Beating the System: Surfing Explorer's Namespace

by Dave Jewell

Wonderful though Delphi is, there are some curious omissions in the functionality which Inprise provide, and that's true even in Delphi 4. One such omission occurs in the `Dialogs` page of the Component Palette. Delphi faithfully provides VCL wrappers for much of the functionality in COMDLG32.DLL, the Common Dialogs library, but there's no dialog for browsing directories. Yes, there are dialogs for opening and saving files, but if you simply want to provide the user with a means to choose a directory (a frequent requirement, particularly important when creating installers), then there's nothing to help you.

In fairness to Inprise, the lack of directory browsing functionality in Delphi is simply a reflection of its absence in the Common Dialogs library, the code just isn't there. But the necessary code *does* exist in the SHELL32.DLL library, a large file that contains most of the guts of the Windows Explorer itself. The necessary API routine is called `SHBrowseForFolder`, and in this article I'll create a new component which makes it easy to browse not only folders, but other areas of the Explorer's namespace as well.

Although the `SHBrowseForFolder` routine is defined in the SHLOBJ.PAS file that ships with Delphi 3, Inprise never actually made use of it in Delphi 3. They did quietly implement a new Delphi 4 routine, `SelectDirectory`, which is hidden away inside FILECTRL.PAS. This routine does make use of the `SHBrowseForFolder` call and, as the name suggests, allows you to easily prompt the user for a directory, displaying the standard Windows directory selection dialog in the process. (Just to add a little confusion here, you should note that Inprise couldn't resist trying out Delphi 4's ability to overload functions with different parameter signatures! You'll find that FILECTRL.PAS contains *two* routines called `SelectDirectory`, only one makes use of `SHBrowseForFolder`).

The `SelectDirectory` routine is very simple to use. You can invoke the standard Windows directory browser dialog using no more code than this:

```
var Dir: String;
begin
  SelectDirectory('Select a '+
    ' directory for Wombat 1.3',
    '', Dir);
```

This will produce the effect shown in Figure 1, returning the selected directory in `Dir`. Like I said, `SelectDirectory` is an easy to use routine, but you pay a heavy price in terms of a lack of features and flexibility. For example, the underlying Windows API call includes the necessary functionality to notify your application whenever the folder browser changes from one directory to another, and this just can't be done through the `SelectDirectory` call. Moreover, `SelectDirectory` will, as the name suggests, only allow you to select directories. It knows nothing about the 'wide blue yonder' of the Explorer namespace.

Understanding SHBrowseForFolder

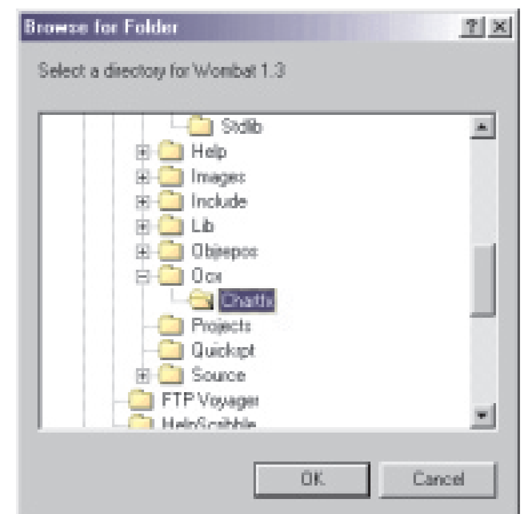
In order to fully exploit the power of `SHBrowseForFolder`, we need to take a closer look at the various bells and whistles it contains. The function prototype for the API call is given below:

```
function SHBrowseForFolder(var
  lpbi: TBrowseInfo): PItemIDList;
  stdcall;
```

As you can see, it takes a single argument: a pointer to a record of type `TBrowseInfo`, and returns another pointer which specifies the location of the selected folder within the Explorer's namespace. Namespace? I've already mentioned the Explorer namespace a couple of times. If it sounds like gobbledegook to you, just bear in mind that `SHBrowseForFolder` isn't specific to disk directories, it can also be used to browse your printers, fonts, Favourites folder, Recycle Bin, even the Network Neighbourhood. Taken together, all these things constitute the Explorer namespace.

Listing 1 shows what the associated `TBrowseInfo` record looks like. This record tells `SHBrowseForFolder` what we want to do, and also returns some important information to us.

The first field here is `hwndOwner`. As with any dialog, it's important



➤ *Figure 1: This is the basic functionality provided by the Inprise SelectDirectory call. You can alter the contents of the text label above the tree-view control, but that's about your lot...*

to provide a handle to a window which 'owns' the dialog and the SHBrowseForFolder dialog is no exception. The second parameter, pidlRoot, is a pointer to a location in the Explorer namespace which represents the starting point for the browse operation. If you set this field to Nil it's interpreted as the root of the entire namespace hierarchy, ie the Windows desktop. If you want to use some other value (such as the Favourites folder), then you need to make use of another shell routine SHGetSpecialFolderLocation. This routine takes a simple integer constant and converts it into a corresponding PItemIDList which can then be stored in the pidlRoot field. The possible integer constants include those shown in Listing 2 and lots more besides, you can find them all in the SHLOBJ.PAS file.

The pszDisplayName field is a pointer to a string containing the returned name of the selected item. Bear in mind that the SHELL32.DLLlibrary won't allocate storage for this item for you. Instead, you must allocate a buffer and save a pointer to this buffer into the pszDisplayName field. You'll then be able to read the string when the function returns. Also remember this display name is only the final part of a pathname: if you want the fully qualified pathname some more magic is required, as we shall see. For this reason, I don't make use of the pszDisplayName in this month's code.

As you'd expect, the lpszTitle field is used to specify the title string for the browse operation. If you look back to the code fragment where I've used SelectDirectory, and then compare this with the corresponding screenshot (Figure 1), you'll see that the designated string has appeared just above the dialog's tree-view control. This corresponds to the string passed through the pszDisplayName field. Surprisingly, Microsoft's interface to SHBrowseForFolder doesn't include the necessary functionality to modify the actual dialog window caption, but it turns out to be a simple matter to do this, and you'll shortly see how I've worked around this limitation.

```
TBrowseInfo = packed record
  hwndOwner: HWND;               { Window handle of dialog box owner          }
  pidlRoot: PItemIDList;         { Root folder from which to browse           }
  pszDisplayName: PAnsiChar;     { Return display name of item selected.      }
  lpszTitle: PAnsiChar;          { text to go in the banner over the tree.    }
  ulFlags: UINT;                 { Flags that control the return stuff        }
  lpfn: TFNBFFCallBack;          { Pointer to callback routine                }
  lParam: LPARAM;                { extra info that's passed back in callbacks }
  iImage: Integer;               { output var: where to return the Image index. }
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
csidl_Recent      =    $0008;     { The Recent documents list }
csidl_Favorites   =    $0006;     { The Favourites folder }
csidl_Programs    =    $0002;     { The Program folder }
```

The next field is ulFlags. This is a series of bit flags which enables us to modify certain aspects of the browser dialog's behaviour. As an example, if you set the bif_BrowseIncludeFiles flag (defined in SHLOBJ.PAS), the dialog will allow you to browse right down to the level of individual files, making the dialog work like a file picker to select an existing file for some purpose. Similarly, setting the bif_StatusText will slightly reduce the height of the dialog's tree-view control, making room for an additional label control within the dialog. Again, we'll return to this.

The lpfn field is used to provide a call-back function whereby the SHELL32 library calls the application in two different situations, firstly, when the dialog has been initialised and, secondly, when the current folder selection changes. This gives us a chance to get 'in on the action' rather than waiting for the dialog to be dismissed. The next field, lParam is for the use of the application, we can put any value we like into this field and it will be passed back to us in the call-back function. See the boxout *Implementing Callback Routines In Delphi* for some general guidelines on how to make use of this. iImage is the final field in the TBrowseInfo record. This is an index into the system image list, and once again, it's something that we'll be covering in more detail later.

Introducing TShellBrowser

Armed with the above information, we can write a non-visual component which, unlike SelectDirectory, gives you access to the full power of SHBrowseForFolder. I decided to call my component TShellBrowser, for the simple reason that, as pointed out earlier, SHBrowseForFolder will let you do more than simply browse directory folders. The complete source code to this component is given in Listing 3. The code has been tested under Delphi 3 and 4. I've tried to make this component work as much like the existing Delphi common dialogs as possible. Thus, there's an Execute method which caused the shell browser dialog to be displayed on the screen. If the user makes a selection then True is returned from this method, otherwise the return value is False. In the former case, the FolderPath property will contain the path which was selected by the user.

The Domain property specifies what part of the Explorer namespace is to be browsed while the Options property can be used to set up which options will be passed to the SHBrowseFolder call in the ulFlags field of the TBrowseInfo record. Perhaps the most interesting flag here is IncludeStatusText. If this flag is set then you'll see an additional label control appear in the Browse dialog, see Figure 2. If you carefully compare Figures 1 and 2, you'll see that there's a second, status label below the title label. With the code presented in this article, the default behaviour of my component is to fill this label with the pathname of the currently selected item. This will happen 'for free', you don't need to add any more code to make it work.

However, if you want to get in on the action, I've also added an

OnSelectionChanged event to the component. Using this event, you can arrange for the TShellBrowser component to notify you whenever the user changes the current selection. Moreover, you've got an opportunity, within the event handler, to accept or reject the selection change. The default behaviour is to accept the selection change, but if you reject it, then the browser's OK button will be disabled. At the same time, the proposed status label text is passed to the event handler, and you can modify this too, if you wish. Thus, instead of displaying the path name, you could display the modification date, byte size, or any other attribute of the selected item. As it stands, my code will assume that you're passing back a file path and, if the file path won't physically fit into the status text label, then it is truncated in a 'nice' manner, by replacing one or more intermediate directories in the path with ellipsis. You can see this behaviour clearly in Figure 2.

The uppermost label can be modified through the LabelTitle property and you can centre the browser dialog on the screen using the Centred property. If you wish, you can specify an initial folder location for the browse operation using the StartDirectory property and you can even change the caption of the dialog box itself with the WindowTitle property. Needless to say, relatively few of these features are available through the raw SHBrowseForFolder interface. It's nice being able to wrap up Microsoft's clunky API calls into nice, reusable Delphi components, but if you can add to the existing functionality at the same time, then so much the better!

OK, so how does it work? At the beginning of Listing 3, I've defined a few data types which correspond to the custom properties required by the browser component. This includes the TShellDomain type and TBrowseOptions for the Options property. Because this is a non-visual component, it's derived from TComponent. No work is done in the component's constructor other than setting properties to their default values.

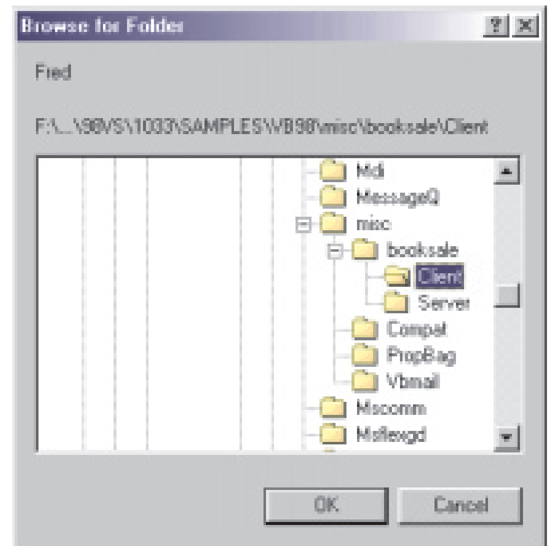


Figure 2: This screenshot shows how it's possible to add an additional status text label. Here, you can see how I've 'massaged' the status text (using MinimizeName in FILECTRL.PAS) so it fits the available space.

The real fun starts in the Execute method. First and foremost, it's necessary to use the ShGetMalloc routine to obtain an interface to the Explorer's so-called task allocator. This is simply a memory allocator/de-allocator routine, buried inside Explorer. We don't actually allocate any memory using this interface but we do need to de-allocate some! Remember I said earlier that SHBrowseForFolder returns a PItemIdList pointer. This is allocated by Explorer on our behalf and it's our responsibility to free it. I've seen one or two shareware components that wrap SHBrowseForFolder which don't bother to do this. You have been warned...

Having got our IMalloc interface, the next job is to fill in the various fields of the TBrowseInfo record. As stated earlier, I didn't use the pszDisplayName field, simply setting it to Nil. (Note if you *do* use this field, conventional wisdom dictates that you should allocate memory for the buffer using the task allocator. That seems to be the way Inprise have done things in the SelectDirectory code). I've used two private functions here, GetFlags and DomainToIDL. These map the Options and Domain properties into the equivalent field values for the TBrowseInfo record. I've also specified that I want to use a callback procedure, BrowserCallbackProc and I've passed the component's instance handle, Self, as the lParam field. Again, see the boxout on implementing Delphi callbacks for a more detailed discussion of this.

Once the SHBrowseForFolder routine has done its stuff, we examine the function result to see if it was successful. If not, the FolderPath property is set to an empty string and False is returned from the Execute method. If successful, the SHGetPathFromIDList routine is called to convert the PItemIDList result into a human-readable string (where *do* Microsoft get all their strange, cruel, API designers from?) which is used to set the FolderPath property. At the same time, we must remember to free pidl using the task allocator interface before returning True for success.

Listing 3 (and facing page)

```
unit ShellBrowser;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs;
type
  TBrowseSelectionChanged = procedure (Sender: TObject;
    var NewFolder: String;  var Accept: Boolean) of Object;
  TShellDomain = ( sdDesktop, sdPrograms, sdControlPanel,
    sdPrinters, sdMyDocuments, sdFavorites, sdStartup,
    sdRecent, sdSendTo, sdRecycleBin, sdStartMenu, sdDrives,
    sdNetwork, sdNetHood, sdFonts );
  TBrowseOptions = ( FileSystemDirsOnly, DontGoBelowDomain,
    IncludeStatusText, ReturnSFAncestors, BrowseComputers,
    BrowsePrinters, BrowseFiles );
  TBrowseOptionSet = set of TBrowseOptions;
  TShellBrowser = class(TComponent)
  private
    fLabelTitle: String;
    fFolderPath: String;
```

```
    fWindowTitle: String;
    fImageIndex: Integer;
    fStartDir: String;
    fReadOnlyStrProp: String;
    fReadOnlyIntProp: Integer;
    fDomain: TShellDomain;
    fCentred: Boolean;
    fOptions: TBrowseOptionSet;
    fSelectionChanged: TBrowseSelectionChanged;
    function DomainToIDL: Pointer;
    function GetFlags: UINT;
    procedure UpdateStatusText (Wnd: hWnd; const Selection:
      String);
  protected
  public
    constructor Create (AOwner: TComponent); override;
    function Execute: Boolean;
  published
    property LabelTitle: String read fLabelTitle
      write fLabelTitle;
    property Centred: Boolean read fCentred
      write fCentred default True;
    property FolderPath: String read fFolderPath
      write fReadOnlyStrProp;
    property WindowTitle: String read fWindowTitle
      write fWindowTitle;
    property StartDirectory: String read fStartDir
      write fStartDir;
    property ImageIndex: Integer read fImageIndex
      write fReadOnlyIntProp;
    property Domain: TShellDomain read fDomain
      write fDomain default sdDesktop;
    property Options: TBrowseOptionSet read fOptions
      write fOptions default [FileSystemDirsOnly];
    property OnSelectionChanged: TBrowseSelectionChanged
      read fSelectionChanged write fSelectionChanged;
  end;
procedure Register;
implementation
uses
  FileCtrl, ShlObj, ActiveX; {ActiveX needed for IMalloc. Sigh...}
procedure CentreWindow (Wnd: HWnd);
var Rect: TRect;
begin
  GetWindowRect (Wnd, Rect);
  SetWindowPos (Wnd, 0,
    (Screen.Width - Rect.Right + Rect.Left) div 2,
    (Screen.Height - Rect.Bottom + Rect.Top) div 2,
    0, 0, swp_NoActivate or swp_NoSize or swp_NoZOrder);
end;
procedure TShellBrowser.UpdateStatusText (Wnd: hWnd; const
  Selection: String);
var
  R: TRect;
  S: String;
  StatusWnd: hWnd;
begin
  // Have we got a status label?
  if IncludeStatusText in fOptions then begin
    // WARNING: Requires carnal knowledge of SHELL32.DLL !
    // If Microsoft change the ID of the status label, the
    // code simply won't be able to trim the text to fit.
    S := Selection;
    StatusWnd := GetDlgItem (Wnd, $3743);
    if (StatusWnd <> 0) and
      IsWindowVisible (StatusWnd) then begin
      // We've got a status window. Should we trim the text?
      GetWindowRect (StatusWnd, R);
      S := MinimizeName(S, Application.MainForm.Canvas,
        R.Right - R.Left);
    end;
      SendMessage(Wnd, bffm_SetStatusText, 0,
        Integer(PChar(S)));
  end;
end;
function BrowserCallbackProc (Wnd: hWnd; uMsg: UINT; lParam,
  lpData: LPARAM): Integer; stdcall;
var
  Accept: Boolean;
  Selection: String;
  Buff: array [0..255] of Char;
  Self: TShellBrowser absolute lpData;
begin
  with Self do case uMsg of
    bffm_Initialized :
    // The initialization call from the browse dialog.
    begin
      // Centre the dialog on screen if fCentred is True.
      if fCentred then CentreWindow (Wnd);
      // Set a custom dialog title if desired.
      if fWindowTitle <> '' then
        SetWindowText (Wnd, PChar(fWindowTitle));
      // Set an initial directory selection if desired
      if fStartDir <> '' then
        SendMessage (Wnd, bffm_SetSelection, Ord(True),
          Integer (PChar (fStartDir)));
      end;
      bffm_SelChanged :
      // This message is received whenever the folder
      // changes in the browser dialog. lParam is a pidl to
      // the newly selected folder.
      begin
        Accept := True;
        // Retrieve the current selection
        SHGetPathFromIDList (PItemIDList (lParam), Buff);
        Selection := StrPas (Buff);
        // Notify application of selection change?
        if Assigned (fSelectionChanged) then
          fSelectionChanged (Self, Selection, Accept);
        // Update status text
        UpdateStatusText (Wnd, Selection);
        // Enable/disable OK button as requested
        SendMessage (Wnd, bffm_EnableOK, 0, Ord (Accept));
      end;
    end;
    Result := 0;
end;
constructor TShellBrowser.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  fCentred := True;
  fOptions := [FileSystemDirsOnly];
end;
function TShellBrowser.DomainToIDL: Pointer;
var
  FolderNum: Integer;
begin
  case fDomain of
    sdPrograms:          FolderNum := csidl_Programs;
    sdControlPanel:      FolderNum := csidl_Controls;
    sdPrinters:          FolderNum := csidl_Printers;
    sdMyDocuments:       FolderNum := csidl_Personal;
    sdFavorites:         FolderNum := csidl_Favorites;
    sdStartup:           FolderNum := csidl_Startup;
    sdRecent:            FolderNum := csidl_Recent;
    sdSendTo:            FolderNum := csidl_SendTo;
    sdRecycleBin:        FolderNum := csidl_BitBucket;
    sdStartMenu:         FolderNum := csidl_StartMenu;
    sdDrives:            FolderNum := csidl_Drives;
    sdNetwork:           FolderNum := csidl_Network;
    sdNetHood:           FolderNum := csidl_NetHood;
    sdFonts:             FolderNum := csidl_Fonts;
    else                 FolderNum := 0;
  end;
  if FolderNum = 0 then
    Result := Nil
  else
    SHGetSpecialFolderLocation (Application.Handle,
      FolderNum, PItemIDList (Result));
end;
function TShellBrowser.GetFlags: UINT;
begin
  Result := 0;
  if FileSystemDirsOnly in fOptions then
    Result := Result or bif_ReturnOnlyFSDirs;
  if DontGoBelowDomain in fOptions then
    Result := Result or bif_DontGoBelowDomain;
  if IncludeStatusText in fOptions then
    Result := Result or bif_StatusText;
  if ReturnSFAncestors in fOptions then
    Result := Result or bif_ReturnFSAncestors;
  if BrowseComputers in fOptions then
    Result := Result or bif_BrowseForComputer;
  if BrowsePrinters in fOptions then
    Result := Result or bif_BrowseForPrinter;
  if BrowseFiles in fOptions then
    Result := Result or bif_BrowseIncludeFiles;
end;
function TShellBrowser.Execute: Boolean;
var
  pidl: PItemIDList;
  ShellMalloc: IMalloc;
  BrowseInfo: TBrowseInfo;
  Buff: array [0..255] of Char;
begin
  Result := False;
  if (ShGetMalloc (ShellMalloc) = S_OK) and
    (ShellMalloc <> Nil) then begin
    BrowseInfo.hwndOwner := Application.Handle;
    BrowseInfo.pidlRoot := DomainToIDL;
    BrowseInfo.pszDisplayName := Nil;
    BrowseInfo.lpszTitle := PChar (fLabelTitle);
    BrowseInfo.ulFlags := GetFlags;
    BrowseInfo.lpfn := BrowserCallbackProc;
    BrowseInfo.lParam := Integer (Self);
    pidl := SHBrowseForFolder (BrowseInfo);
    if pidl = Nil then
      fFolderPath := ''
    else begin
      Result := SHGetPathFromIDList (pidl, Buff);
      fFolderPath := StrPas (Buff);
      fImageIndex := BrowseInfo.iImage;
      ShellMalloc.Free (pidl);
    end;
  end;
end;
procedure Register;
begin
  RegisterComponents ('The X Factor', [TShellBrowser]);
end;
end.
```

Within the callback procedure, two possible messages can be received. The first one is `bffm_Initialized`. This is a signal to say that the dialog has been initialised and is about to be displayed. It gives us a chance to perform any needed customisation of our own. Firstly, I check the `Centred` property, centring the dialog on screen if it happens to be set to `True`. This functionality isn't directly supported by `SHBrowseForFolder`, but as you can see, it's easy to add.

Next, I see if a custom window caption has been specified. If so, the `SetWindowText` routine is called to do the business. Again, this is extra functionality over and above what Microsoft's code supplies. Finally, if the user wants to set an initial directory for browsing, then I make use of the `bffm_SetSelection` message to make this directory the current selection.

The second message is `bffm_SelChanged` which, as the name suggests, is sent whenever the user changes the current selection. I wanted to provide a lot of flexibility here, so I arranged for an acceptance parameter to be passed to the application along with a `var` string giving the path of the selected item. If the application clears the `Accept` variable, then the `OK` button is disabled in the dialog. Equally, any change to the passed path is reflected in the status text that's displayed through the `UpdateStatusText` method.

The `UpdateStatusText` code is, it has to be admitted, sailing a little close to the wind in terms of good programming practice! I wanted to be able to trim the displayed pathname so that it would fit nicely into the status text label. In an ideal world, Microsoft would have done this for you, inside the SHELL32.DLL code. At the very least, they should have programmatically provided a mechanism for an application to query the pixel length of the status label. As ever, you get absolutely no help whatsoever, and you have to do the whole thing for yourself.

After a little dabbling, I discovered that the dialog item ID of the status text label was `$3743`. Armed with this information, it's easy to use standard API techniques to obtain a handle to the text label, query its size and then use the built-in `MinimizeName` routine (it's amazing what little-known goodies are hiding in the FILECTRL.PAS file) to shrink the pathname to fit. I've tried to make this code as 'fail-safe' as possible. If the designated dialog item doesn't exist or if it's not visible for any reason, then the `MinimizeName` routine won't be called, but the unchanged string will simply be passed through the `bffm_SetStatusText` call.

Implementing Callback Routines In Delphi

The more programming I do in Delphi, the more I wonder how I ever managed to stay sane while working with C/C++. At one time, with older C/C++ development systems, you had to use special syntax to declare the callback routine, and you also had to add it to a list of exports in a special definition file. By contrast, writing callback routines in Delphi is so easy; but here's a few hints and tips.

In 32-bit Delphi, the most important thing to do is ensure that the `stdcall` keyword is included in declaration of the routine, as I've done for `BrowserCallbackProc` function. You can't directly use methods of an object as callback routines because an object's `Self` parameter is passed as a hidden parameter to every method. Because the `Self` parameter is absent, this creates a potential problem: how do we establish the 'context' of the object within the callback routine? Or, to put it another way, how do we access our own methods and fields? An obvious solution is to store the object instance in a private `var` which is defined within the `implementation` part of a unit. This will work for us because a programmer is only ever likely to use one instance of `TShellBrowser` at any one time. But more generally, this solution won't work when there are multiple instances of an object in existence.

Fortunately, there's a much more elegant approach, and that's to make use of the `lParam` parameter which is passed to many callback routines. Look at the SDK documentation for `EnumChildWindows`, `EnumDesktopWindows`, `EnumFonts`, etc. You'll see that in each case, a `lParam` argument is passed to the callback routine. By specifying that `lParam` should be equal to our object instance, we can explicitly pass `Self` rather than having it passed implicitly through an ordinary method call. Moreover, because the callback routine isn't a method, the Delphi compiler will happily allow us to define a variable called `Self`, as you'll see from my code. If you employ the `absolute` directive to alias the `Self` variable to the `lParam` argument everything will work as advertised, and if you then write a `with Self` statement, then you can essentially forget about the fact that you're not in a regular method of the object.

Adding the `lParam` facility to API callbacks was a rare stroke of genius on Microsoft's part, although I tend to doubt that they were looking ahead to object oriented programming at the time. By their very nature, strokes of genius tend to be few and far between, and this seems to be particularly true at Redmond. As proof, I offer you some of the more recent API callback routines such as `EnumDateFormats`, `EnumCalendarInfo`, and others. Needless to say, the `lParam` field is conspicuous by its absence.

Conclusions

Well, that's it. `TShellBrowser` makes it very easy to query the user for a directory but it will also let you surf the other domains within the Explorer namespace. At the same time, `TShellBrowser` adds a great deal of extra functionality you won't find in the barefoot `SHBrowseForFolder` interface.

There is, however, one loose end. What is the significance of the mysterious `ImageIndex` property that's exported by my component, and how would you use it? To answer that question, we need to explore the delights of the system image list, and that's exactly what we'll be doing next month!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level work. He is Technical Editor of *Developers Review* which is also published by iTec. You can contact Dave at Dave@HexManiac.com