Beating the System: Now Where Did I Put That File...?

by Dave Jewell

Last month, you'll recall that I introduced a new, drop-in Delphi class called `TZipFile`, the idea being that you can use `TZipFile` to peek inside a ZIP file and enumerate the various files contained therein.

As promised, this month I'm presenting the code for another class, `TTreeFind`, which you can use to enumerate and search for files on one or more disk drives, even when those files might be stashed away inside a ZIP file. Finally, this month I'll also provide the code for a simple test-bed program which shows how to tie this all together.

Sixteen Bits? Just Say No...

As a general rule, I try and make my code work under both 16-bit and 32-bit versions of Windows. In this month's column, I'm not going to bother to do that: it's strictly 32-bit only. No, this isn't laziness on my part, there are some good pragmatic reasons for this decision. Many moons ago, before Windows dominated the market, I did most of my software development under DOS. At the time, I regularly used a disk management program (which had better be nameless!) that allowed me to quickly scan my disk drives for matching files, change file attributes, and so forth.

This program was a big improvement on COMMAND.COM because it presented a graphical tree-view of a disk, albeit in a text mode window. However, problems arose when hard disk drives started getting bigger. My trusty disk management program had been designed to work with floppy disk drives and early disk drives, often no more than 10Mb in size! When faced with a 100Mb drive and several thousand files, it quickly ran out of memory and went belly-up. It's for just this reason that this month's code is 32-bit only.

I make extensive use of a `TStringList` object to store the list of matching files, something that works fine in 32-bit Delphi, but will quickly get you into trouble under 16-bit Delphi where the maximum capacity of a list is 16,380 items. Now admittedly, you might not have sixteen thousand files on a single hard disk, but the code presented here can scan multiple disk drives in one operation, as we shall see. With today's multi-gigabyte drives, a 16-bit program will bump into that upper limit *much* too often. On the other hand, a 32-bit list can store over 134 *million* items. I reckon it'll be a while before hard disk drives get *that* big!

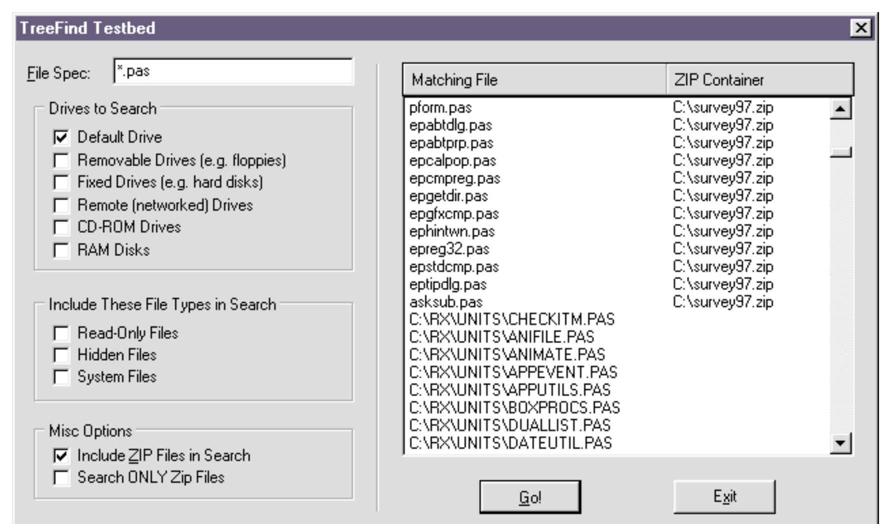
OK, so why use a list? Why can't I process each file, 'as it comes' so to speak? The reason is that I wanted to write a completely general purpose tree searching unit. My little testbed program simply displays a set of files that match the entered file specification, and allows you to 'launch' a selected file. However, in a real-world example, you might want to write a program that (for example) searches your hard disk for .ICO files, and displays them in an owner-draw list-box, allowing you to select a suitable icon for your next project. Some tasks require a list, while others (such as deleting all files of a certain type) can be done sequentially. To be completely general purpose, you've got to implement some sort of list mechanism.

The Testbed Program

The testbed is shown running in Figure 1. Both this program (FINDER.EXE) and the source code are included on this month's cover disk, but do please bear in mind that this particular program isn't the object of the exercise: it's simply a demonstration to show you how to use the classes I've developed over the last couple of months.

As you can see, quite a number of different options can be specified. The first box on the left hand side contains the drive selection parameters: which disk drives are searched for matching files. The `Default Drive` box tells the program to search the default drive.

➤ *Figure 1: Here's the test-bed program, Finder, in action. As you can see from the list-box, zipped file entries are identified by filename with the enclosing name of the ZIP file in the second column.*

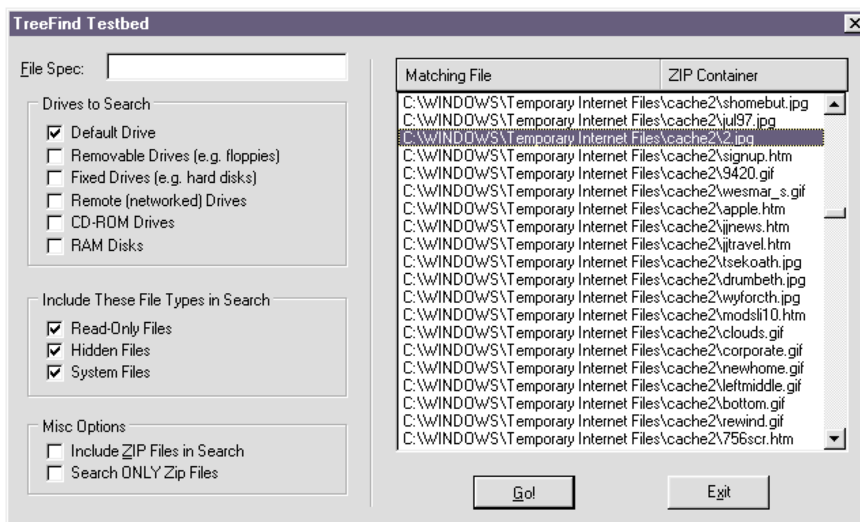


Under DOS and Windows, every process has its own idea of what the 'default' drive might be; the default drive simply depends upon what it was previously set to last time. For example, if you execute a program on drive D:, you'll find that D: is the default drive when the program starts running. However, if the program changes the default (or 'current') drive to C:, then it will stay this way until it's changed again. Because of this uncertainty, it's best not to use the default drive facility, although I have provided it as an option in the code.

The next five drive selection parameters are very straightforward. In each case, clicking a particular check-box includes that drive category in the file search. For example, click the `Removable Drives` checkbox and any floppy disks will be included in the search, click `Fixed Drives` and all your hard disks will be included too. This is all well and good, you may say, but what if you want to just search a specific drive? In order to address that possibility, you can enter an optional drive letter in the `File Spec` box. In other words, entering `*.PAS` will cause the program to do a search for Pascal files on all the checked drive categories. However, if you enter `C:*.PAS` as the file specification, then *only* drive C: will be searched. This is important: specifying an explicit drive letter will override any drive selection parameters. You can see how this works in my testbed program: as soon as you type a drive letter, all drive selection controls are disabled. Naturally, if you don't check any of the drive selection parameters *and* you don't specify a drive letter, you'll get an empty list every time!

Below the drive selection parameters are the `Read-Only`, `Hidden` and `System` checkboxes. These control which file attributes are included in the search. By default, only "normal" files are searched for. You might not realise this, but DOS supports not only hidden files, but hidden directories as well.

Here's an interesting little experiment you can try: this



➤ *Figure 2: Would the Windows 95 Explorer ever lie to you? As a matter of fact, yes. The picture shown here represents the true state of the Internet Explorer temporary files cache, but it ain't what Explorer tells you is going on. Read the accompanying text for a more detailed explanation.*

➤ *Facing page: Listing 1*

```
unit TFind;
interface
uses Wintypes, WinProcs, SysUtils, Classes,
  Controls, Forms, Dialogs, FileCtrl;
const
  { Attribute bits }
  TF_ReadOnly   =    $0001;  { Include read-only files }
  TF_Hidden     =    $0002;  { Include hidden files }
  TF_SysFile    =    $0004;  { Include system files }
  TF_AllAttribs =    $0007;  { mask for attributes }
  { Drive Flags - only apply if no drive letter given }
  TF_DefDrive   =    $0008;  { search default drive }
  TF_Removable  =    $0010;  { search removable drives }
  TF_Fixed      =    $0020;  { search fixed drives }
  TF_Remote     =    $0040;  { search networked drives }
  TF_CDROM      =    $0080;  { search CDROM drives }
  TF_RamDisk    =    $0100;  { search RAM disks }
  TF_AllDrives  =    $01f8;  { mask for drive flags }
  { Misc flags }
  TF_ZIPOnly    =    $4000;  { ONLY look inside ZIP files  }
  TF_ZIP        =    $8000;  { Include ZIP files in search }
type
  TTreeFindProgress = procedure(Sender: TObject;
    const Dir: String) of object;
  TTreeFind = class (TObject)
  private
    flags: Word;
    fSpec: String;
    fFileSpec: String;
    fList: TStringList;
    fProgress: TTreeFindProgress;
    function BuildDriveList(DriveList: TStringList):Boolean;
    procedure TreeSearch (const Spec: String);
    procedure SearchZipFile (const ZipFileName: String;
      const Spec: String);
  public
    constructor Create;
    destructor Destroy; override;
    property SearchFlags: Word read flags write flags;
    property FileSpec: String read fFileSpec
      write fFileSpec;
    property TheList: TStringList read fList;
    property Progress: TTreeFindProgress read fProgress
      write fProgress;
    procedure Execute;
  end;
implementation
uses Match, Zip;
constructor TTreeFind.Create;
begin
  flags := TF_DefDrive;
  fSpec := '*.*';
  fList := TStringList.Create;
end;
destructor TTreeFind.Destroy;
begin
  fList.Free;
  Inherited Destroy;
end;
function TTreeFind.BuildDriveList(DriveList: TStringList):
  Boolean;
var
  Str: String;
  DType, Idx: Integer;
  DCB: TDriveComboBox;
begin
  Result := True;
  { If no drive flags specified, time to bottle out }
  if Flags and TF_AllDrives = 0 then begin
    Result := False;
    Exit;
  end;
  { First, handle the simple TF_DefDrive case }
  if Flags and TF_DefDrive = TF_DefDrive then begin
    Flags := Flags and (not TF_DefDrive);
    GetDir (0, Str);
    DriveList.Add (UpperCase (Copy (Str, 1, 2)));
  end;
  { If other drive flags also present ...}
  if Flags <> 0 then begin
    { Create temporary er... hack... to enumerate drives! }
    DCB := TDriveComboBox.Create (Application.MainForm);
    try
      DCB.Parent := Application.MainForm;
      DCB.Visible := False;
      DCB.TextCase := tcUpperCase;
      { Loop through each drive in the list }
      for Idx := 0 to DCB.Items.Count - 1 do begin
        Str := Copy (DCB.Items [Idx], 1, 2);
        DType := GetDriveType (PChar (Str + '\'));
        if (DType > Drive_No_Root_Dir) { Valid drive } and
          (Flags and (1 shl (DType + 2)) <> 0) then
          DriveList.Add (Str);
      end;
    finally
      DCB.Free;
    end;
  end;
end;
procedure TTreeFind.SearchZipFile(
  const ZipFileName: String; const Spec: String);
var
  idx: Integer;
  zp: TZipFile;
  fName: String;
begin
  zp := TZipFile.Create (ZipFileName);
  try
    for idx := 0 to zp.FilesCount - 1 do begin
      fName := ExtractFileName (zp.FileName [idx]);
      if IsMatch (Copy (Spec, 3, 255), fName) then
        fList.Add (fName + #9 + ZipFileName);
    end;
  finally
    zp.Free;
  end;
end;
procedure TTreeFind.TreeSearch (const Spec: String);
var
  Dir: String;
  Err: Integer;
  SearchRec: TSearchRec;
begin
  try
    { Find first matching file }
    Err := FindFirst ('*.*', Flags and TF_AllAttribs,
      SearchRec);
    GetDir (0, Dir);
    if Dir [Length (Dir)] <> '\' then Dir := Dir + '\';
    if Assigned (fProgress) and (Flags and TF_ZipOnly = 0)
      then fProgress (Self, Dir);
    { Loop for all files which match the specification }
    while Err = 0 do begin
      if Flags and TF_ZipOnly = 0 then
        if IsMatch(Copy (Spec, 3, 255), SearchRec.Name) then
          fList.Add (Dir + SearchRec.Name);
      { If doesn't match the spec, might still be ZIP file }
      if (Flags and (TF_ZIP or TF_ZIPOnly) <> 0) and
        IsMatch ('*.ZIP', SearchRec.Name) then begin
        { Time to do some ZIP parsing ! }
        fProgress (Self, Dir + SearchRec.Name);
        SearchZipFile (Dir + SearchRec.Name, Spec);
      end;
      Err := FindNext (SearchRec);
    end;
    FindClose (SearchRec);
    { Find first sub-directory (if any) }
    Err := FindFirst ('*.*', (Flags and TF_AllAttribs) or
      faDirectory, SearchRec);
    { Loop for all sub-directories in this directory }
    while Err = 0 do begin
      if ((SearchRec.Attr and faDirectory = faDirectory) and
        (SearchRec.Name [1] <> '.')) then begin
        ChDir (SearchRec.Name);
        TreeSearch (Spec);
        ChDir('..');
      end;
      Err := FindNext (SearchRec);
    end;
    FindClose (SearchRec);
  except
    { Should probably handle List-full errors here, but this
      isn't likely to be an issue for 32-bit Delphi. }
  end;
end;
procedure TTreeFind.Execute;
var
  Idx: Integer;
  DirStash: String;
  DriveList: TStringList;
begin
  fList.Clear;
  fSpec := fFileSpec;
  if fSpec = '' then fSpec := '*.*';
  DriveList := TStringList.Create;
  try
    DriveList.Sorted := True;
    { If drive letter specified, only one drive to check }
    if fSpec [2] = ':' then begin
      DriveList.Add (UpperCase (Copy (fSpec, 1, 2)));
      Delete (fSpec, 1, 2);
    end else if not BuildDriveList(DriveList) then Exit;
    Screen.Cursor := crHourglass;
    try
      { Now apply TreeSearch to each drive }
      for Idx := 0 to DriveList.Count - 1 do begin
        { Save current directory for the drive }
        GetDir (Ord (DriveList [Idx][1]) - $40, DirStash);
        { Start from root }
        ChDir (DriveList [Idx] + '\');
        { Do the search }
        TreeSearch (DriveList [Idx] + FSpec);
        { Restore stashed directory }
        ChDir (DirStash);
      end;
    finally
      Screen.Cursor := crDefault;
    end;
  finally
    DriveList.Free;
  end;
end;
end.
```

A Match Made In Heaven...

Up until now, I haven't discussed the format of the file specification which you use to perform file matching. I've actually cheated in this respect: the code presented here uses a unit, MATCH.PAS, which was developed by a chap called Kevin Boylan. Kevin's code, in turn, is based on public domain C code written by J Kerceval and uploaded to CompuServe many moons ago.

The advantage of using this code is that it will do far more for you than ordinary DOS wildcard filename matching. You get all the usual stuff such as '*' and '?' matching, but it also implements a mini regular expression matching engine. For example, if you want to search for all filenames which begin with a digit, you can enter the following as the file specification `[0-9]*`. Alternatively, if you want to exclude any filenames that begin with 'z', you can enter `[^z]*`.

Other examples are given in the source code for the MATCH.PAS unit, which is included on this month's disk. I haven't exhaustively tried out the many different pattern matching possibilities available with this code and I don't comment on whether or not it's bug free. If you find any problems, you can either report them to Kevin or you can try to fix the code yourself.

Incidentally, while developing the `TFIND` unit, I toyed with the idea of completely decoupling the pattern matching algorithm from the file searching code. In other words, it would be up to the application program to supply a pointer to a procedure which is called from the file searching code every time a match needs to be performed. The advantage of this approach is that you can write different applications with different matching algorithms without touching the file searching code in any way. I didn't pursue this, but it would be very easy to implement.

Before looking at the code for the test-bed application, let's now turn to the `TFIND` unit itself, the code for which is given in Listing 1. Here, you can see that things are based around a new class, `TTreeFind`. The interface to this class is very straightforward: you create an object of type `TTreeFind`, set the `SearchFlags` property according to the various search flags you want to use, set the `FileSpec` property to whatever file specification you're interested in and then call the `Execute` method. Once the `Execute` method returns to the caller, `TheList` will contain a list of matching file names.

That's Progress...

In addition to this simple interface, you can also supply an optional call-back progress procedure. Progress is perhaps something of a misnomer: DOS has such a primitive programming interface that it is impossible to determine the number of files in a particular subdirectory without actually enumerating each file one by one. Thus, if we wanted to implement some sort of 'percentage done' progress gauge, we'd have to make two complete passes through all the selected disk drives: once to determine the total file count, and then once again in order to do the 'for real' file matching. This is obviously not an option from an efficiency point of view. Thus, the progress hook contents itself with passing back the name of the current search directory so that the user can visually see that things are still 'progressing'!

I won't discuss the meaning of all the `TF_xxxx` flags which can be `OR`'d into the `SearchFlags` property. We've already covered the various possibilities in some detail. Similarly, the constructor and destructor calls for `TTreeFind` are very straightforward, the main job being to create and destroy the internal `TStringList` member. Things get more interesting when we turn to the `Execute` method. As a convenience, you'll see that a blank file specification gets converted into `*.*`. This is generally a good thing, but see my later caveats on file deletion!

Because the `TTreeFind` class can potentially operate on several different drives, we need to create a list of drives that we're going to search. This is done through another `TStringList` variable. You'll notice I set the `Sorted` property of the variable to `True`. This isn't merely so that the drives get enumerated in a predictable order, there's a much more subtle point here: when you set the `Sorted` property in a `TStringList` variable, it will automatically cause duplicate entries to be rejected by virtue of the default setting of the `Duplicated` property. We need to ensure that duplicates get rejected because, if you select the `TF_DefDrive` flag and also select `TF_Fixed`, then one of your hard disks will get entered into the list twice, and we obviously don't want a particular drive to be scanned more than once.

If a drive letter has been specified, then, as mentioned earlier, all drive search parameters are ignored, and only the specified drive letter is entered into the `DriveList` variable. If this is not the case, then the `BuildDriveList` routine is called to add one or more drives to the list according to the `SearchFlags` which have been supplied. Once this is done, the cursor is changed to an hourglass, and each drive in the list is enumerated, setting the directory to the root of the drive, and calling the `TreeSearch` method to perform the actual hierarchical scan. Afterwards, the previous current directory is restored and once all drives have been processed the cursor is restored to normal. Finally, `DriveList` is destroyed and the method exits.

So how does `BuildDriveList` work? Well, here again, I've gone in for a bit of judicious cheating. It occurred to me that rather than write a chunk of messy, low-level code to determine what drives were present, it would be easier to just let the VCL library take the

strain! Thus, I create an on-the-fly TDriveComboBox control on whatever the application's main form might be, ensure that it's invisible and use it to determine what drive letters are actually available. Yes, it's naughty, but it works! You'll also see that I deliberately chose the various TF_xxxx flags so that they could be bit-shifted into the appropriate place for masking against the values returned from GetDriveType, making it easy to distinguish the different types of drive that we're interested in.

This leads onto a discussion of TreeSearch, which is the real heart of the code. It's here that we get to do the recursive directory walk. When discussing this project with the Editor, he stressed that it was important to provide support for long filenames. The good news is that, by making use of the FindFirst/FindNext routines in 32-bit Delphi, we get long filename support for free. Aha! Is this another nefarious reason why Dave insisted on doing this in 32-bits, I hear you cry!

As ever, there are some interesting considerations in implementing this code. I needed to be able to detect the presence of ZIP files, but how to do it efficiently? If I'd passed the supplied file specification directly the FindFirst API routine, then specifying (for example) *.PAS would cause all .ZIP files to be missed. Worse, the extended search syntax which is supported by the MATCH unit would have hopelessly confused the simple-minded FindFirst code.

The solution is pretty obvious: always pass *.* to the FindFirst routine and then do any file matching on whatever is returned from each API call. Of course, this means that the matching routine will be called once for each entry in the directory so it should be reasonably fast. The Progress hook is called twice within TreeSearch; once whenever we enter a new directory, and once whenever we start scanning a ZIP file.

The calling code can (if wanted) easily discriminate between the two cases by just looking for the .ZIP suffix on the end of the passed string. If you wanted to get fancy, you could pass a VAR Boolean variable as an additional parameter, giving the caller the opportunity to request that certain directories and/or ZIP files be excluded from the search. I leave this and other enhancements to you!

The final routine in the TFIND unit is SearchZipFile. Based on last month's TZipFile class, it should be pretty obvious what's going on here. The ZIP file is opened and the code loops through the various files, calling the same match algorithm to determine whether or not there's a match with the file specification. By using the same match code, we can do exactly the same fancy regular-expression searches right inside the ZIP file.

Notice that when adding a Zipped entry to the list, it's added as two strings (filename then ZIP filename) separated by a tab character. It's the responsibility of the calling application to watch out for an embedded tab character and interpret this as a zip entry.

Finder, The Testbed Program

This brings us neatly onto a discussion of Finder, the sample program which is included on this month's cover disk, the source for which is shown in Listing 2. The two-column listbox is implemented using my old trick with tab-delimited strings, hence the presence of the familiar `NextSection` routine. I really *must* turn this into a custom control!

Each of the various checkboxes on the left hand side of the window have their `Tag` property set equal to the value of the search flag that they control, and they all point to a single common `OnClick` handler. If you haven't used this technique in your own programs you should, it greatly reduced the amount of code you have to write. The only wrinkle here is the `ZIP Files Only` checkbox: if this is checked, then the `Include ZIP Files` checkbox is essentially superfluous so it's disabled by the `OnClick` code.

In a similar fashion, you'll remember I stated that the drive search parameters are ignored when a drive letter is included in the file specification. Accordingly, the `OnChange` handler for the file specification edit box checks for a drive letter (actually, it just checks that the second character is a colon, but this is probably good enough!) and enables or disabled the six affected checkboxes as appropriate.

The `GoClick` method is called when you click the `Go` button. It creates an instance of a `TTreeFind` object, initialises the `SearchFlags`

➤ *Listing 2*

```
unit Ufinder;
interface
uses
  WinTypes, WinProcs, SysUtils, Messages, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, Buttons, TFind,
  FileCtrl, ExtCtrls, ShellAPI;
type
  TForm1 = class(TForm)
    ListBox1: TListBox;
    FileSpec: TEdit;
    Go: TBitBtn;
    Label1: TLabel;
    Bevel1: TBevel;
    GroupBox1: TGroupBox;
    CheckBox1: TCheckBox;
    CheckBox2: TCheckBox;
    CheckBox3: TCheckBox;
    CheckBox4: TCheckBox;
    CheckBox5: TCheckBox;
    CheckBox6: TCheckBox;
    GroupBox2: TGroupBox;
    CheckBox7: TCheckBox;
    CheckBox8: TCheckBox;
    CheckBox9: TCheckBox;
    Button1: TButton;
    GroupBox3: TGroupBox;
    CheckBox10: TCheckBox;
    CheckBox11: TCheckBox;
    Header1: THeader;
    procedure GoClick(Sender: TObject);
    procedure CheckBox1Click(Sender: TObject);
    procedure Button1Click(Sender: TObject);
    procedure FileSpecChange(Sender: TObject);
    procedure ListBox1DrawItem(Control: TWinControl;
      Index: Integer; Rect: TRect; State: TOwnerDrawState);
    procedure Header1Sized(Sender: TObject; ASection,
      AWidth: Integer);
    procedure ListBox1DblClick(Sender: TObject);
private
  SearchFlags: Word;
  procedure MyProgressHook(
    Sender: TObject; const Dir: String);
public
end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.MyProgressHook (Sender: TObject;
  const Dir: String);
begin
  Caption := 'Scanning: ' + Dir;
end;
procedure TForm1.GoClick(Sender: TObject);
var tf: TTreeFind;
begin
  ListBox1.Clear;
  tf := TTreeFind.Create;
  tf.FileSpec := FileSpec.Text;
  tf.SearchFlags := SearchFlags;
  tf.Progress := Self.MyProgressHook;
  Go.Enabled := False;
  tf.Execute;
  Caption := 'TreeFind Testbed';
  Go.Enabled := True;
  if tf.TheList.Count = 0 then
    Listbox1.Items.Add('No matching files found')
  else
    Listbox1.Items.Assign (tf.TheList);
  tf.Free;
end;
procedure TForm1.CheckBox1Click (Sender: TObject);
begin
  with Sender as TCheckBox do
    if Checked then
      SearchFlags := SearchFlags or Tag
    else
      SearchFlags := Searchflags and (not Tag);
    CheckBox10.Enabled := not CheckBox11.Checked;
end;
procedure TForm1.Button1Click(Sender: TObject);
begin
  Close;
end;
procedure TForm1.FileSpecChange(Sender: TObject);
var DrvFlgs: Boolean;
begin
  DrvFlgs := (Length (FileSpec.Text)< 2)
             or (FileSpec.Text[2]  <> ':');
  CheckBox1.Enabled := DrvFlgs;
  CheckBox2.Enabled := DrvFlgs;
  CheckBox3.Enabled := DrvFlgs;
  CheckBox4.Enabled := DrvFlgs;
  CheckBox5.Enabled := DrvFlgs;
  CheckBox6.Enabled := DrvFlgs;
end;
function NextSection (var Str: String): String;
var idx: Integer;
begin
  if Str <> '' then idx := Pos (#9, Str) else idx := 0;
  if idx = 0 then begin
    NextSection := Str;
    Str := '';
  end else begin
    NextSection := Copy (Str, 1, idx - 1);
    Delete (Str, 1, idx);
  end;
end;
procedure TForm1.ListBox1DrawItem(Control: TWinControl;
  Index: Integer; Rect: TRect; State: TOwnerDrawState);
var
  Str: String;
  x, idx: Integer;
begin
  with ListBox1, ListBox1.Canvas, Header1 do begin
    idx := 0;
    FillRect (Rect);
    x := Rect.left + 3;
    Str := Items [Index];
    while Str <> '' do begin
      TextOut (x, Rect.top, NextSection (Str));
      Inc (x, SectionWidth [idx]);
      Inc (idx);
    end;
  end;
end;
procedure TForm1.Header1Sized(Sender: TObject;
  ASection, AWidth: Integer);
begin
  ListBox1.Invalidate;
end;
procedure TForm1.ListBox1DblClick(Sender: TObject);
var
  fName: String;
  TabPos: Integer;
begin
  with ListBox1 do
    if ItemIndex <> -1 then begin
      fName := Items [ItemIndex];
      { Is this a ZIP entry ? }
      TabPos := Pos (#9, fName);
      if TabPos <> 0 then
        fName := Copy(fName, TabPos + 1, 255);
      ShellExecute(Handle, 'open', PChar (fName), Nil, Nil,
        SW_ShowNormal);
    end;
  end;
end.
```

and FileSpec properties and points the Progress property at a routine which merely updates the form caption, simple, but effective. Before calling the Execute method, the Go button is disabled. This raises the question of re-entrancy; it goes without saying that you shouldn't change any of the TTreeFind object's properties or call the Execute method from inside the progress hook! If you do, large hairy warts will grow all over your program, if not yourself. It's unlikely that this could happen accidentally, except possibly in a multi-threaded program but if you want to positively guard against the possibility, then you could put an exclusion lock into the TTreeFind code, the lock being active while a search is in progress. If you have other user interface elements to drive during a file search, then you should also include a call to Application.ProcessMessages inside the progress hook.

The last piece of interesting code is the double-click handler for the list-box. It checks to see if it's a zipped entry and gets the appropriate filename from the selected listbox entry before calling ShellExecute to actually 'launch' the application associated with the clicked file type.

So Where's The Doomsday Device?

Years ago, during the height of the Cold War, there were stories of a nuclear weapon called the Doomsday Device. This American (or Russian, depending on who told the story) bomb was believed to be so powerful that if ever detonated, it would destroy all life on the planet. I certainly don't know how to build such a device and, even if I did, I wouldn't tell you. But TTreeFind potentially gives you the Delphi programmer's equivalent! I have deliberately not included any code to copy , move or delete files because I don't want to be responsible for unleashing destruction on your hard disk. If you want to write code like this:

```
with tf do
  for Idx := 0 to TheList.Count - 1 do
    DeleteFile (TheList [Idx]);
```

then you're on your own; or at least, you very soon will be once your email package has been vaped from your hard disk! All I'm stressing is that you need to use tree-walking code with some care. By all means write your own Doomsday Device, but *please* show the user a list of files that are going to be deleted by a particular operation and give the user the option of cancelling before letting slip the dogs of war!

Finally, please note that the version of ZIP.PAS included on this month's cover disk is ever so slightly different to what I presented last time round. It's not a bug fix, but I've altered an exception generating routine so that, if an invalid ZIP file is encountered, then the offending filename is reported by the exception. The new line of code is shown below, it's in TZipFile.SetZipName:

```
if tailPos < 0 then raise
  EZipErr.CreateFmt ('%s is not
  a valid ZIP file', [FileName]);
```

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 Dave@HexManiac.com

On our Web site:
http://www.itecuk.com

Here's some of what you can find:

- Updated program and data files for TDMAid, the Article Index Database.
- TDMaid Online for immediate access!
- The Delphi Magazine Book Review Database.
- Is your companion disk dead? The source and example files from the articles for the last few issues are here for download.*
- Details of what's in the next issue.
- Back issues: contents and availability.
- Sample articles from back issues.
- Links to other great Delphi sites.