Beating The System: Taming The File System, 2

by Dave Jewell

In last month's column, I began the development of a VCL class which encapsulates much of the file system, eliminating the need to endlessly re-invent the wheel when working with files and directories. The emphasis in the first article was at the disk drive level, but this month we'll turn our attention to folders and files.

Folder Navigation Features

In order to simplify file operations from the viewpoint of the application programmer, I added a property called `FolderName` to last month's code:

```
property FolderName: String
  read fFolderName
  write SetFolderName;
```

This property indicates the current absolute folder location, the place where the `TFileSystem` object is 'pointing', so to speak. All file operations are considered to be relative to this location. Thus, if you read the `FolderName` property, you might get back a value of `C:\WINDOWS\TEMP` indicating that you're pointed at this folder.

However, in order to make things even easier to use, I decided to add some shortcut features to this property. For example, if the current `FolderName` is set as above and you want to move up one directory level to C:\WINDOWS, it isn't necessary for the application to figure out the appropriate directory name. Instead, you can just assign the string `..` to the `FolderName` property. When you read `FolderName`, it will then return `C:\WINDOWS` as you'd expect. This makes it very much easier to move up a directory at a time. Obviously, the `..` shortcut will be ignored if the `TFileSystem` object is already pointing at the root directory!

Throughout this discussion, the term 'folder' is synonymous with 'subdirectory' and 'directory' ('folder' is a lot easier to type!).

As another shortcut, setting the `FolderName` property to an empty string is interpreted as moving to the root folder of the current drive. Additionally, if you try setting `FolderName` to `Fred` (for example) then `TFileSystem` will assume that `Fred` is a folder which exists at the current folder level. In other words, if `FolderName` is set to `C:\WINDOWS` and you then try to set `FolderName` to `Fred`, then this will be interpreted as an attempt to set the directory location to C:\WINDOWS\FRED. If no such folder is found, then the assignment to `FolderName` will be ignored.

➤ *Table 1*

Example	Meaning
(Empty string)	Set the root directory on the current drive
.	Set the current directory on the current drive (effectively, a no-op)
..	Go up one directory level if not already at the root
Fred	Go to the directory 'Fred' at the current directory level (current drive)
\Fred	Go to the directory '\Fred' (below root on current drive)
C:\Fred	Go to the directory 'C:\Fred' on designated drive

```
for Idx := 0 to FileSystem.Folders.Count - 1 do begin
  FileSystem.FolderName := FileSystem.Folders [Idx];
  ... more code ...
end;
```

➤ *Above: Listing 1*

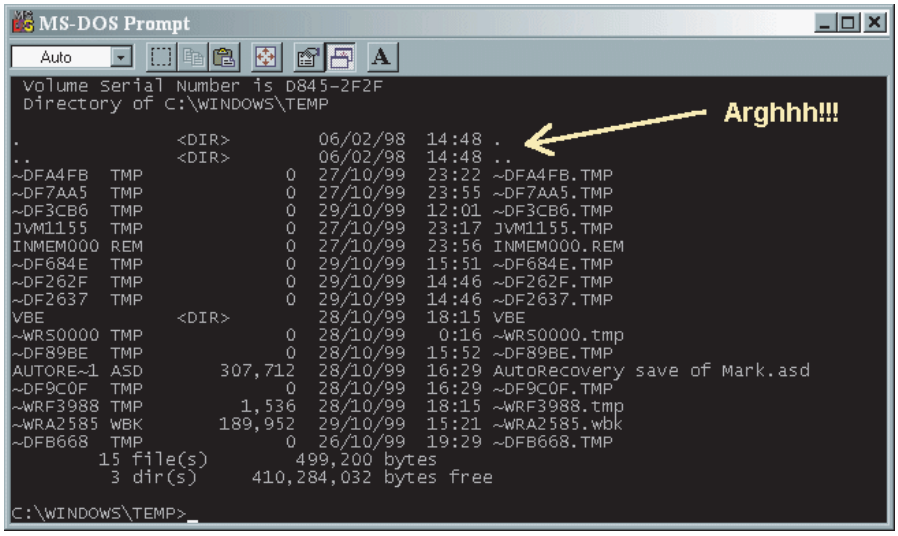
➤ *Below: Listing 2*

```
for Idx := 0 to FileSystem.Folders.Count - 1 do begin
  Str := FileSystem.FolderName + FileSystem.Folders [Idx];
  ... more code ...
end;
```

I've also implemented a `TStringList` property called `Folders` which is automatically filled with all the available folder names at each directory level. This means that when (for example) performing some operation on a group of folders you might be tempted to do something like Listing 1.

This will not work! Assuming that you're at the root directory level, you might have three subdirectories called A, B and C, and these are the directory names that will be placed into the `Folders` list. Once the first assignment to `FolderName` has taken place, the file system component will rebuild the `Folders` property to match whatever folders are located within the A folder, and things will go pear-shaped in very short order. A much better approach would be to do something similar to Listing 2.

Here, we never alter the value of `FolderName` within the loop. Each time round the loop we build a complete path specification from the `FolderName` (which is guaranteed to always end with a backslash) and successive elements of the `Folders` list.



➤ *Figure 1: One of the banes of my life, the '.' and '..' pseudo-directory entries which, thanks to Microsoft's somewhat kack-handed implementation, show up in directory lookup operations, providing an instant gotcha when writing tree-walking directory routines.*

The full list of property assignment possibilities for `FolderName` is given in Table 1.

In the last case, the `TFileSystem` object ends up 'pointing' at a different drive, with consequent changes to the `DriveLetter` (and other drive-specific) properties.

In addition to all the above, there's also the rather thorny possibility that you might try assigning a string such as `C:WOMBAT` to `FolderName`, ie a drive specification with a relative pathname. Hopefully, you wouldn't do this, but, if you did, what's the problem? If the specified drive letter agrees with the current drive, then you can safely interpret it as `WOMBAT`, ie just assume it's relative to the current folder. But if a different drive is specified, then what do we do? Should we assume it's relative to the root of the indicated drive, or relative to the current directory of that drive, or what? In the end, I decided to play safe and ignore such requests if the drive letter differs from the current drive, but it might be better to disallow folder name assignments of that type altogether.

Armed with the above working specification, you can see the implementation of the new `SetFolderName` method in Listing 3. This represents a sort of 'delta' of last month's code listing, showing only the new stuff that's been added since last time. The `SetFolderName` routine is now called from the end of the `Refresh` method, such that the `TFileSystem` object always ends up pointing at the root folder when changing from one drive to another.

`SetFolderName` simply exits if it gets given a folder name comprising a single dot (the current directory), whereas if an empty string is passed, then it's converted to the root folder of the current drive. Similarly, passing the string `..` causes the current folder position to be backed up by one directory level as mentioned previously.

For each of the possible types of path specification passed to the routine, we end up with a single, well-formed (hopefully!) path string of the form `X:\YYYYYY`. The final job of the code is to make sure that the first letter of the path specification is uppercased, and then check that the directory exists. If it doesn't, then the code simply exits. If the directory does exist, then `SetFolderName` ensures that the new folder name terminates with a backslash, and that it's different to the existing folder name. If it is, then the `RefreshFolderAndFileList` routine is called.

The `DirectoryExists` routine is used to check for the presence of a particular directory. I wrote my own `DirectoryExists` routine to do this, and the code is included in Listing 3, it's pretty obvious how it works. Because this is such a useful routine, I made it into a class function so that you can call it without actually instantiating a `TFileSystem` object. One of the most useful (but little known) Delphi units is the `FileCtrl` unit where you'll find another version of `DirectoryExists`: there's not much to choose between them.

To FileClose Or Not To FileClose, That Is The Question

Assuming that the target directory exists, then the `RefreshFolderAndFileList` routine gets invoked. As you've probably guessed, I added another property, `Folders`, to my file system object, implementing it as a `TStringList`. This property contains the list of folders that exist at the current directory level:

```
property Folders: TStringList
  read fFolders;
```

As ever, this declaration has to go into the `public` (rather than `published`) part of the class declaration because Delphi doesn't like read-only published properties. I really hate having to write dummy property 'setter' procedures just to make a property appear in the property inspector. Sigh... maybe this silly limitation will get fixed in Kylix, but I'm not holding my breath.

The code for `RefreshFolderAndFileList` clears the existing list of folders and then uses the familiar `FindFirst` and `FindNext` routines to iterate through the list of available files, searching for directories and adding them to the `Folders` property.

As I may have said last time round, there's one thing that's virtually guaranteed to trip up any programmer who writes his/her first tree-walking recursive directory scanner, and that's the gotcha associated with the '.' and '..' directory names. As we've already seen, I effectively support this convention when assigning to the

FolderName property, but I definitely don't believe that these pseudo-directory names should be returned as part of a list of available files/folders. For this reason, the RefreshFolderAndFileList routine specifically looks for any directory which begins with a period and excludes it from the list of returned names.

If only Microsoft had done the same thing when they implemented the low-level routines that implement FindFirst and FindNext functionality in the MSDOS kernel! If they'd done so, it would have saved everyone a lot of grief. To this day, when you type DIR at a DOS prompt, you'll see '.'

➤ *Listing 3*

```
procedure TFileSystem.SetFolderName (Value: String);
var Idx: Integer;
begin
  // Do the trivial stuff first....
  if Value = '.' then
    Exit;
  if Value = '' then
    Value := fDriveLetter + ':\';
  // Handle a request to go up one level
  if Value = '..' then begin
    // Already at root
    if Length (fFolderName) = 3 then
      Exit;
    Idx := Length (fFolderName) - 1;
    while fFolderName[Idx] <> '\' do
      Dec(Idx);
    Value := Copy(fFolderName, 1, Idx);
  end;
  // Handle relative path (no drive letter or backslash)
  if (Value [1] <> '\') and (Value [2] <> ':') then
    Value := fFolderName + Value;
  // Handle an absolute path (no leading drive letter)
  if Value [1] = '\' then
    Value := fDriveLetter + ':' + Value;
  // Handle a path -- with drive letter
  if Value [2] = ':' then begin
    Value [1] := UpCase (Value [1]);
    if Value [1] <> fDriveLetter then
      Exit;
    if Value [3] <> '\' then
      Value := fFolderName + Copy(Value, 3, MaxInt);
  end;
  // At this point, Value should be in the form X:\YYYYYY
  // Now, we need to check that the wanted path exists
  if not DirectoryExists(Value) then
    Exit;
  // Finally, set new folder name and refresh folder list
  if Value [Length (Value)] <> '\' then
    Value := Value + '\';
  if AnsiLowerCaseFileName(Value) <>
   AnsiLowerCaseFileName(fFolderName) then begin
    fFolderName := Value;
    RefreshFolderAndFileList;
  end;
end;
procedure TFileSystem.SetFileTypes (Value: TFileTypes);
begin
  if Value <> fFileTypes then begin
    fFileTypes := Value;
    RefreshFolderAndFileList;
  end;
end;
class function TFileSystem.DirectoryExists(
  const DirName: String): Boolean;
var OldDir: String;
begin
  OldDir := GetCurrentDir;
  try
    Result := SetCurrentDir(DirName);
  finally
    SetCurrentDir(OldDir);
  end;
end;
function TFileSystem.MatchingFile(Rec: TSearchRec): Boolean;
begin
  Result := True;
  // Read-only file ?
  if ((Rec.Attr and faReadOnly) < >0) and
    (ftReadOnly in fFileTypes) then Exit;
  // Hidden file ?
  if ((Rec.Attr and faHidden) <> 0) and
    (ftHidden in fFileTypes) then Exit;
  // System-file ?
  if ((Rec.Attr and faSysFile) <> 0) and
    (ftSystem in fFileTypes) then Exit;
  // Archive file ?
  if ((Rec.Attr and faArchive) <> 0) and
    (ftArchive in fFileTypes) then Exit;
  Result := Rec.Attr = 0;
end;
procedure TFileSystem.RefreshFolderAndFileList;
var
  Err: Integer;
  Rec: TSearchRec;
begin
  fFolders.Clear;
  fFiles.Clear;
  fTotalFileSize := 0;
  Err := FindFirst (fFolderName + '*.*', faAnyFile, Rec);
  try
    while Err = 0 do begin
      if (Rec.Attr and faDirectory) <> 0 then begin
        // Ignore the accursed '.' and '..' names
        if Rec.Name [1] <> '.' then
          fFolders.Add (Rec.Name);
      end else if (Rec.Attr and faVolumeID) = 0 then
        // Not a directory, not a volumeID - must be a file!
        if MatchingFile (Rec) then begin
          fFiles.Add (Rec.Name);
          fTotalFileSize := fTotalFileSize + Rec.Size;
        end;
      Err := FindNext (Rec);
    end;
  finally
    FindClose (Rec);
  end;
end;
procedure TFileSystem.TreeWalkFiles (Proc: TWalkProc);
var Continue: Boolean;
begin
  Screen.Cursor := crHourGlass;
  try
    Continue := True;
    if Assigned(Proc) then
      FileWalker(fFolderName, Proc, Continue);
  finally
    Screen.Cursor := crDefault;
  end;
end;
procedure TFileSystem.TreeWalkFolders (Proc: TWalkProc);
var Continue: Boolean;
begin
  Screen.Cursor := crHourGlass;
  try
    Continue := True;
    if Assigned(Proc) then
      FolderWalker(fFolderName, Proc, Continue);
  finally
    Screen.Cursor := crDefault;
  end;
end;
procedure TFileSystem.FileWalker(const Folder: String;
  Proc: TWalkProc; var Continue: Boolean);
var
  Err: Integer;
  Rec: TSearchRec;
begin
  Err := FindFirst (Folder + '*.*', faAnyFile, Rec);
  try
    while (Err = 0) and Continue do begin
      if (Rec.Attr and faDirectory) <> 0 then begin
        // Ignore the accursed '.' and '..' names
        if Rec.Name [1] <> '.' then
          FileWalker(Folder+Rec.Name+'\', Proc, Continue);
      end else if (Rec.Attr and faVolumeID) = 0 then
        // Not a directory, not a volumeID - must be a file!
        if MatchingFile (Rec) then begin
          Proc (Folder + Rec.Name, Rec, Continue);
        end;
      Err := FindNext (Rec);
    end;
  finally
    FindClose (Rec);
  end;
end;
procedure TFileSystem.FolderWalker (const Folder: String;
  Proc: TWalkProc; var Continue: Boolean);
var
  Err: Integer;
  Rec: TSearchRec;
begin
  Err := FindFirst (Folder + '*.*', faAnyFile, Rec);
  try
    while (Err = 0) and Continue do begin
      if (Rec.Attr and faDirectory) <> 0 then begin
        // Ignore the accursed '.' and '..' names
        if Rec.Name [1] <> '.' then
          FileWalker(Folder+Rec.Name + '\', Proc, Continue);
        Proc(Folder+Rec.Name+'\', Rec, Continue);
      end;
        Err := FindNext (Rec);
    end;
  finally
    FindClose (Rec);
  end;
end;
```

and '..' included as part of the listing and even included in the directory count at the bottom of the screen. On the positive side, there's some evidence that Microsoft might be coming round to my way of thinking on this: you'll find that the `Folders` collection implemented in the Visual Basic FSO library doesn't include these pseudo-directories in it.

You'll notice, incidentally, that the `RefreshFolderAndFileList` routine calls `FindClose` whether or not the initial call to `FindFirst` resulted in an error. Because of the presence of those two accursed pseudo-directories, it's hard to see how an error might occur. In other words, even an empty folder will contain two entries, and therefore calling `FindFirst` on an empty folder with a file specification of `*.*` won't cause a problem. But for the sake of argument, let's suppose that there was an error. The question then becomes, is it ok to call `FindClose` if the initial `FindFirst` call returned an error? This particular question occupied a number of Delphites on CIX recently, with some folks insisting that you really should only call `FindClose` if the initial `FindFirst` call was successful. Happily, it's very much a moot issue: if you examine Borland's implementation of `FindClose` (look in the SYSUTILS.PAS file) you'll see that it cunningly checks the value of the actual search handle, stored as part of the search record, and only calls the underlying API `FindClose` routine if the handle is valid. I love it when I see clever code like this. It's this sort of knack for simplifying and taming the cumbersome Windows API that Borland are particularly good at. For other thoughts in a similar vein, take a look at Brian Long's excellent article on the `safecall` keyword in last month's issue.

Files Versus Folders

As the name suggests, the `RefreshFolderAndFileList` routine also has the job of filling the `Files` property (another `TStringList`) with a list of all the files at the current directory level. It makes sense to do the two jobs at the same time with a single `FindFirst`/`FindNext` loop, and that is exactly what happens. In order to discriminate between files and folders, the code checks the `faDirectory` bit of the file attribute in the search record. However, if this bit is zero, we cannot guarantee that we are dealing with a file, because it might be a volume ID!

As you may appreciate, this is another example of poor design on the part of Microsoft. Rather than implementing the volume ID in the boot sector of a disk, Microsoft plonked it into the root folder of a volume, and compounded the error by making the volume ID subject to directory lookups, just like any other file or directory entry! You could argue that they had no choice but to put it in the directory area, because there was no room in the boot sector (though I'm not sure if this is true). But whoever decided that the DOS-level `FindFirst`/`FindNext` routines would return the volume ID directory entry has the same relationship to API design as Cyril Smith has to ballet! A much better idea would have been to implement a completely new operating system function to get/set the volume label, and completely ignore the volume ID directory entry when performing lookups. Do I sometimes sound like a frustrated would-be operating system designer? Yup, I guess I do...

In last month's code, you'll remember that I added a property called `DriveTypes`, the idea being that `TFileSystem` would ignore all drives that weren't of a specific type. In the same way, I've added another property `FileTypes` which allows you to perform filtering based on a set of file attributes. It's defined as in Listing 4.

Notice that I haven't included volume IDs and directory attributes because they don't logically belong here! A file is a file, and a directory is a directory, and never the twain shall meet! In MSDOS, the archive bit is arguably rather vestigial: the idea is that backup software and other archiving tools set this bit to indicate whether or not a particular file has been backed up. If you trawl through your hard disk with Windows Explorer, you'll find that many files have the archive bit set and for this reason the `FileTypes` property includes the `ftArchive` bit by default. You need to fully appreciate how the `FileTypes` property works. If you, for example, set this property to `[ftHidden]` then it simply means that hidden files will be added to the `Files` property *in addition* to normal files.

```
TFileType   = (ftReadOnly, ftHidden, ftSystem, ftArchive);
TFileTypes  = set of TFileType;
property FileTypes: TFileTypes read fFileTypes write SetFileTypes
  default [ftArchive];
```

➤ *Listing 4*

For the sake of convenience, I've also added a `TotalFileSize` property. Given a specific set of files in the `Files` property, `TotalFileSize` will return the total byte size of all the files in the list. This calculation is performed within the `RefreshFolderAndFileList` routine while the available files and folders are being enumerated.

First Steps In Tree-Walking

Although the `TFileSystem` object is deeply wonderful as far as it goes (wot, me, biased?) my real intention here was to write a unit that would make it simple to perform relatively high-level operations such as, for example, deleting a directory tree, moving a directory tree, calculating the size of a directory tree and so forth. In order to do this, we need (drum-roll, please) the inevitable tree-walking algorithm!

In case you were wondering, the reason I have a rather special fondness for tree-walking algorithms is because the first ever magazine article I wrote was for a tree-walker, in the (long defunct) *Program Now* magazine.

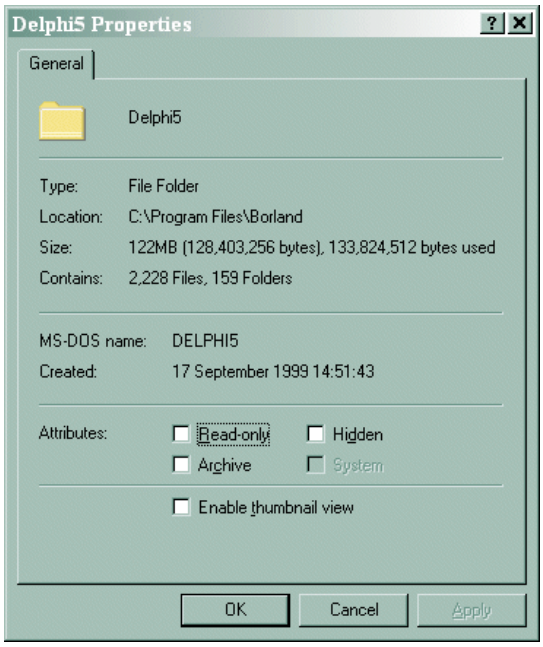
A little known feature of Windows Explorer is that if you right click on a folder name and then select `Properties` from the ensuing

pop-up context menu, Explorer will do a tree-walk of the folder and then display an information panel telling you the total size of the folder, including all its sub-folders (see Figure 2). I wanted to make it easy for an application programmer to do the same thing using the `TFileSystem` object, and you'll see how easy this is a little later when we look at this month's test bed program.

In order to be versatile, a tree-walking algorithm needs to be completely general, which simply means that it doesn't know (or care) what it's supposed to do for each file or folder that it visits. As pointed out above, you might want to delete all the files in a tree, copy the files, search for specific filenames, or whatever. The tree-walking algorithm doesn't know and doesn't care what you want to do, but for every 'node' that it visits, you need to supply a call-back procedure which does the real work:

```
TWalkProc = procedure (
  const Name: String;
  const Info: TSearchRec;
  var Continue: Boolean)
  of Object;
```

The `TFileSystem` object actually implements two distinct tree-walking routines, `TreeWalkFiles` and `TreeWalkFolders`. As the names suggest, the first deals with files and the second with folders. Both take a callback routine of type `TWalkProc`. The first routine calls the procedure which you supply for every *file* that it encounters while the second calls the procedure for every encountered *folder*. In the case of `TreeWalkFolders`, any child folders are enumerated (and the callback routine is called for each of them) before the call-back is invoked for the parent folder.



➤ *Figure 2: A little-known feature of Windows Explorer is the ability to display the total size of a specified folder using the Properties item on the pop-up context menu. Here, we can see that Delphi 5 weighs in at a surprisingly trim 122Mb.*

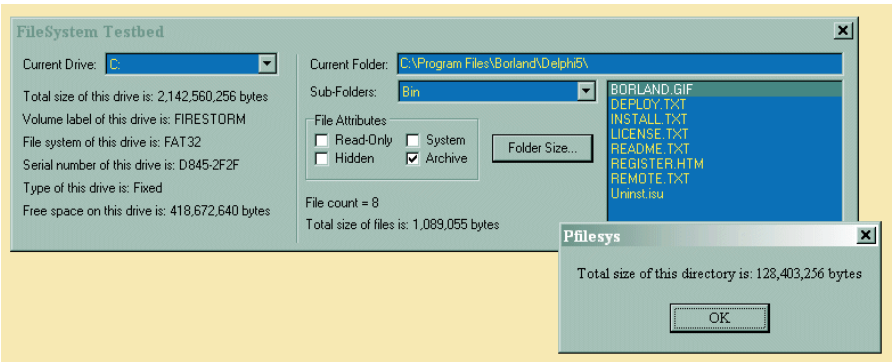
This is important because you might, for example, want to do a tree-walking delete on a directory tree. MSDOS won't allow you to delete a subdirectory which itself contains subdirectories, and that's true even if the child directories are themselves empty. Thus, you have to walk the tree removing the deepest level 'great-great-grandchildren' before you can then delete the 'great-grandchildren' before you can... etc! By implementing `TreeWalkFolders` in such a way that the callback routine is called for the most nested directories first, all this comes out in the wash.

In order to make things as flexible as possible, the `TWalkProc` callback routine takes three parameters. The first is the fully-qualified pathname of the object being enumerated. In the case of `TreeWalkFolders`, this will be the full name of each directory, terminated with a backslash, for example `C:\WINDOWS\SYSTEM\`. In the case of `TreeWalkFiles`, it will obviously be the name of each file that's encountered. *You should note carefully*, however, that the `TreeWalkFiles` routine respects the current value of the `FileTypes` property as discussed above. Thus, if you want to make darn sure that you enumerate *all* the files in a directory tree, you should set `FileTypes` to `[ftReadOnly, ftHidden, ftSystem, ftArchive]` before calling `TreeWalkFiles`.

The second parameter passed to the callback routine is the `TSearchRec` record that was used to enumerate the file or folder. This contains some useful information such as the size of the file, the modification date/time of the file or folder, and also includes another field, `FindData`, which is a record of type `TWin32FindData`. This contains a *lot* of extra information (possibly more than you want!) including the name of the file in old-style 8.3 character format.

The final parameter, `Continue`, is a `Boolean Var` parameter which is set to `True` by default. If you set this

➤ *Figure 3: This month's test bed application for the TFileSystem object demonstrates how to use the general-purpose TreeWalkFiles routine to tot up the size of all the files in a given directory tree. Use it to delete all the files on your disk if you like, but don't blame me...*



to False, the tree-walking algorithm will terminate. This is useful if (for instance) you want to stop a tree-walk when you've found a particular file, or perhaps you run out of disk space part way through copying a large directory tree and need to stop the bus at the earliest opportunity!

The Test Bed Program

Putting all this together, I built a revised test bed program to demonstrate this month's new additions to the code. You can see the revised program running in Figure 3, and the important parts of the code in Listing 5. Drive specific information is displayed on the left-hand side of the program window with folder and file specific information on the right. At any point, the Current Folder edit box shows the folder to which the TFileSystem object is 'pointing' and the Sub-Folders combobox contains the list of sub-folders (if any) that are located below the current folder. The listbox on the left shows the list of files which reside at the current directory level. This file list can be filtered by using the four file-attribute checkboxes which (if clicked) cause an updated file list to be displayed immediately.

It is instructive to run this little program on your Windows\System directory. On my machine, it informed me that I had 2,329 files in my system directory, and that they occupied almost 400Mb of disk space. Aren't you just looking forward to moving to a really *grown-up* operating system like Windows 2000? ☺.

As you can see, the test bed program also displays the total number of files in the Files property at any time, in addition to the total byte size of all the files at the current directory level. What it doesn't show is the total size of all the files, including sub-folders. Recalculating this on the fly would be a somewhat time-consuming process, and so I added a push-button to calculate this information on request, demonstrating the use of the TreeWalkFiles routine.

As you can see, using TreeWalkFiles is very straightforward. The test bed form declaration

➤ *Listing 5*

```
var
  FileSys: TFileSystem;
procedure TForm1.FormCreate (Sender: TObject);
var Idx: Integer;
begin
  FileSys := TFileSystem.Create (Self);
  with FileSys do begin
    DriveTypes := [ fsFixed, fsRemote, fsCDROM ];
    for Idx := 0 to DriveCount - 1 do DriveList.Items.Add
     (Drives [Idx] + ':');
    DriveList.ItemIndex := 0;
    DriveListChange (Self);
    // Set File attribute checkboxes according to current
    // 'FileTypes'
    cbReadOnly.Checked := ftReadOnly in FileTypes;
    cbHidden.Checked   := ftHidden in FileTypes;
    cbSystem.Checked   := ftSystem in FileTypes;
    cbArchive.Checked  := ftArchive in FileTypes;
  end;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  FileSys.Free;
end;
function TForm1.FormatBigBytes (const Msg: String; Value:
  TLargeInteger): String;
var
  Dbl: Double;
begin
   Dbl := Value;
   Result := Format (Msg + ' %n', [Dbl]);
   Result := Copy(Result, 1, Length(Result)-3)+' bytes';
end;
procedure TForm1.DriveListChange(Sender: TObject);
var S: String;
  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
  with FileSys do begin
    // First, point TFileSystem object at the new drive
    DriveLetter := DriveList.Text [1];
    // Now display the various drive properties
    VolSize.Caption := FormatBigBytes(
      'Total size of this drive is:', TotalSize);
    S := VolumeName;
    if S = '' then
      S := '[None]';
    VolName.Caption := Format(
      'Volume label of this drive is: %s', [S]);
    FSystem.Caption := Format(
      'File system of this drive is: %s', [FileSystem]);
    SerNum.Caption := Format(
      'Serial number of this drive is: %s', [SerialNumber]);
    DrvType.Caption := Format(
      'Type of this drive is: %s',
      [StrDriveType(DriveType)]);
    FreeSp.Caption := FormatBigBytes(
      'Free space on this drive is:', FreeSpace);
    UpdateFolderList('');
  end;
end;
procedure TForm1.UpdateFolderList(const FolderName: String);
begin
  if FolderName <> '' then
    FileSys.FolderName := FolderName;
  CurFolder.Text := FileSys.FolderName;
  FolderList.Items.Assign(FileSys.Folders);
  FolderList.ItemIndex := 0;
  FileList.Items.Assign(FileSys.Files);
  FileList.ItemIndex := 0;
  FileCount.Caption := Format(
    'File count = %d', [FileList.Items.Count]);
  TotFileSize.Caption := FormatBigBytes(
    'Total size of files is:', FileSys.TotalFileSize);
end;
procedure TForm1.FolderListChange(Sender: TObject);
begin
  UpdateFolderList (FolderList.Text);
end;
procedure TForm1.CurFolderKeyPress(
  Sender: TObject; var Key: Char);
begin
  if Key = #13 then UpdateFolderList (CurFolder.Text);
end;
procedure TForm1.cbReadOnlyClick(Sender: TObject);
var ft: TFileType;
begin
  with Sender as TCheckBox do begin
    ft := TFileType (Tag);
    if Checked then
      FileSys.FileTypes := FileSys.FileTypes + [ft]
    else
      FileSys.FileTypes := FileSys.FileTypes - [ft];
    FileList.Items.Assign (FileSys.Files);
    FileList.ItemIndex := 0;
    FileCount.Caption := Format(
      'File count = %d',[FileList.Items.Count]);
    TotFileSize.Caption := FormatBigBytes(
      'Total size of files is:', FileSys.TotalFileSize);
  end;
end;
procedure TForm1.SumProc (const Name: String; const Info:
  TSearchRec; var Continue: Boolean);
begin
  DirSize := DirSize + Info.Size;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  DirSize := 0;
  FileSys.TreeWalkFiles (SumProc);
  ShowMessage(FormatBigBytes(
    'Total size of this directory is:', DirSize));
end;
end.
```

includes a private TLargeInteger called DirSize which is used to accumulate the total size of all the encountered files. It gets initialised to zero within the Button1Click routine and a call is then made to TreeWalkFiles, passing it the address of the SumProc callback routine. The only job of this routine is to add the size of each file to DirSize and Button1Click then displays the total number of bytes using the very handy-dandy little FormatBigBytes routine that I use elsewhere in the test bed code.

Reassuringly, the test bed code returns exactly the same total byte size for a folder as is reported by Windows Explorer, assuming that you take care to check all the file attribute flags as I've already discussed.

Next Month

As with last month's code, the TFileSystem object is still contained within the same source file as the test bed code because it's often easier to develop a component that way. Complete project files (for Delphi 4) are included on this month's companion disk along with a read-to-run (packaged) EXE file (which needs the Delphi 4 runtime package).

Oh yes, I lied. Last time round I promised that we'd take a look at file notifications, in other words getting automatically notified by Windows that some file or directory has changed. In fact, I've decided to defer that discussion until part 3 of this mini-series on encapsulating the Windows file system. Part 3 will be appearing in Issue 54 of *The Delphi Magazine* along with some other TFileSystem goodies.

Next month, however, we'll be taking a break from the TFileSystem component. Instead, I'll be taking a look at some of the issues that software developers will come up against in making the transition to Windows 2000. See you then!

The Faux Pas Survival Guide

This is the title of a book which bears the subtitle *The Fine Art of Removing Your Foot from Your Mouth*. I suddenly realised that I had a pressing need for this publication when I received a deluge of email messages drawing my attention to an embarrassing typo in my article on code generation and optimisation hints and tips (Issue 50). Happily, so many readers mentioned this to me that my embarrassment was more than compensated for by a certain smug gratification at the number of folks who read this column!

For the sake of the lone reader who wasn't quite as eagle-eyed as the rest (you know who you are...), here's the problem: I gave a code snippet which illustrated how to pre-initialise a number of object instances to Nil, and I then demonstrated how to use this technique to simplify what would have otherwise been a complicated arrangement of nested try...finally blocks. This technique relies upon the fact that calling Free on a Nil object instance has no effect. The code that I presented was intended to place the actual constructor calls *inside* a try...finally block, but they were inadvertently placed *above* the block. The corrected code fragment should look like this:

```
Bitmap1 := Nil; Bitmap2 := Nil; MemStream := Nil;
try
  Bitmap1 := TBitmap.Create;
  Bitmap2 := TBitmap.Create;
  FileStream := TFileStream.Create (SomeFile, SomeMode);
  >>—— more code goes here ——<<
finally
  Bitmap1.Free;
  Bitmap2.Free;
  FileStream.Free;
end;
```

As I mentioned back then, the idea is that if any constructor should fail and raise an exception, control will be passed to the finally clause, automatically freeing up any objects that *were* successfully created. This technique can greatly simplify otherwise complex code and it really does work (when it's been typed in correctly ☺). First prize in this month's *Spot the Gaff* competition goes to Calum Anderstrem who was well ahead of the rest of the field!

Incidentally, if the title of the above book is of any interest to you, you can find more details on the internet at www.aeu-inc.com /cgi-local/shop.pl/page=faux_pas_survival_guide.htm /SID=2976086

I've no idea whether or not it's available in the UK, but the author definitely isn't Dennis Norden!

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