Issue Date: FoxTalk November 1999

Using the Windows Shell API to Display a GetFolder Dialog Box

Andrew Coates
a.coates@civilsolutions.com

This month, Andrew Coates presents a technique for displaying the familiar Windows shell interface to allow the user to select a folder.

Awhile back, I presented a technique for converting a path with a mapped drive to its UNC equivalent (see "Un-Mapping Mapped Network Drives" in the December 1998 issue of *FoxTalk*). In that article, I discussed some of the deficiencies of FoxPro's GETDIR() function. In particular, I noted that the dialog box that's displayed doesn't allow the user to select from drives that aren't either local drives or mapped network shares. This month, I'll present a technique that allows you to use the Windows Shell API to display the familiar Windows treeview of the drive hierarchy (like the one that appears in the left-hand pane of Windows Explorer). At the same time, I'll talk a little about using structures and API calls, especially those containing pointers, and I'll introduce a really cool tool for handling structures that was developed by Christof Lange and Mark Wilden.

Getting the folder

FoxPro has a native function that allows a user to select a folder–GETDIR(). The dialog box displayed when the function is called is shown in **Figure 1**.

***Figure 1:* The dialog box that's displayed when calling the FoxPro function GETDIR().**

The problem with this function is that it doesn't allow the user to navigate outside the local drives (or those network shares that are mapped to local drive letters). Increasingly, users need to be able to specify a folder regardless of whether they've got it mapped or not. The Windows shell has an appropriate interface for this–for example, the treeview in the left pane of Windows Explorer. Ideally, the user should be presented with a dialog box like the one shown in **Figure 2**.

***Figure 2:* The dialog box that should be displayed when calling the FoxPro function GETDIR().**

Hunting for a shell

I got the idea for this function from a tip by Scott Lewis (slewis@vcce.com) that I saw in May 1999. At the time, I looked at it longingly, but I saw that the technique used a structure that included pointers to strings and set it aside as something to investigate further "when I had time." More recently, a couple of my clients have been clamoring for the ability to select a network path without having to choose a specific file, so I dusted off the e-mail archive and dug out the original posting.

I still had the problem of how to deal with the requirement of a structure that contains pointers to variable length strings. This isn't something that's been easy to do in VFP. I searched around on the Web for awhile and finally came across a program by Christof Lange and Mark Wilden called struct.vcx. I won't go into a great deal of detail about the workings of this excellent resource (they've done a good job of documenting the tool), but suffice it to say that it does all that I needed to get the job done.

The problem

To start, I'll state the required outcome: "Display a dialog box that allows the user to choose any local folder, mapped folder, or network share with a familiar Windows interface. Return the full path to the folder chosen or an empty string if no folder is chosen."

The solution

The code used to solve the problem is included in the accompanying **Download file**. It's also shown in **Listing 1**.

Listing 1. Calling the Shell API's SHBrowseForFolder dialog box from within VFP.

```
* Program....: GETFOLDER.PRG
* Version....: 1.0
* Author.....: Andrew Coates
* Date.......: September 3, 1999
* Notice.....: Copyright © 1999 Civil Solutions
*   All Rights Reserved.
* Compiler...: Visual FoxPro 06.00.8492.00 for Windows
* Abstract...: Displays a Windows SHBrowseForFolder()
*   dialog box and returns the path selected by the user.
* Receives...: Optional prompt to display at the top
*   of the dialog box and optional handle to the
*   calling window.
* Returns....: String of path chosen (no trailing \)
* Requires...: struct.vcx (including convert.fll)
* Changes....:
lParameters tcPrompt, tnhWndOwner
* Make sure the struct class is available
if ! 'struct' $ lower(set('classlib'))
  set classlib to struct addi
endif
* These #DEFINEs allow you to control what's displayed
* and what's returned. See the Shell API documentation
* for a full description.
* For finding a folder to start document searching
#DEFINE BIF_RETURNONLYFSDIRS 0x1
* For starting the Find Computer
#DEFINE BIF_DONTGOBELOWDOMAIN 0x2
#DEFINE BIF_STATUSTEXT 0x4
#DEFINE BIF_RETURNFSANCESTORS 0x8
* Browsing for Computers
#DEFINE BIF_BROWSEFORCOMPUTER 0x1000
* Browsing for Printers
#DEFINE BIF_BROWSEFORPRINTER 0x2000
* Browsing for Everything
#DEFINE BIF_BROWSEINCLUDEFILES 0x4000
* Used to initialize a buffer passed by reference
#DEFINE MAX_PATH 260
* API declarations
* Frees memory allocated to the ID list returned by
* SHBrowseForFolder
Declare CoTaskMemFree in ole32.dll INTEGER hMem
* Displays the browse dialog box
Declare Integer SHBrowseForFolder in shell32.dll ;
  STRING lpbi
* Decodes the path from the ID number returned from
* SHBrowseForFolder
Declare Integer SHGetPathFromIDList in shell32.dll ;
  INTEGER pidList, STRING @ lpBuffer
* Make sure the location of convert.fll is in the path
if ! './FLLS' $ upper(set('path'))
  local lcPath
  lcPath = set('path')
  lcPath = lcPath + iif(empty(lcPath), '', ';') + './flls'
  set path to &lcPath.
endif
* Get the parameters into local variables with
* sensible default values
local lnhWndOwner, lcPrompt
if vartype(tnhWndOwner) # 'N'
  lnhWndOwner = 0
else
  lnhWndOwner = tnhWndOwner
endif
if vartype(tcPrompt) # 'C'
  lcPrompt = 'Locate'
else
  lcPrompt = tcPrompt
endif
* Declare the local variables used to call the API
local iNull, lpIDList, lResult, sPath, udtBI
iNull = 0
* Create a struct-based class (defined below)
local loBrowseInfo
loBrowseInfo = createobject('Browseinfo')
* Set the parameters of the structure
```

```
With loBrowseInfo
  .hWndOwner = lnhWndOwner
  .lpszTitle = lcPrompt
  .ulFlags = BIF_RETURNONLYFSDIRS
EndWith
* Convert the structure to a string
udtBI = loBrowseInfo.GetString()
* Call the API
lpIDList = SHBrowseForFolder(udtBI)

sPath = ''
* If something was returned
If lpIDList # 0
  * initialize a buffer for the decoded path
  sPath = repl(chr(0), MAX_PATH)
  * decode the path
  lResult = SHGetPathFromIDList(lpIDList, @ sPath)
  * free the memory allocated to the IDList
  =CoTaskMemFree(lpIDList)
  * strip the trailing chr(0)s
  iNull = At(chr(0), sPath)
  If iNull # 0
    sPath = Left(sPath, iNull - 1)
  EndIf
EndIf
* Return the decoded path
return sPath

DEFINE CLASS BrowseInfo as Struct
* This is the C Struct definition
*!*  typedef struct _browseinfo {
*!*    HWND hwndOwner;
*!*    LPCITEMIDLIST pidlRoot;
*!*    LPSTR pszDisplayName;
*!*    LPCSTR lpszTitle;
*!*    UINT ulFlags;
*!*    BFFCALLBACK lpfn;
*!*    LPARAM lParam;
*!*    int iImage;
*!*  } BROWSEINFO, *PBROWSEINFO, *LPBROWSEINFO;
hWndOwner = 0
pIDLRoot = 0
pszDisplayName = 0
lpszTitle = ''
ulFlags = BIF_RETURNONLYFSDIRS
lpfnCallback = 0
lParam = 0
iImage = 0
cMembers = "HWND l:hWndOwner, " + ;
  "LPCITEMIDLIST l:pIDLRoot, " + ;
  "LPSTR l:pszDisplayName, " + ;
  "LPCSTR pz:lpszTitle, " + ;
  "UINT ul:ulFlags, " + ;
  "BFFCALLBACK l:lpfnCallback, " + ;
  "LPARAM l:lParam, " + ;
  "int l:iImage"
ENDDEFINE
```

After a standard header, the program receives two optional parameters–a string that's to be displayed at the top of the dialog box (see **Figure 2**) and a handle to the calling window. It then ensures that the struct class library (more about this later) is loaded. I've included the #DEFINEs inline in the code rather than in a separate header file. The complete list of these constants and their effect on the dialog box is in the MSDN documentation. On the MSDN CD, search for SHBrowseForFolder, and then click on the BROWSEINFO structure hyperlink.

Next, three API functions that will be used are DECLAREd. I'll discuss their use when they're called in the program. The important point to note here is that the SHBrowseForFolder declaration simply specifies a STRING as the parameter passed, in spite the fact that the API definition (from MSDN) specifies that a complex structure needs to be passed. I'll show you how to build that string so that the API gets what it needs a little later on.

The class library we'll use to build the structure needs to be able to find convert.fll, which is a file that ships with the class library. The SET PATH code I've used is one way to make sure it can find it. You might want to use a different technique, but the essential point is that the class library must be able to find the fll or the code will fail. The parameters passed to the program are validated next, and either the values passed or sensible default values are assigned to local variables for use in the rest of the program.

Now the fun begins. To create a string that represents the structure that SHBrowseForFolder requires, I've used the public domain class library called struct.vcx that I mentioned earlier. This allows you to define a class (based on the struct class) with properties for each element of the structure. A members property tells the class what the API expects each member of the structure to contain–integers, fixed length strings, pointers to variable length strings, or almost anything else. I'll discuss this tool's use in some more detail a little later in the article. For now, just follow through the code with me. A class–BrowseInfo–is DEFINEd as descending from the Struct class at the bottom of the program. An instance of this class is created now and non-default properties are set.

Now comes the key point of the process. The GetString() method of the struct-based object is called, and this returns a string that represents the structure that the API function expects. Behind the scenes, the object allocates memory, gets pointers to strings, and generally hides stuff that I'd rather not know about . The good bit is that it returns the information the API function needs in a format it understands. Objects based on the class also clean up after themselves by de-allocating memory, so you won't get any memory leaks if the object is destroyed in the normal way (either by having its last variable reference go out of scope or by being explicitly released).

Once the string representation of the structure has been obtained, it's passed to the SHBrowseForFolder API function, which displays the dialog box with the appropriate prompt. This function returns an integer, which is actually a pointer to an ID number in a list of folders. Fortunately, the API also provides a way to decode the path into a string, given the ID. To do that, allocate a buffer of CHR(0)s into which the path can be placed. Pass the ID and the buffer (by reference) to the API function SHGetPathFromIDList, and the string representation of the path will be placed in the buffer.

Here's another key point. The documentation for the SHBrowseForFolder function states that "The calling application is responsible for freeing the returned item identifier list by using the shell's task allocator." This means that you need to explicitly free the memory used by the function once you're finished with the list. To do this, call the CoTaskMemFree API function with the ID. This will ensure that your application won't "leak" memory by failing to release it back to the available pool once it's finished with it.

All that's left to do is strip the path out of the buffer of CHR(0)s and return it to the calling program. Note that if the user pressed Cancel in the Browse dialog box, the API function returns 0 and the path just remains an empty string.

Structural engineering

The last part of Listing 1 is a class definition. This is the class that's used to create the string that represents the structure required by the API function. I'd encourage you to carefully read the excellent documentation provided with the class to gain a full understanding of how to use it. I'll just touch on a couple of points that arise from the particular implementation shown here.

First, I've included the full C++ definition of the structure as a comment. This gives you a point of reference when defining the properties of the class, as well as providing a link back to the original documentation for the structure.

Each element of the structure is represented by a custom property of the class. In this case, there are eight elements. Each property is given a default value. The last property definition is crucial. The cMembers property is defined in the struct base class, and it provides the class with the information it needs to create the string passed to the API function. Each custom property is listed with its type in the form:

[Description] :

specifies the C type expected by the API function for that element. The codes used in Listing 1 are l (long integer), pz (pointer to a variable length string), and ul (unsigned long integer).

specifies the name of the custom property in the class that corresponds to that element.

The [Description] is an optional component for use by the developer. It's ignored by the class in the construction of the string, and the documentation suggests that you use it to display the C type defined in the structure.

Conclusion

As you can see from **Figure 2**, the problem has been solved.

Even if some of the native VFP dialog boxes are a little out of date, the Windows API provides the interface calls you need to make your application look and feel like a true Windows program. Using the struct class allows you to pass complex (and even nested) structures where required. Working out how to call the API from VFP, even when it was designed for use by C++ programmers, is relatively simple once you know where to look and have the tools to make it happen. Make the effort to get up the initial learning curve. It's well worth it because it gives you another rich set of tools that help make your programming life easier.