Beating the System: Managing Desktop Shortcuts

by Dave Jewell

A few months back, *Beating The System* looked at how to implement custom context menus in the Windows 95 and NT 4.0 Shell. The feedback from this article suggested that readers were interested in learning about other ways in which the Explorer can be customised through the various COM interfaces which it provides.

Accordingly, this month we're going to take a look at shell links, otherwise known as shortcuts. As you'll probably appreciate, a big benefit of shortcuts is the way in which they attempt to 'virtualise' the location and name of a target. For example, try right clicking on the Windows desktop and then set up a new shortcut to your Delphi development. On my system, this has a fully qualified pathname of:

```
C:\Program Files\Borland\
  Delphi 2.0\Bin\Delphi32.exe
```

If you temporarily rename the Delphi32.exe file you'll find that clicking the shortcut will still fire up Delphi. This happens because Windows contains built-in operating system hooks which gain control whenever certain operations take place, such as renaming or deleting a file. When a file gets renamed, the shell checks to see if the old file name was referenced by any existing shortcuts. If so, they get transparently modified so that they now refer to the new file name.

Moving a file is handled in a similar way, but sometimes Windows can lose track of a target. We've probably all seen the 'waving torch-light' animation that comes up when you click on a shortcut which references a missing target. Windows can scan the whole disk looking for the missing target and tries to find a match based on name, file type, size and file modification date – but it doesn't always succeed in tracking down the correct file.

Shortcut Implementation

Shell links are implemented using small files with an extension of .LNK. If you've got any shortcuts installed on your desktop, you'll probably find some of these .LNK files if you look in the directory C:\WINDOWS\DESKTOP. This is usually referred to as the desktop folder and is the default place where Windows' logical 'view' of the desktop is stored.

There's a one-to-one correspondence between shortcuts and link files. In fact, if you create a new desktop link using the Explorer and then go into the aforementioned directory, you'll find that a new .LNK file has been created. If you manually delete the .LNK file, you'll then see the shortcut disappear from your desktop after a few seconds. Windows is continually monitoring file I/O within the system and Explorer is continually trying to ensure that the desktop view perceived by the user agrees with the contents of the desktop folder.

In theory, it should be possible to create, destroy and edit links simply by creating, deleting and modifying .LNK files, but this approach assumes that we know the internal structure of a .LNK file, which we don't. In any case, such a strategy would make us vulnerable to future changes in the format of shortcut files. There's absolutely no reason to do this because Microsoft have already provided us with all the functionality we need through a COM interface.

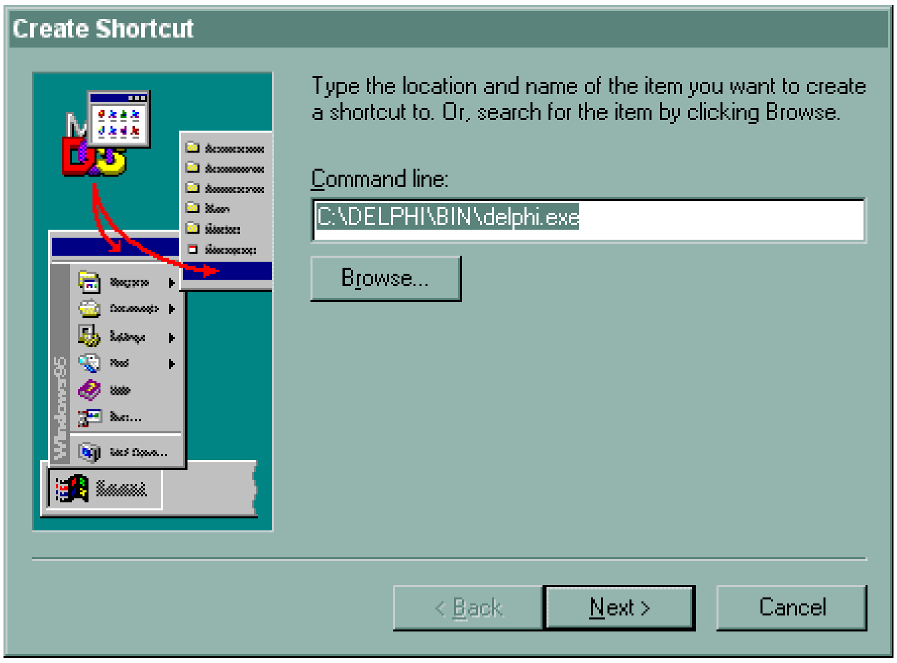
Introducing TShellLink

A .LNK file encapsulates quite a number of different properties relating to a shortcut. These include the full pathname of the target (the executable which we want to start running when the shortcut is triggered), command line parameters, an initial working directory, a description string, a window state (whether you want the application to start off minimised, maximised or whatever), the icon to use for displaying the shortcut and so forth. All this functionality is provided through a COM interface called `IShellLink`. I decided that it would be more fun to do things the Delphi way and package up this functionality into a component which you can place on a form. It's well known that Delphi 3 will allow you to quickly convert an existing VCL component into an ActiveX control which can be used, in a language independent manner, from many other development systems. This fact alone should perhaps encourage you to create components more enthusiastically than you've done in the past!

Having said that, this component is, by its very nature, relatively specialised. It's likely to be used mainly by those who want to write an install program for their software, but it also illustrates some general principles of COM usage from Delphi which apply, needless to say, to the other COM interfaces provided by the shell.

The full source code to the component is provided in Listing 1. As you can see, it's a non-visual component which inherits directly from `TComponent`. In order to create a new shortcut, you simply initialise the six properties to their desired values and then call the `Save` routine. This is rather analogous to (for instance) the `TOpenDialog` component where you set up a number of properties and then call `Execute` to get the actual job done.

Although six properties are provided, you don't need to use them all if you don't want to. For example, assuming that you've already


Create Shortcut

Type the location and name of the item you want to create a shortcut to. Or, search for the item by clicking Browse.

Command line:

C:\DELPHI\BIN\delphi.exe

➤ *Although the Explorer can create shell links under user control, it's important to be able to do this programmatically – especially useful when writing installer applications.*

placed a `TShellLink` component onto your form, you can create a shortcut with as little as three lines of code:

```
ShellLink1.TargetPath :=
  'c:\msoffice\winword\winword.exe';
ShellLink1.LinkPath :=
  'My First Shell Link!';
ShellLink1.Save;
```

In this example, the target executable is set to the Word for Windows executable, the name of the shortcut is set to `My First Shell Link!` and the `Save` function is then called to save the shortcut to disk, creating a new .LNK file. The shell will immediately display the new shortcut on the desktop. By default, my component will always create shortcuts on the desktop, at the top level of the Explorer's 'name space'. However, you can also place a shortcut into an existing folder like this:

```
ShellLink1.LinkPath :=
  'Stuff\My First Shell Link!';
```

This will create the shortcut in the folder `Stuff`.

The six properties give a reasonable degree of control over the shortcut, though I haven't tried to provide the full functionality of the `IShellLink` interface. You can't, for example, set up shortcuts to non-file objects such as printers and you haven't got any control over the icon which the shell uses to display an icon. By default, the shell will use the first icon in the target executable, which is almost always what you want to happen. If you want to extend `TShellLink`, you are of course, free to do so. It should be easy to figure out how to incorporate extra functionality based on the code I've provided.

As a value added bonus, there's a cute bit of extra functionality embedded into `TShellLink`. If you set the `LinkPath` property (either at design time or at run time) to the name of an existing shell link, the other properties will be immediately changed to reflect the contents of the link file. This means that you can use `TShellLink` not only to create new shell links, but also to examine the 'contents' of existing ones.

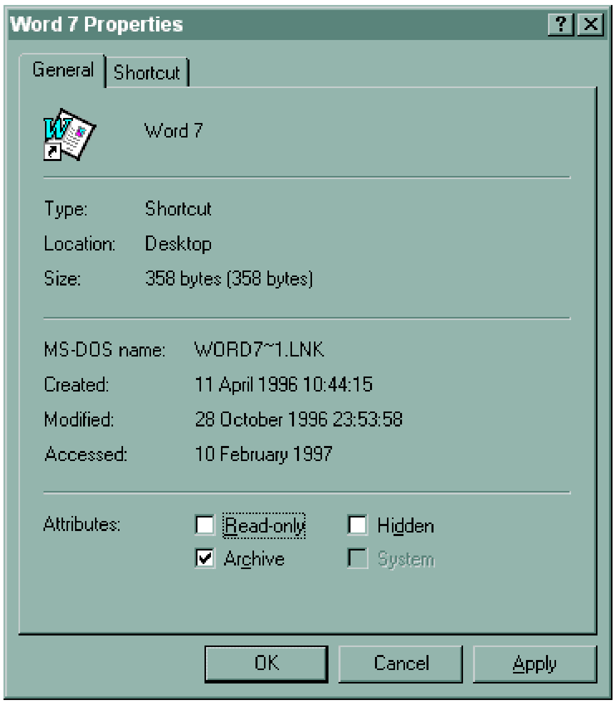
How It Works

In our previous foray into the world of context menus, I deliberately eliminated all the lengthy OLE/COM interface files associated with shell programming so that we could concentrate on the basics of talking to the shell from Delphi without getting bogged down in a lot of unnecessary and superfluous detail. This time round, I've bitten the bullet and made use of the `OLE2` unit (included with Delphi), along with `ShellAPI` and `ShellObj` (which are on this month's disk along with the component source code). These units contain all the CLSID definitions and interface declarations needed to communicate with the shell via COM and undoubtedly we'll be looking at more of these interfaces in a future article.

As with all COM programming, it's important to ensure that the `CoInitialize` routine is called at some point before making any other calls involving the COM library. This is done by the simple expedient of placing the library initialisation call into the component's constructor. Likewise, we call `CoUninitialize` when the object is destroyed. Although, strictly speaking, only one library initialisation call is needed, the COM library internally keeps track of how many `CoInitialize` calls are issued and the library is only released from memory when a matching number of `CoUninitialize` calls have been received. By doing things this way, we really don't care whether or not the host program uses COM/OLE.

The class definition for `TShellLink` is very straightforward and, unlike a lot of components, there are no read/write procedures associated with the published properties. The only exception to this is the `SetLinkPath` routine which is invoked when the `LinkPath` property is changed. We need to get control at this point because, as mentioned earlier, changing `LinkPath` will cause the component to look for an existing link file of the same name, and update the other property values as appropriate. This being the case, interesting things happen only when the `Save` routine is called, or when the `LinkPath` property is altered. We'll look at these two scenarios in turn.

Like the `Execute` method in the VCL code for the common dialog wrappers, the `Save` function



Word 7 Properties

General | Shortcut

Word 7

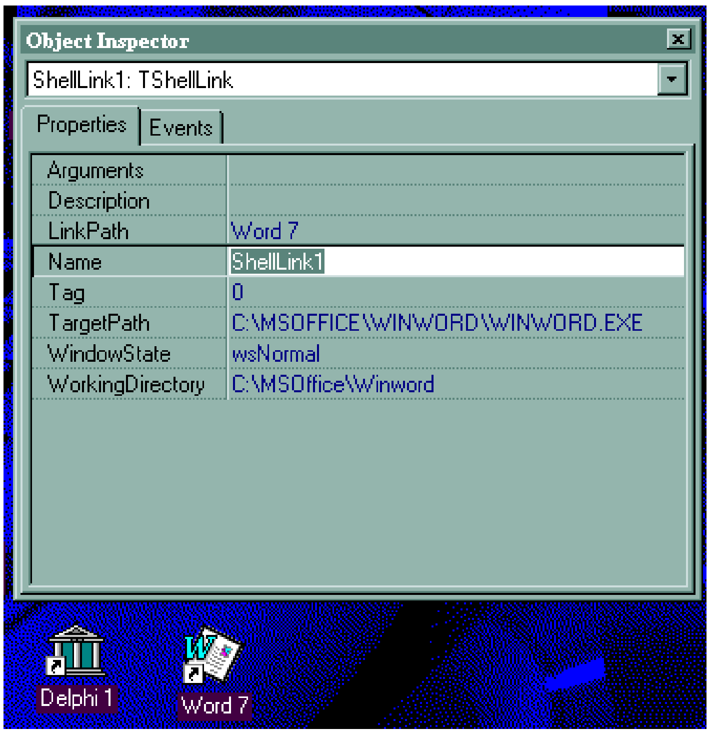
Type: Shortcut
Location: Desktop
Size: 358 bytes (358 bytes)

MS-DOS name: WORD7~1.LNK
Created: 11 April 1996 10:44:15
Modified: 28 October 1996 23:53:58
Accessed: 10 February 1997

Attributes: [ ] Read-only [ ] Hidden [✓] Archive [ ] System

OK | Cancel | Apply

➤ *Figure 2: When working with shell links, you can use the Properties dialog (right-click on an existing shortcut and then select Properties) to verify that you're putting the correct information into your shell link.*



Object Inspector

ShellLink1: TShellLink

Properties | Events

Arguments	
Description	
LinkPath	Word 7
Name	ShellLink1
Tag	0
TargetPath	C:\MSOFFICE\WINWORD\WINWORD.EXE
WindowState	wsNormal
WorkingDirectory	C:\MSOffice\Winword

➤ *Figure 3: Using TShellLink, you can set the LinkPath property to an existing .LNK file, and the component will automatically update its properties based on the contents of the link file.*

returns True for success and False for failure. That said, I don't religiously check the error codes returned from each COM method call. You might wish to tighten things up in this respect. The first job of the Save routine is to call FixupLinkPath. This takes a 'user-friendly' link file name such as My Shortcut and converts it into a fully qualified file name which, in this case, might be something like

```
c:\windows\desktop\
   My Shortcut.lnk
```

The friendly version of the link path is what gets displayed in the Object Inspector at design time, the idea being that the user of the component neither knows nor cares where .LNK files are physically stored, this is all taken care of by the component.

The FixupLinkPath code checks to see if the .LNK file extension is present and, if not, adds it. It then uses the ExtractFileDrive routine as a quick and dirty way of seeing if this is a fully qualified pathname starting from a drive specification. If so, then it's assumed that the component user wants to specify an absolute drive and directory location for the link file. If there's no drive letter, then the GetDeskTopFolder routine is called to bolt the name of the desktop folder onto the front of the link file path.

The reason I don't just hardwire c:\windows\desktop into the code is because the desktop folder location isn't cast in concrete. It's possible to change the folder location by massaging precisely the registry entries that are used by the GetDeskTopFolder routine. Don't do this at home folks! The GetDeskTopFolder code is quite straightforward and simply fetches the desktop folder location from the registry, ensuring that it's terminated with a backslash.

Having called FixupLinkPath to get a full name for the proposed link file, the Save code then calls the somewhat klunky MultiByteToWideChar routine in order to convert the filename into a wide-char (Unicode) string. This is necessary because link files are an example of OLE persistent files, and the persistent file interface insists upon filenames in wide-char format.

In order to simplify the code and provide a couple of reusable routines, I've included two small routines: GetIShellLink, which returns an interface pointer to a new instance of IShellLink, and GetIPersistFile which, given a pointer to an IShellLink interface, returns a pointer to the persistent file interface for that object. Both these routines are called with try-finally blocks to ensure that the Release methods are called for each of the interfaces that we retrieve.

The next job is to copy the current set of component properties into the new shell link object. I've used the same type TWindowState type that the VCL uses to implement a form's WindowState property and arranged for it to default to wsNormal. While this is likely to be adequate for most applications, you may want finer control. If so, you can implement the full set of SW_xxxx codes, as documented in the Windows SDK, and provide a property which gives access to all the possible values.

At this point, the link file doesn't physically exist on disk. When we call the Save method (not our Save

➤ *Listing 1*

```
unit ShellLink;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs;
type
  TShellLink = class(TComponent)
  private
    fTargetPath: String;
    fLinkPath: String;
    fDescription: String;
    fArguments: String;
    fWorkingDirectory: String;
    fWindowState: TWindowState;
    procedure SetLinkPath (const Val: String);
    procedure Resolve (const FullLinkPath: String);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    function Save: Boolean;
  published
    property WindowState: TWindowState read fWindowState
      write fWindowState default wsNormal;
    property TargetPath: String read fTargetPath
      write fTargetPath;
    property LinkPath: String read fLinkPath
      write SetLinkPath;
    property Description: String read fDescription
      write fDescription;
    property Arguments: String read fArguments
      write fArguments;
    property WorkingDirectory: String
      read fWorkingDirectory write fWorkingDirectory;
  end;
procedure Register;

implementation
uses Ole2, ShellAPI, ShellObj;

function GetIShellLink: IShellLink;
//  Create an instance of the IShellLink interface
begin
  if CoCreateInstance(CLSID_ShellLink, Nil, 1,
    IID_IShellLink, Result) < 0 then
    Exception.Create (’Can’’t get a shell link’);
end;

function GetIPersistFile (link: IShellLink): IPersistFile;
// Given an IShellLink, get the IPersistFile interface.
begin
  if link.QueryInterface (IID_IPersistFile, Result) < 0 then
    Exception.Create (’Can’’t get a persistent file’);
end;

function GetDeskTopFolder: String;
// Return location of Explorer’s “live” desktop data
// Yes, we could use SHGetDesktopFolder, but this is simpler
const
  ShellFolders = ’Software\Microsoft\Windows\’+
    ’CurrentVersion\Explorer\Shell Folders’;
var
  Key: hKey;
  bytes: DWord;
  szDest: array [0..Max_Path - 1] of Char;
begin
  Result := ’’;
  if RegOpenKeyEx (HKey_Current_User, ShellFolders, 0,
    Key_Read, Key) = 0 then
    try
      bytes := sizeof (szDest);
      if RegQueryValueEx (Key, ’Desktop’, Nil, Nil,
        @szDest, @bytes) = 0 then begin
        Result := szDest;
        Result := Result + ’\’;
      end;
    finally
      RegCloseKey (Key);
    end;
end;

function FixUpLinkPath (const LinkPath: String): String;
// Convert user-supplied link path into fully qualified path
begin
  Result := LinkPath;
  if Pos (’.lnk’, LowerCase (Result)) = 0 then
    Result := Result + ’.lnk’;
  { Is this a fully-qualified pathname? }
  if ExtractFileDrive (Result) = ’’ then begin
    if Result[1] = ’\’ then Result := Copy (Result, 2, 255);
    Result := GetDeskTopFolder + Result;
  end;
end;
{ TShellLink }

constructor TShellLink.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  CoInitialize (Nil);
  WindowState := wsNormal;
end;

destructor TShellLink.Destroy;
begin
  CoUninitialize;
```

```
  Inherited Destroy;
end;

procedure TShellLink.SetLinkPath (const Val: String);
begin
  if fLinkPath <> Val then begin
    fLinkPath := Val;
    Resolve (FixUpLinkPath (fLinkPath));
  end;
end;

procedure TShellLink.Resolve (const FullLinkPath: String);
var
  swCmd: Integer;
  link: IShellLink;
  persist: IPersistFile;
  FindData: TWin32FindData;
  buff: array [0..511] of Char;
  wLinkPath: array [0..Max_Path-1] of WideChar;
begin
  if FileExists (FullLinkPath) then begin
    { Pathname must be in WideChar format }
    MultiByteToWideChar (cp_ACP, 0, PChar(FullLinkPath),
      -1, wLinkPath, Max_Path);
    { Get a pointer to the wanted interface }
    link := GetIShellLink;
    try
      // First, make sure we can get IPersistentFile
      persist := GetIPersistFile (link);
      try
        // Load the persistent object
        if persist.Load(wLinkPath, stgm_Read) >= 0 then begin
          link.GetPath(buff, sizeof(buff), FindData,
            slgp_ShortPath);
          TargetPath := buff;
          link.GetDescription (buff, sizeof (buff));
          Description := buff;
          link.GetArguments (buff, sizeof (buff));
          Arguments := buff;
          link.GetWorkingDirectory (buff, sizeof (buff));
          WorkingDirectory := buff;
          link.GetShowCmd (swCmd);
          case swCmd of
            sw_Minimize, sw_ShowMinimized:
              fWindowState := wsMinimized;
            sw_ShowMaximized:
              fWindowState := wsMaximized;
            else
              fWindowState := wsNormal;
          end;
        end;
      finally
        persist.Release;
      end;
    finally
      link.Release;
    end;
  end;
end;

function TShellLink.Save: Boolean;
var
  swCmd: Integer;
  link: IShellLink;
  persist: IPersistFile;
  wLinkPath: array [0..Max_Path-1] of WideChar;
begin
  Result := False;
  { LinkPath must be in WideChar format }
  MultiByteToWideChar(cp_ACP, 0,
    PChar(FixupLinkPath(LinkPath)), -1, wLinkPath, Max_Path);
  { Get a pointer to the wanted interface }
  link := GetIShellLink;
  try
    // First, make sure we can get IPersistentFile
    persist := GetIPersistFile (link);
    try
      // Set target and description strings
      link.SetPath (PChar (UpperCase (TargetPath)));
      link.SetDescription (PChar (Description));
      link.SetArguments (PChar (Arguments));
      link.SetWorkingDirectory (PChar (WorkingDirectory));
      case WindowState of
        wsMinimized:  link.SetShowCmd (sw_ShowMinimized);
        wsMaximized:  link.SetShowCmd (sw_ShowMaximized);
        wsNormal:     link.SetShowCmd (sw_ShowNormal);
      end;
      persist.Save (wLinkPath, True);
      Result := True;
    finally
      persist.Release;
    end;
  finally
   link.Release;
  end;
end;

procedure Register;
begin
  RegisterComponents (’Shell Tools’, [TShellLink]);
end;
end.
```

method, but the Save method associated with the IPersistInterface!) the file is written to disk and within a second or two gets 'spotted' by Explorer and added to the desktop.

The operation of the SetLinkPath code is very similar except that, this time, of course, we're reading data from a persistent file. Having checked that the new LinkPath differs from the existing one, the property string is updated, FixupLinkPath is called once more to produce a fully qualified pathname, and the private Resolve method is called to see if the file exists. If so, the Resolve code goes through the same steps as before, converting the filename to wide-char format, and instantiating interfaces to IShellLink and IPersistFile. This time, of course, the Load method is called for the persistent file, and the various fields of the link file are then used to update the component properties.

Bugs!

Although this month's code is (as far as I know!) bug free, any problems that you do encounter will be due, in part, to the fact that the IShellLink interface has a number of idiosyncrasies of its own, some of which I've found, some of which I haven't, and all of which are undocumented!

For example, you'll quickly find that the Description property appears to have no effect whatsoever on the displayed title of the shortcut in Explorer! Contrary to what the Microsoft SDK documentation will tell you, the IShellLink:SetDescription call seems to be resolutely ignored by the shell, although I've verified that the supplied description string is actually being written into the .LNK file.

If you want to give your shortcut a name on the desktop such as Link To Wombat, then you need to ensure that the name of the .LNK file (as specified in the LinkPath property) is Link To Wombat preceded by whatever other folder name you might wish to place the shortcut into. In other words, the user-visible name of a shell link seems to be entirely determined by the name of the .LNK file and has nothing to do with the so-called description string. Ho hum.

You'll also notice that I massage TargetPath with a call to UpperCase immediately before calling IShellLink:SetPath. If you don't do this, you'll find that things definitely won't work as advertised! Without the call to UpperCase, the shell link will appear with a 'My Computer' icon, and double-clicking it will bring up a new Explorer window instead of launching the intended application. I strongly suspect that there's some antique code buried in the shell which can only validate the location of the target executable if the entire pathname is supplied in upper-case. If you find any other bugs, do let me know!

Next Month: Explorer Buttons!

Last month we looked at how to add CoolBars to your application and populate them with Delphi components. The accompanying disk thoughtfully included a shareware component which (in conjunction with the CoolBar) gives an 'Office97-style' look to your programs. I planned to use this component in my own applications and therefore I spent some time examining it in detail... and unfortunately I found a number of very serious problems.

For example, if you use a button colour of anything other than clBtnFace, the control in question won't even draw itself properly. Despite the claim that "The component is very resource friendly", it internally allocates a TTimer component and uses timer events to get control and decide whether or not it should draw itself in an active or inactive state as the mouse is moved in and out of the control: you'll see a great deal of flickering taking place if you hold the mouse down over the control and then drag the mouse (still held down) outside of the control and around the form. Worst of all (and almost unbelievably!) this component allocates a hidden TForm component for every instance of the control that's created! You might think you've only got one form in your application, but if you're using ten of these buttons, you've actually got *eleven* allocated TForms!

I don't enjoy doing a hatchet-job on other people's software, but in this particular case, I feel that I'm more than justified. My father used to say 'If you want a job doing properly, do it yourself' and sadly this adage often proves to be true.

Accordingly, next month we'll look at how to design an Explorer-style button which really is resource-friendly and doesn't require an associated TTimer or TForm lurking in the background.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or 102354,1572 on CompuServe.