Beating the System: Taming The File System, 1

by Dave Jewell

Speaking as a confirmed Delphi zealot, there are precious few aspects of Visual Basic that I long to see in Delphi. A rare exception is the file system library (sometimes also called the FSO object model) which was introduced in Visual Basic 6.0. This object oriented file system library interface allows you to easily navigate around the various drives, folders and files attached to your PC and contains many routines for doing useful work, such as moving files and folders from one place to another. Strictly speaking, the file system library isn't actually part of VB at all. Rather, it's implemented in the Microsoft Scripting Runtime (SCRRUN.DLL) which is implemented as a COM server. By adding the Scripting Runtime to your Visual Basic project (use the `References` option in VB's `Project` menu), the file system API becomes instantly available to your latest VB project.

At this point you're probably thinking 'Aha! Why not simply import the Scripting Runtime COM server into Delphi?' Well, you can certainly do that and it will work, after a fashion. The code fragment in Listing 1 shows how simple it is to exploit the file system library from Delphi 5. In this case, I'm assuming that you've imported the Scripting Runtime into Delphi, installed it as a component on the palette, and dropped a component of type `TFileSystemObject` onto the current form, renaming the component to `FS`.

The code simply accesses the `Drives` property of the file system object, `Drives` being implemented as an OLE collection. The OLE collection has a default member called `Item` (and since this code fragment was written in Delphi 5, the compiler understands about default members) meaning that I can simply refer to my drive C: by using `C` as an index into the collection. This returns a new object reference of type `IDrive`, and accessing the `TotalSize` property of this drive object will give me the total size of the C: partition. Dead simple. In a similar fashion, I could figure out the available drive space, determine if the drive was ready (in the case of a volume that supported removable media), read the disk serial number and so forth.

```
procedure TForm1.FormCreate(Sender: TObject);
var
  Drive: IDrive;
begin
  Drive := FS.Drives ['C'];
  ShowMessage (IntToStr (Drive.TotalSize));
end;
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
Sub ShowDriveList
  Dim fs, d, dc
  Set fs = CreateObject("Scripting.FileSystemObject")
  Set dc = fs.Drives
  For Each d in dc
    ... lots more code here ...
  Next
  MsgBox s
End Sub
```

On the negative side, I still feel that Delphi's relationship with COM isn't quite as cosy as it could be, and that's still the case when you look at Delphi 5. As an example, consider the `Drives` collection that I mentioned a moment ago. Visual Basic programmers have the luxury of being able to easily enumerate the members of an OLE collection because support for doing so is built right into the language, courtesy of the handy-dandy `FOR EACH` construction: see Listing 2.

Unfortunately, this is a luxury unknown to Delphi programmers. If you want to enumerate all the members of an OLE collection in Delphi, a certain amount of rather counter-intuitive code is required, and all the more so when you realise that Scripting Runtime's file system model uses several different collections. In addition to this, there are the usual considerations such as whether or not Microsoft's SCRRUN.DLL library is distributable, whether it exists on the target machine, how to install and register it, and so forth.

For all these reasons and more, I decided to write my own encapsulation of the file system under Delphi. I can't promise you that it has as much functionality as the FSO library (it doesn't) but I can guarantee that it's a lot easier to use, deploy and extend in any way you like.

Introducing The TFileSystem Class

In the Scripting Runtime, you have a single file system object which contains a collection of `Drive` objects, each of which corresponds to a logical drive on your computer. At the next level you have a `Folders` collection which contains a list of `Folder` objects, and hanging off each `Folder` is yet another collection, `Files`, which describes the files contained within each directory.

Although this arrangement is elegant and intuitive, I wanted to take an approach that was simpler to implement, and I didn't want to have (for example) `Folder` objects

lying around in memory that might potentially be invalidated when another process deletes a sub-directory. I therefore decided to 'flatten out' the FSO hierarchy into a single file system object which is highly modal, the `TFileSystem` class being the result. In other words, the contents of the various properties change depending on which drive, directory and file you might be looking at.

Let's start at the top level and work our way down. If you examine the code in Listing 3 you'll see that `TFileSystem` has a property called `Drives`. Because this is an array property, it can't appear in the `published` section of the class definition and has to go into the `public` area as a runtime only property.

As an aside, personally, if I'd been involved with the Object Inspector code in Delphi 5, I would not have bothered with the brain-dead 'property categories' scheme that Borland have come up with. I think it would have been far more useful to come up with some mechanism for displaying array properties in the Object Inspector at design-time. And yes, I would definitely have wanted to see an Object Inspector that's capable of displaying read-only properties! End of diatribe...

`Drives` is a simple array of characters. If you supply it with an index value, it will give you the drive letter corresponding to that particular drive. This is always an upper case character. Thus, an index of zero will correspond to drive A: on most computers, and `A` will be returned as the function result from the 'getter' routine, `GetDriveChar`. If you examine this routine, you'll see that it merely returns a character from the `fDrives` string, a private variable within the `TFileSystem` object. My initial implementation of `fDrives` used a `TList`, but I eventually realised that a string would be even simpler!

Obviously, you need to supply the `Drives` property with a valid index, but how do you know how many logical drives are available? The answer is to use the `DriveCount` property which, internally, doesn't do anything more complex than get the length of the `fDrives` variable. Thus, assuming that you've created a `TFileSystem` object called `FileSys`, you could populate a listbox with available drive letters as easily as this:

```
for Idx := 0 to
  FileSys.DriveCount-1 do
  ListBox1.Items.Add(
    FileSys.Drives[Idx] + ':');
```

If you run this code on my computer, you will end up with a listbox that contains `C:`, `D:`, `E:` and `F:`, the drive letters of my four hard disk partitions.

Hmmm... so what's happened to my floppy drive, my ZIP drive and the CD-ROM drive? The answer is that I designed `TFileSystem` so that, by default, it only references fixed disks, which will most often be the type of disk you're interested in.

Thus, in my case, the `fDrive` string will consist of the letters `CDEF`. You can easily change this default usage through the `DriveTypes` property, which specifies a set of the drive types you're interested in. By default, this has a value of `[fsFixed]`, giving fixed drives only.

I designed things this way so that if, for instance, you're only interested in one particular type of drive (for example CD-ROM drives), then you can set the `DriveTypes` property accordingly and then you know that each of the drives accessible through `TFileSystem` is of the specified type. Conversely, you can set `DriveTypes` to include *all* possible drive types if it makes sense for the application in hand.

Each time that you change the value of the `DriveTypes` property, it's important to bear in mind that the list of available drives will be reinitialised according to which drives fit the specified criteria. Most people only have a single CD-ROM drive on their system, and if you therefore set `DriveTypes` to `[fsCDROM]`, you'll typically find that `DriveCount` will return a value of 1 and `Drives[0]` will correspond to the drive letter of your CD drive.

Another important property here is `DriveLetter`. Once you've set up `DriveTypes` as required, you can choose any available drive from the `Drives` array and set the `DriveLetter` property to 'point' at the required drive. This is what I meant when I said that `TFileSystem` was essentially a modal design. At any time, you can also read from the `DriveLetter` property to see which is the currently selected drive. If you try and set `DriveLetter` to a drive type which doesn't agree with the current value of `DriveTypes`, then the property will not be changed. In other words, if you (for example) specify that you want to work with fixed disks and then try and set `DriveLetter` to the drive letter of your floppy disk, then you'll be politely ignored. As an added convenience, whenever you change `DriveTypes`, the `DriveLetter` property will automatically be initialised to the *first* found drive which fits that type set. Thus, on my system, a value of `[fsFixed]` will set up drive C: as the currently selected drive.

➤ *Listing 3*

```
unit UFileSys;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, Grids, RzLstBox;
type
  TForm1 = class(TForm)
    DriveList: TRzTabbedListBox;
    procedure FormCreate(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
```

```
implementation
{$R *.DFM}
type
  EFileSystem = class (Exception);
  TDriveType = ( fsUnknown, fsNoRoot, fsRemovable, fsFixed,
    fsRemote, fsCDROM, fsRAMDisk );
  TDriveTypes = set of TDriveType;
  TFileSystem = class (TComponent)
  private
    fDriveLetter: Char;
    fSerialNumber: DWord;
    fDriveType: TDriveType;
{ Continued on facing page... }
```

```
{ Continued from facing page }

    fDriveTypes: TDriveTypes;
    fFileSystem, fDrives, fVolumeName: String;
    fTotalSize, fAvailableSpace, fFreeSpace: TLargeInteger;
    procedure InitDrivesList;
    function GetDriveCount: Integer;
    function GetIsReady: Boolean;
    procedure SetDriveLetter (Value: Char);
    function GetDriveChar (Index: Integer): Char;
    function GetUsedSpace: TLargeInteger;
    function GetSerialNumber: String;
    procedure SetVolumeName (Value: String);
    procedure SetDriveTypes (Value: TDriveTypes);
  public
    constructor Create (AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Refresh;
    property Drives [Index: Integer]: Char
      read GetDriveChar;
  published
    // Drive-specific stuff....
    property DriveLetter: Char read fDriveLetter
      write SetDriveLetter;
    property DriveType: TDriveType read fDriveType;
    property IsReady: Boolean read GetIsReady;
    property VolumeName: String read fVolumeName
      write SetVolumeName;
    property FileSystem: String read fFileSystem;
    property SerialNumber: String read GetSerialNumber;
    property SerialNum: DWord read fSerialNumber;
    property TotalSize: TLargeInteger read fTotalSize;
    property FreeSpace: TLargeInteger read fFreeSpace;
    property UsedSpace: TLargeInteger read GetUsedSpace;
    property AvailableSpace: TLargeInteger
      read fAvailableSpace;
    property DriveCount: Integer read GetDriveCount;
    property DriveTypes: TDriveTypes read fDriveTypes
      write SetDriveTypes default [fsFixed];
  end;
{ TFileSystem }
constructor TFileSystem.Create (AOwner: TComponent);
begin
  Inherited Create (AOwner);
  SetDriveTypes ([fsFixed]);
end;
destructor TFileSystem.Destroy;
begin
  Inherited Destroy;
end;
function TFileSystem.GetDriveCount: Integer;
begin
  Result := Length (fDrives);
end;
function TFileSystem.GetDriveChar (Index: Integer): Char;
begin
  Result := #0;
  if (Index >= 0) and (Index < Length (fDrives)) then
    Result := fDrives [Index + 1];
end;
procedure TFileSystem.InitDrivesList;
var
  p: PChar;
  Buff: array [0..255] of Char;
begin
  fDrives := '';
  GetLogicalDriveStrings(sizeof(Buff), Buff);
  p := Buff;
  while p^ <> #0 do begin
    if TDriveType(GetDriveType(p)) in fDriveTypes then begin
      fDrives := fDrives + UpperCase (p^);
      // If this is first, make it current drive by default
      if Length (fDrives) = 1 then
        SetDriveLetter (p^);
    end;
    Inc (p, 4);
  end;
end;
function TFileSystem.GetUsedSpace: TLargeInteger;
begin
  Result := fTotalSize - fFreeSpace;
end;
function TFileSystem.GetSerialNumber: String;
begin
  // Precision specifier in the format string ensures that
  // leading zeroes actrually get printed instead of being
  // silently discarded....
  Result := Format ('%.4x-%.4x', [HiWord (fSerialNumber),
    LoWord (fSerialNumber)]);
end;
procedure TFileSystem.SetDriveLetter (Value: Char);
begin
  Value := UpCase (Value);
  if (Value <> fDriveLetter) and (Pos (Value, fDrives) > 0)
    then begin
    fDriveLetter := Value;
    fDriveType :=
      TDriveType(GetDriveType(PChar(Value+':\')));
    Refresh;
  end;
end;
function TFileSystem.GetIsReady: Boolean;
var
  errMode, FindErr: Integer;
  SearchRec: TSearchRec;
begin
  Result := fDriveType in [fsFixed, fsRemote, fsRAMDisk];
  if not Result then begin
    errMode := SetErrorMode (sem_FailCriticalErrors);
    try
      FindErr :=
        FindFirst(fDriveLetter+':\', faAnyFile, SearchRec);
      try
        Result := FindErr = 2;
      finally
        FindClose (SearchRec);
      end;
    finally
      SetErrorMode (errMode);
    end;
  end;
end;
procedure TFileSystem.SetDriveTypes (Value: TDriveTypes);
begin
  if Value <> fDriveTypes then begin
    fDriveTypes := Value;
    InitDrivesList;
  end;
end;
procedure TFileSystem.Refresh;
var
  Junk: DWord;
  szVolumeName, szFileSystem: array [0..255] of char;
begin
  // Initialise drive-information properties
  if not GetIsReady then
    raise EFileSystem.Create('Drive not ready');
  GetDiskFreeSpaceEx (PChar (fDriveLetter + ':\'),
    fAvailableSpace, fTotalSize, @fFreeSpace);
  GetVolumeInformation (PChar (fDriveLetter + ':\'),
    szVolumeName, sizeof (szVolumeName), @fSerialNumber,
    Junk, Junk, szFileSystem, sizeof (szFileSystem));
  fVolumeName := szVolumeName;
  fFileSystem := szFileSystem;
end;
procedure TFileSystem.SetVolumeName (Value: String);
begin
  if GetIsReady and (fVolumeName <> Value) then begin
    if Length (Value) > 11 then
      Value := Copy (Value, 1, 11);
    SetVolumeLabel(PChar(fDriveLetter+':\'), PChar(Value));
    Refresh; // Ensure fVolumeName reflects reality...
  end;
end;
//--End of TFileSystem component--------------
procedure TForm1.FormCreate (Sender: TObject);
var
  S: String;
  Idx: Integer;
  TotBytes: Double;
  FileSys: TFileSystem;
  function StrDriveType (Typ: TDriveType): String;
  begin
    case Typ of
      fsRemovable: Result := 'Removable';
      fsFixed:     Result := 'Fixed    ';
      fsRemote:    Result := 'Remote   ';
      fsCDROM:     Result := 'CD-ROM   ';
      fsRAMDisk:   Result := 'RAM-Disk ';
      else         Result := '-unknown-';
    end;
  end;
begin
  FileSys := TFileSystem.Create (Self);
  with FileSys do try
    DriveTypes :=
      [ fsRemovable, fsFixed, fsRemote, fsCDROM ];
    for Idx := 0 to DriveCount - 1 do begin
      try
        DriveLetter := Drives [Idx];
        S := DriveLetter + ':' + #9 + VolumeName + #9 +
          FileSystem + #9 + SerialNumber + #9;
        TotBytes := TotalSize;
        S := S + Format ('%n', [TotBytes]);
        SetLength (S, Length (S) - 3);
        S := S + #9 + StrDriveType (DriveType);
      except
        on EFileSystem do S := Drives [Idx] + ':' + #9 +
          '---<not ready>---';
        end;
        DriveList.Items.Add (S);
    end;
  finally
    FileSys.Free;
  end;
end;
end.
```

Retrieving And Setting Drive Information

OK, so we can tell TFileSystem what drive type(s) we're interested in, we can enumerate the available drives, and we can specify which drive we want to work with. What else is available? For starters, the DriveType property returns the type of the currently selected drive. For obvious reasons, this isn't likely to be very informative unless you've set DriveTypes so as to reference more than one possible drive type.

More useful is the IsReady property. This returns a Boolean value according to whether or not the currently selected drive is ready (by the way, in case you haven't noticed, I've modelled these various property names on the corresponding properties in the FSO library). For fixed, networked and RAM drives, this will always return true. However, for drives which take removable media (including CD-ROM drives), the TFileSystem class interrogates the drive to determine whether or not the drive has media loaded.

TFileSystem also has four properties which relate to the size of the currently selected drive and each of these properties returns a TLargeInteger, which, as you may know, is a 64-bit integer value. Why 64-bit integers? Well, sat just behind me is a machine with a 20Gb hard disk! As you'll no doubt appreciate, a 32-bit integer is only capable of 'counting up' to around 2Gb, which means that if you create a partition greater than this, you won't be able to retrieve the size of the partition using ordinary integers, a scenario that's highly likely with today's enormous (but cheap) hard disks.

At this point, I should confess that this code won't work with Delphi 3, the main reason being that Delphi 3 doesn't correctly handle TLargeInteger. To be more precise, the *real* problem is that some of the API declarations in the Delphi 3 version of the Windows unit are just plain wrong. If you carefully examine the Delphi 3 declaration of GetDiskFreeSpaceEx (for example) and compare it with the equivalent declaration in Delphi 4 or 5, you'll see that the former refers to 32-bit Integer arguments, while the more recent declarations correctly declare the arguments as being 64-bit TLargeInteger values. Whilst in practice this isn't much of a problem, it isn't the only issue relating to the GetDiskFreeSpaceEx routine.

GetDiskFreeSpaceEx is needed in order to correctly report the size of partitions over 2Gb in size, but this API routine didn't exist in very early versions of Windows 95. In Delphi 3, you're on your own, and it's up to you to use GetProcAddress to figure out whether or not this routine is implemented inside the KERNEL32 library. Fortunately, those clever souls at Borland have cunningly hijacked the GetDiskFreeSpaceEx routine in Delphi 4 and 5, using some behind-the-scenes code to automatically determine whether or not the API routine exists, and implementing some equivalent code inside SYSUTILS if it doesn't. The bottom line is that Delphi 3 is really looking a bit long in the tooth for this particular task, and I'm going to be lazy and let Delphi 4 and 5 take the strain. If you want to convert the code so as to work with Delphi 3, then look at the Delphi 4 version of SYSUTILS, and in particular, you'll need to take the BackfillGetDiskFreeSpaceEx routine and transplant it into your Delphi 3 application, with suitable code to ensure that it gets called if the 'real' GetDiskFreeSpaceEx routine isn't present in the Windows kernel. *[Or, you could check Brian Long's solution in the Delphi Clinic in Issue 27, which modifies* DiskFree *to use floating point numbers. Ed.]*

The most important of these four properties is TotalSize, which returns the total byte size of the partition. If you look at Figure 1, you'll see the normal Windows Disk Properties dialog. In this case, it's being used to examine my F: drive, and is reporting a total size of 3,070,205,952 bytes. Superimposed over the bottom of this dialog is the message-box output of a small Delphi test program, written using TFileSystem, which is also showing the total size of drive F: Notice that there is complete agreement between the two programs. The message box was constructed as shown in Listing 4.

Yes, it's nice to see that Format doesn't break when you supply a 64-bit argument as the counterpart to a %d format string! Aren't they a clever bunch in Scotts Valley?

The other three 64-bit properties are FreeSpace (whose meaning should be blisteringly obvious), UsedSpace and AvailableSpace. The value returned from UsedSpace is calculated by subtracting the amount of free space from the total size of the drive since, as far as I know, there is no API call to retrieve the used space directly. You might be wondering why there's an AvailableSpace *and* a FreeSpace property, don't they both amount to the same thing?

Usually, yes, but sometimes, no. This distinction is made for the sake of those multi-user network set-ups which allocate disk quotas on a per-user basis. This means that, for example, there might be 20Gb of disk space available on a large server drive, but your quota means that you've only got access to, say, 5Gb. If you're running a standalone machine, then AvailableSpace and FreeSpace will both return the same value, but if you're working with quotas then AvailableSpace will indicate how much free disk space is available to *you*, whereas FreeSpace will return the total amount of free space on the drive. For obvious reasons, you should always go with the AvailableSpace figure because it will always reflect what can be allocated by you.

The code fragment in Listing 4 also introduces another property, VolumeName. This corresponds to the volume label of the currently selected disk and, on a FAT file

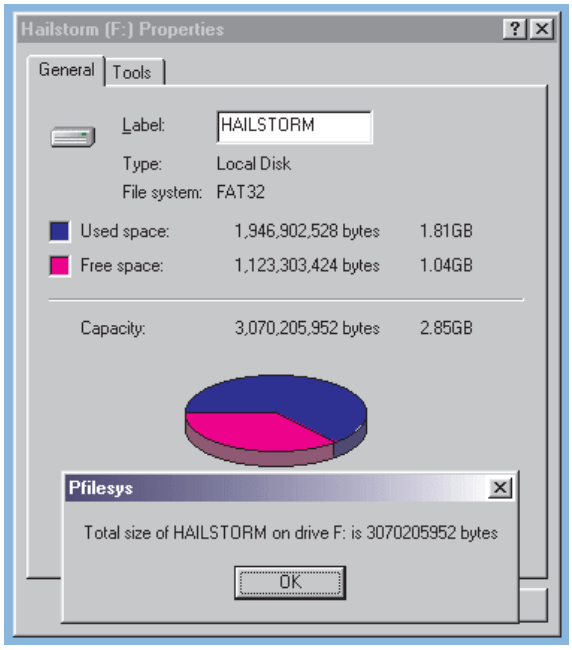
➤ *Listing 4*

```
FileSys.DriveLetter := 'F';
ShowMessage (Format ('Total size of %s on drive %s: is %d bytes',
  [FileSys.VolumeName, FileSys.DriveLetter, FileSys.TotalSize]));
```

system, can never exceed 11 characters in length because Microsoft used the classical 8-character filename with 3-character extension in which to store the label. This is also a writeable property: just assign to the `VolumeName` property and the volume label of the selected disk will be updated accordingly. Do bear in mind, though, that MS-DOS automatically uppercases volume labels, so what you read might not be exactly what you write!

There are three other `TFileSystem` properties that we haven't mentioned so far. All three are read-only properties. The first, `FileSystem`, returns an ASCII description of the file system corresponding to the current drive. `SerialNumber` provides an ASCII representation of the serial number belonging to the current drive. For compatibility with MS-DOS, I've formatted this string into two four-digit groups separated by a hyphen, eg `15F6-0B50`. However, if you want to access the serial number as a single, 'raw', 32-bit quantity (for example, when performing serial number checks in a copy protection scheme), then you can just use the `SerialNum` property instead. Long-standing readers of this column may recall the agonies I went through in Issue 7 when trying to access a drive's serial number from 16-bit Delphi; it's nice being able to access it as a simple property of our `TFileSystem` object. Maybe 32-bit Windows does have its uses after all...

Having described `TFileSystem` from a usage point of view, there's relatively little left to say about the code itself. The only routine worthy of note here is `GetIsReady`, which has the job of checking to see if the currently selected drive is ready in the sense of removable media being present. If the drive type is fixed, remote or RAM disk, then we assume that everything is OK and simply return `True`. For removable media (including CD-ROM drives), the code has to find some way of physically checking for the presence of the disk. One possibility would be to look for a file with a very unlikely name, and another possibility would be to try creating (and then immediately deleting) a file with some randomly generated name. The second option would have the advantage of determining whether or not the media was write-protected. However, I decided to play safe by using a relatively innocuous `FindFirst` routine to check for media present.



➤ *Figure 1: With judicious use of the TLargeInteger type, our TFileSystem component will return exactly the same disk drive metrics as does Windows Explorer itself. However, bear in mind that a little extra work is required if you want to get this working with Delphi 3.*

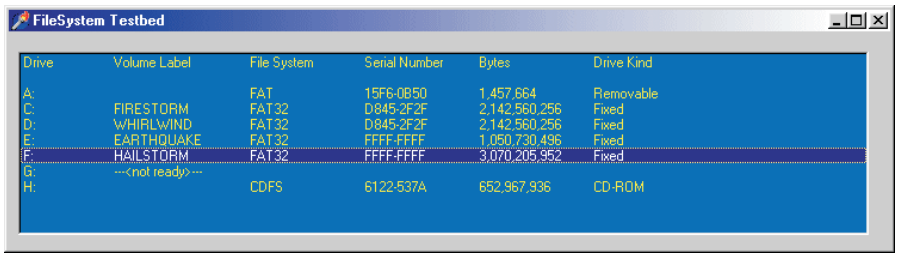
An interesting wrinkle here is that my code always checks for the presence of a file called `X:\` where `X` is the drive letter in question. According to the documentation, `FindFirst` should enable one to check for the existence of a particular directory and, in theory, the root directory should be no exception! However, even if a valid, formatted disk is present, `FindFirst` will return an error code of 2 (file not found) if you specify the root directory! I thought that perhaps this was a strange quirk of Windows 98, but I tried the same code under a beta of Windows 2000 with exactly the same result. So the bottom line is that an error code of 2 (file not found) indicates that the root directory actually *was* found! Any other error code is interpreted as drive not ready. Microsoft, don't you just love 'em...

Next Month

The code in Listing 3 includes the `TFileSystem` component *and* a simple testbed application, which can be seen running in Figure 2. I find that it is often simpler to develop a component in this manner and then separate it from the test application once it's reasonably complete.

Complete project files (for Delphi 4) are included on this month's disk along with a ready-to-run EXE file. You should bear in mind that I used one of the Raize components, `TRzTabbedListBox`, to

➤ *Figure 2: Here's the small testbed program in action. It simply iterates through all the available drives, retrieving various properties for each one. Notice that in this case drive G: (my ZIP drive) is reported as not ready because no media was present when the program executed.*



implement the 'columnised' listbox. Therefore, unless you've got the Raize components, you won't be able to rebuild the application. But in any event the aim of the testbed is to demonstrate use of the `TFileSystem` component.

You'll notice that almost all the implemented properties simply access a 'cached' private variable when reading. This means that you might have a potential problem if another process comes along and (for example) eats a couple of gigabytes of disk space after `TFileSystem` has already decided how much disk space is available. In order to address this issue, I've added a `public` routine, `Refresh`, which should be called just before retrieving any critically important information such as the amount of available disk space. In a similar vein, you'll notice that the `SetVolumeName` routine calls `Refresh` immediately after changing the volume label of a drive, so as to ensure that the cached volume label is consistent with whatever MS-DOS decided to set it to.

One way around this would be hit the disk every time you want to retrieve disk information, and an even better technique would be to implement a file notification whereby Windows itself lets us know when something has changed. This is something that we'll look at next month.

More importantly, next month's instalment will add substantially more flesh onto the `TFileSystem` skeleton, making it possible to examine the folders and files on a specific disk, retrieve information relating to particular folders and files, and browse around within the overall directory structure. So, as Dr.Bob says, stay tuned...

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