neither one of these actually needs to access Win32. The QueryStr function uses the RegQueryValueEx Win32 API function to read a value from the registry. Once a value is retrieved, the Null is stripped off. If the value can't be found, the function returns "*ERROR*" to differentiate between missing keys/values and an empty ("") value. WriteStr also uses a Win32 API call using the RegSetValueEx Win32 function. If you write to an entry that doesn't exist, it will be created, but only if the key (that is, the directory) exists. The function returns .T. on success or false if it fails.

You can find Integer versions of the Read and Write functions in the accompanying **Download file**. If you plan on working with the registry extensively you'll likely want to add support for adding and deleting keys and values using RegCreateKey, RegDeleteKey, and RegDeleteEntry. I don't foresee using the registry in FoxPro 2.x for much more than simple extraction and occasional value modification, so I haven't bothered to implement them. You can use these registry functions I described earlier as templates.

In addition CALL32.DLL includes a simplified ReadRegistry function to retrieve registry values. It's easier to use for simple registry reads since you don't have to mess with opening and closing registry keys or key handles. But keep in mind that if you read or write multiple entries on the same key, it's faster to open the key then do the reads consecutively, rather than opening and closing the key for each individual access.

Here's the code:

```
************************************************
PROCEDURE rdregstr
*****************
***    Author: Rick Strahl
***  Function: Reads String value from the Registry
***    Assume: Requires CALL32.DLL
***            Works on Binary entries as well
***            as long as no NULLs are part of val
***      Pass: pcRoot  - string registry key value
***                      "HKEY_CLASSES_ROOT"
***                      "HKEY_CURRENT_USER"
***                      "HKEY_LOCAL_MACHINE"
***                      "HKEY_USERS"
***            pcSubKey- Subkey 'path'
***            pcValue -  Actual entry to read
***    Return: key value, "", "*ERROR*"
************************************************
PARAMETERS pcroot,pcsubkey,pckey,pnlength
PRIVATE lcresult,lnlength,lnresult
IF PARAMETERS()<3 OR ;
      TYPE("pcRoot")#"C" OR ;
      TYPE("pcSubKey")#"C" OR ;
      TYPE("pcKey")#"C"
   RETURN "*ERROR*"
ENDIF
```

```
pnlength=IIF(TYPE("pnLength")="N",pnlength,2048)
lcresult=SPACE(pnlength)
*** long PASCAL _export ReadRegistry
*** (LPCSTR cHKey,LPSTR Subkey,
***  LPSTR Value,LPSTR Result,INT Length)
lhreg=regfn("ReadRegistry","C@C@C@CI","L","call32.dll")
lnresult=callfn(lhreg,lcroot,@pcsubkey,;
               @pckey,@lcresult,pnlength)
*** Return only left of Null
lcresult=TRIM(LEFT(lcresult,AT(CHR(0),lcresult)-1))
IF lnresult#0
   RETURN "*ERROR*"
ENDIF
RETURN lcresult
```

You'd read a value like this:

```
lcroot="HKEY_LOCAL_MACHINE"
lcsubkey="SOFTWARE\Microsoft\Windows\CurrentVersion"
? rdRegStr(lcRoot,lcSubkey,"RegisteredOwner")
```

I hope the tools I've presented here are useful to you and help you extend the life of your FoxPro 2.x applications on 32-bit Windows platforms.