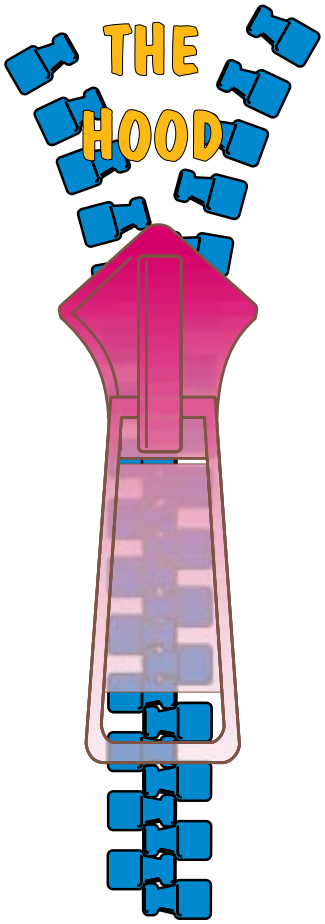
Beating the System

by Dave Jewell

ZIP FILES: PEEKING UNDER THE HOOD



In this month's article, I'm going to describe the internal structure of those ubiquitous ZIP files and explain how you can write Delphi programs that make use of this information. Specifically, this month I'll concentrate on the development of a drop-in Delphi component that can be used to give a neat, object-oriented interface to ZIP files. Next month, I'll make use of this component (and a few more bells and whistles) to develop a program which can scan your hard disk for files, even searching for them inside any ZIP files it encounters. The program will include the ability to launch your favourite ZIP file utility (WinZip, or whatever) from where you can extract the files you're interested in.

Introducing TZipFile...

The code for my ZIP-sniffing component is given in Listing 1. Before doing anything else, there are a few important caveats I should point out. In general, when writing articles for programming magazines, I don't much mind what people do with the code. However, in this particular case I'm retaining rights to the code because I intend to use it as the basis for a shareware component that will allow Delphi programs to access ZIP files, decompress zipped files and perhaps compress files as well. That's quite a bit of work and it isn't finished yet, but you can see some hints in the code listing such as the presence of `Password` and `ExtractDir` properties. If you want to use any of this code for non-commercial purposes, then go right ahead. If you want to use the code as the basis for a file-find utility like that I've outlined above, then I'm perfectly happy with that too. But please don't simply take my code, add data decompression routines, and sell it as your own ZIP component. If you do that, it'll be time to phone the lawyers!

Also, please bear in mind that although for convenience I've referred to `TZipFile` as a component, it's not currently configured as such. To get it on your Component Palette, you'd need to add a `RegisterClasses` call and have it derive from `TComponent` instead of `TObject`. I haven't bothered doing this because the class contains a large number of indexed properties and relatively few non-indexed properties. As you'll appreciate, indexed properties aren't directly supported by the Object Inspector, so I can't see much benefit in making `TZipFile` into a design-time component in the normal sense.

Finally, please bear in mind that this code is 'work in progress' and isn't guaranteed bug free. In particular, I haven't tried porting the code to 16-bit Delphi yet. However, I believe that the present code more than adequately implements the functionality needed by next month's file-searching program. If I discover any 'gotchas' in the meantime, then I'll let you know next month!

With the preliminaries out of the way, let's look at the properties and methods provided by `TZipFile`. In order to use the component, you obviously need to create an instance of `TZipFile`, optionally passing it the fully-qualified name of an existing ZIP file. The specified ZIP file is then examined and the various other component properties are set up according to the information contained therein. If you don't want to specify a ZIP file name when the component is created, then you can pass an empty string. At any time, you can examine the `ZipName` property to determine the ZIP file currently associated with the component. To examine the contents of another ZIP file, simply set the `ZipName` property to point to the new file.

Perhaps the most important non-indexed property associated with the component is `FilesCount`. As the name suggests, this tells the application how many entries are contained within the ZIP file. If you try and read this property before specifying a ZIP file name, the component will raise a `EZipErr` exception with the text *No ZIP file specified*. The `ExtractDir` and `Password` properties are reserved for future use (as mentioned earlier!) and don't do anything in the present implementation. `LowerCaseNames` determines whether or not the name of files contained within the ZIP should be returned as lower case names. By default, this option is on. Bear in mind that pathname information is never forced to lower case, only the filename. Finally, the `SortStyle` and `ReverseSort` properties determine the order in which the filenames are logically sorted. `SortStyle` defaults to `sRaw`, meaning that the files are returned in the same order that they exist within the ZIP file

directory. You can choose any of the values from Table 1 for this property.

In conjunction with the above, you can also set the `ReverseSort` property. This will reverse the 'sense' of the sort, so that what was the first item will be the last, and so on. As you'll see when we look at the code, changing the `SortStyle` property is quite a rapid operation: I've taken some pains to ensure that this is the case. Re-sorting the files does not require the ZIP file directory to be re-read from disk. As my test vehicle, I have a large ZIP file (about 3Mb) containing almost 3,500 icons. Using my little test program, `TZipFile` will re-sort this file in around one second on a 200MHz Pentium Pro machine. If you factor out the time required by the VCL to reload the `TListBox` components with the re-sorted filename list, the actual sort time appears to be around a quarter of a second. Re-sort performance seems to be as good (if not better) than WinZip, the market leader.

Finally, there are a set of thirteen indexed properties, each of which appears as an array property to the calling code. You can access these arrays using an index in the range `[0..FilesCount - 1]`. It goes without saying that the supplied index is always relative to the current `SortStyle` and `ReverseSort` properties. In other words, if you access an array with an index value of five, and then change `SortStyle` or `ReverseSort`, the chances are that an index of five will then give you a completely different entry to what you had before. Thus, if you're looping through a property array, don't change the sort configuration until you've finished the loop, or you'll obviously get nonsensical results.

It would be tedious to describe each of these array properties in detail, so I'll simply list them in Table 2. The `CompressMethod` array property indicates the type of compression used to compress a file entry. The type of this property is `CompressType`, as shown in the code listing. You will almost certainly never see `ResTokenised`, `ResEnhancedDeflate` or `ResPKLibrary` compression types in any ZIP file you're likely to come across. As far as I'm aware, the first two have never been supported in generally available ZIP software, and the last one is reserved for use by the PKWare Data Compression Library. For the convenience of the application program, the `CompressMethodName` property returns a plain English description of the compression method. If the associated code offends the internationalisation purists (!) you can easily put the relevant strings into resources.

The structure of ZIP files allows an arbitrary "comment" to be associated with each stored file. The `CommentLength` property returns the length of the comment (in bytes) associated with a particular entry, but in the code presented here, I haven't (as yet) provided a mechanism for accessing the comment data itself; it's really irrelevant as far as next month's file search utility is concerned.

The `CompressionRatio` array property is used to return the compression ratio for an entry, expressed as a percentage. I chose to return this value as an integer for the sake of language independence (I have plans for porting this code to non-Delphi development environments) but if you want the exact floating-point value, you can obviously calculate it by looking at the `CompressedSize` and `OriginalSize` properties for the entry in question. Internally, when the

SRaw	Files are sorted as they appear in the ZIP file directory
SFullName	Files are sorted alphabetically by their full name
SFileName	Files are sorted alphabetically by file name only
SPathName	Files are sorted alphabetically by path name only
SCompressedSize	Files are sorted in order of compressed size
SOriginalSize	Files are sorted in order of original (uncompressed) size
SCompressRatio	Files are sorted in order of their compression ratio
SDate	Files are sorted in order of modification date and time

➤ *Table 1: SortStyle values*

FullName	Full name of the entry – pathname and filename, eg WOMBAT\SMURF.ICO
FileName	Filename part of the entry, eg. SMURF.ICO
PathName	Pathname part of the entry, eg WOMBAT\
Encrypted	True if this entry is encrypted and requires a password
DiskNumber	Starting disk number (for disk-spanning archives)
Crc32	32-bit CRC check for this file
CompressMethod	Method used to compress this file
DateTime	Modification date/time for this file
CompressedSize	Compressed size of the file in bytes
OriginalSize	Uncompressed size of the file in bytes
CompressMethodName	Plain-English description of the compression method
CommentLength	Length of file comment in bytes
CompressionRatio	Compression ratio expressed as an Integer

➤ *Table 2: Array properties*

➤ *Facing page: Listing 1*

```
unit Zip;
interface
{$A-}
uses WinTypes, WinProcs, SysUtils, Classes, Match;
type
  EZipErr = class(Exception);
  SortType = (sRaw, sFullName, sFileName, sPathName,
sCompressedSize, sOriginalSize, sCompressRatio, sDate);
  CompressType = (Stored, Shrunk, Reduce1, Reduce2, Reduce3,
Reduce4, Imploded, Res Tokenised, Deflated,
ResEnhancedDeflate, ResPKLibrary);
  TZipFile = class(TObject)
  private
    Dir: TList;
    SortMap: TList;
    fd: Integer;
    fSort: SortType;
    pTail: Pointer;
    SelFiles: Integer;
    fName: String;
    fExtractDir: String;
    fPassword: String;
    fLowerCaseNames: Boolean;
    fReverseSort: Boolean;
    procedure LoadDirectory;
    procedure UnloadDirectory;
    function GetSigOffset(Signature: LongInt): LongInt;
    function GetDirectoryEntry(Idx: Integer): Pointer;
    function GetFilesCount: Integer;
    procedure SortFiles;
    procedure DoSort(L, R: Integer);
    function GetFullName(Index: Integer): String;
    function GetFileName(Index: Integer): String;
    function GetPathName(Index: Integer): String;
    function GetEncrypted(Index: Integer): Boolean;
    function GetCompressMethod(Index: Integer):
      CompressType;
    function GetCompressMethodName(Index: Integer): String;
    function GetCompressionRatio(Index: Integer): Integer;
    function GetDiskNumber(Index: Integer): Integer;
    function GetCrc32(Index: Integer): LongInt;
    function GetCompressedSize(Index: Integer): LongInt;
    function GetOriginalSize(Index: Integer): LongInt;
    function GetDateTime(Index: Integer): TDateTime;
    function GetCommentLength(Index: Integer): Word;
    procedure SetZipName(const FileName: String);
    procedure SetSortType(Val: SortType);
    procedure SetReverseSort(Val: Boolean);
  public
    constructor Create(const FileName: String);
    destructor Destroy; override;
    procedure Reset;
    property FullName [Index: Integer]: String read
GetFullName; default;
    property FileName [Index: Integer]: String
      read GetFileName;
    property PathName [Index: Integer]: String
      read GetPathName;
    property Encrypted [Index: Integer]: Boolean
      read GetEncrypted;
    property DiskNumber [Index: Integer]: Integer
      read GetDiskNumber;
    property Crc32 [Index: Integer]: LongInt read GetCrc32;
    property CompressMethod [Index: Integer]: CompressType
      read GetCompressMethod;
    property DateTime [Index: Integer]: TDateTime
      read GetDateTime;
    property CompressedSize [Index: Integer]: LongInt
      read GetCompressedSize;
    property OriginalSize [Index: Integer]: LongInt
      read GetOriginalSize;
    property CompressMethodName [Index: Integer]: String
      read GetCompressMethodName;
    property CommentLength [Index: Integer]: Word
      read GetCommentLength;
    property CompressionRatio [Index: Integer]: Integer
      read GetCompressionRatio;
  published
    property ZipName: String read fName write SetZipName;
    property SortStyle: SortType
      read fSort write SetSortType;
    property ExtractDir: String
      read fExtractDir write fExtractDir;
    property Password: String
      read fPassword write fPassword;
    property ReverseSort: Boolean
      read fReverseSort write SetReverseSort default False;
    property LowerCaseNames: Boolean read fLowerCaseNames
      write fLowerCaseNames default True;
    property FilesCount: Integer read GetFilesCount;
  end;
implementation
type
  PTailRec = ^TailRec;
  TailRec = record    { End of central dir: 'tail' }
  Signature: LongInt; { should be $06054b50 }
  ThisDisk: Word;     { # of this disk }
  DirDisk: Word;      { # of disk with central dir start }
  NumEntries: Word;   { # of central dir entries this disk }
  TotEntries: Word;   { # of central dir entries total }
  DirSize: LongInt;   { size of the central directory }
  DirOffset: LongInt; { offset of c-dir wrt starting disk }
  BannerLength: Word; { size of following comment if any }
end;
type
  PDirEntry = ^DirEntry;
  DirEntry = record             { Central Directory entry }
  Signature: LongInt;           { should be $02014b50 }
  CreatorVersion: Word;    { version of ZIP that created it }
  ExtractorVersion: Word;  { version of ZIP needed for extract }
  GenBits: Word;                { general purpose bit flags }
  CompressMethod: Word; { compression method for this file }
  DateTime: LongInt;    { file modification date/time }
  crc32: LongInt;               { 32-bit file CRC }
  CompressedSize: LongInt;      { compressed size of file }
  OriginalSize: LongInt;        { uncompressed size of file }
  FileNameLen: Word;            { length of filename }
  ExtraLen: Word;               { length of extra info }
  CommentLen: Word;             { length of comment stuff }
  DiskNumStart: Word;           { starting disk number }
  IFileAttribs: Word;           { File attributes }
  XFileAttribs: LongInt;        { External file attributes }
  HeaderPos: LongInt;           { offset of local header }
end;
function GetDirEntrySize(const Entry: DirEntry): Integer;
begin
  with Entry do Result := sizeof(DirEntry) + FileNameLen +
                          ExtraLen + CommentLen;
end;
function IsValidTailPos(fd: Integer; tailPos: LongInt):
  Bool;
var tail: TailRec;
begin
  { This function is needed to cope with nested ZIP files  }
  { Without it, we might accidentally accept a tail marker }
  { inside a nested ZIP rather than the ZIP's own marker ! }
  Result := False;
  _llseek(fd, tailPos, 0);
  _lread(fd, @tail, sizeof(TailRec));
  if tail.Signature = $06054b50 then begin
    _llseek(fd, tail.DirOffset, 0);
    _lread(fd, @tail, sizeof(LongInt));
    Result := tail.Signature = $02014b50;
  end;
end;
function FindSig(fd: Integer; buff: PChar; len: Integer;
  fPos, Signature: LongInt): integer;
var
  p: PChar;
  pp: ^LongInt absolute p;
begin
  Result := -1;
  if len <> 0 then begin
    p := buff;
    while len <> 0 do begin
      if (pp^ = Signature) and
        IsValidTailPos(fd, fpos + p - buff) then begin
        Result := p - buff;
        Exit;
      end;
      Inc(p);
      Dec(len);
    end;
  end;
end;
{ These utility routines extract various fields
  from a DirEntry }
function DirGetFullName(pde: PDirEntry): String;
var
  Idx: Integer;
begin
  Result := '';
  if pde <> Nil then with pde^ do begin
    {$IFDEF WIN32}
     SetLength(Result, FileNameLen);
    {$ELSE}
     Result[0] := Chr(FileNameLen);
    {$ENDIF}
    Move((PChar(pde) + sizeof(DirEntry))^, Result [1],
         FileNameLen);
    { Massage UNIX forward slashes to Wintel backslashes }
    for Idx := 1 to Length(Result) do
      if Result [Idx] = '/' then Result [Idx] := '\';
  end;
end;
function DirGetCompRatio(pde: PDirEntry): Double;
begin
  Result := 0;
  if pde <> Nil then with pde^ do
    if OriginalSize <> 0 then
      Result := ((OriginalSize - CompressedSize) * 100) /
                 OriginalSize;
end;
constructor TZipFile.Create(const FileName: String);
begin
  fd := -1;
  SortMap := TList.Create;
  fLowerCaseNames := True;
  fReverseSort := False;
  SetZipName(FileName);
end;
```

{ Continued on page 38... }

```
{ Continued from page 37... }
procedure TZipFile.SetZipName(const FileName: String);
var
  tail: TailRec;
  tailPos: LongInt;
  szName: array [0..255] of Char;
begin
  UnloadDirectory;
  fName := '';  fPassword := '';
  { If filename is empty, just exit }
  if FileName = '' then Exit;
  { Get filename and make sure it has a proper extension }
  StrPCopy(szName, FileName);
  if StrPos(szName, '.') = Nil then lstrcat(szName, '.zip');
  { Now try to open the file }
  fd := _lopen(szName, of_Read or of_Share_Deny_Write);
  if fd = -1 then
    raise EZipErr.Create('Cannot open specified file');
  fName := StrPas(szName);
  { OK - it's there, but is it a valid ZIP file ? }
  tailPos := GetSigOffset($06054b50);
  if tailPos < 0 then
    raise EZipErr.Create('Not a valid ZIP file');
  { Found the directory tail - ensure no disk spanning }
  _llseek(fd, tailPos, 0);
  _lread(fd, @tail, sizeof(TailRec));
  if (tail.ThisDisk <> 0) or (tail.DirDisk <> 0) then
    raise EZipErr.Create(
      'Disk spanning not yet implemented');
  { Read directory tail and banner into our data structure }
  GetMem(pTail, sizeof(TailRec) + tail.BannerLength);
  _llseek(fd, tailPos, 0);
  _lread(fd, PChar(pTail), sizeof(TailRec) +
         tail.BannerLength);
  { Now get central directory & ensure all files selected }
  LoadDirectory;
end;

destructor TZipFile.Destroy;
begin
  UnloadDirectory;
  SortMap.Free;
  Inherited Destroy;
end;

procedure TZipFile.LoadDirectory;
var
  p: PChar;
  de: DirEntry;
  sz, Idx: Integer;
  function NonBlankEntry: Boolean;
  begin
    Result := (de.CompressedSize <> 0) or
      (de.OriginalSize <> 0) or (de.CompressMethod <> 0);
  end;
begin
  { Initialize directory TList }
  Dir := TList.Create;
  Dir.Capacity := PTailRec(pTail)^.NumEntries;
  { Seek to start of file }
  _llseek(fd, PTailRec(pTail)^.DirOffset, 0);
  { Read each entry in consecutively }
  for Idx := 0 to PTailRec(pTail)^.NumEntries - 1 do begin
    _lread(fd, @de, sizeof(de));
    sz := GetDirEntrySize(de);
    GetMem(p, sz);
    Move(de, p^, sizeof(de));
    _lread(fd, p + sizeof(de), sz - sizeof(de));
    { If this is a blank 'directory-marker' record skip it }
    if NonBlankEntry then Dir.Add(p) else FreeMem(p, sz);
  end;
  Reset;
end;

function TZipFile.GetDirectoryEntry(Idx: Integer): Pointer;
begin
  if Dir = Nil then
    raise EZipErr.Create('No ZIP file specified');
  if fReverseSort then
    Idx := SortMap.Count - 1 - Idx;
  Result := SortMap [Idx];
end;

procedure TZipFile.DoSort(L, R: Integer);
var
  P: Pointer;
  I, J: Integer;
  function SortCompare(Key1, Key2: PDirEntry): Integer;
  var
    D1, D2: Double;
    S1, S2: String;
  begin
    D1 := 0; D2 := 0; Result := 0; { Just to shut compiler up }
    case fSort of
      sFullName, sFileName, sPathName:
        begin
          S1 := DirGetFullName(Key1);
          S2 := DirGetFullName(Key2);
          if fSort = sFileName then begin
            S1 := ExtractFileName(S1);
            S2 := ExtractFileName(S2);
          end;
          if fSort = sPathName then begin
            S1 := ExtractFilePath(S1);
            S2 := ExtractFilePath(S2);
          end;
          Result := CompareText(S1, S2);
        end;
      sDate, sCompressedSize, sOriginalSize, sCompressRatio:
        begin
          if fSort = sDate then begin
            D1 := FileDateToDateTime(Key1^.DateTime);
            D2 := FileDateToDateTime(Key2^.DateTime);
          end;
          if fSort = sCompressedSize then begin
            D1 := Key1^.CompressedSize;
            D2 := Key2^.CompressedSize;
          end;
          if fSort = sOriginalSize then begin
            D1 := Key1^.OriginalSize;
            D2 := Key2^.OriginalSize;
          end;
          if fSort = sCompressRatio then begin
            D1 := DirGetCompRatio(Key1);
            D2 := DirGetCompRatio(Key2);
          end;
          if D1 = D2 then
            Result := 0
          else if D1 > D2 then
            Result := 1
          else
            Result := -1;
        end;
    end;
  end;
begin
  repeat
    I := L; J := R; P := SortMap [(L + R) shr 1];
    repeat
      while SortCompare(SortMap [I], P) < 0 do Inc(I);
      while SortCompare(SortMap [J], P) > 0 do Dec(J);
      if I <= J then begin
        SortMap.Exchange(I, J);
        Inc(I);
        Dec(J);
      end;
    until I > J;
    if L < J then DoSort(L, J);
    L := I;
  until I >= R;
end;

procedure TZipFile.SortFiles;
var Idx: Integer;
begin
  { First, clear the sort map }
  SortMap.Clear;
    SortMap.Capacity := FilesCount;
    { Initialise the sort map for 'sRaw' mode }
    for Idx := 0 to FilesCount - 1 do
      SortMap.Add(Dir [Idx]);
    { Now do the actual sort }
    if fSort <> sRaw then
      DoSort(0, SortMap.Count - 1);
end;

procedure TZipFile.SetSortType(Val: SortType);
begin
  fSort := Val;
  if Dir <> Nil then
    SortFiles;
end;

procedure TZipFile.SetReverseSort(Val: Boolean);
begin
  fReverseSort := Val;
end;

procedure TZipFile.Reset;
var idx: Integer;
begin
  SetSortType(fSort);
  SelFiles := FilesCount;
  for idx := 0 to SelFiles - 1 do
    PDirEntry(GetDirectoryEntry(idx))^.Signature := 1;
end;

procedure TZipFile.UnloadDirectory;
  procedure FreeList(var List: TList);
  var
    p: Pointer;
    Idx: Integer;
  begin
    if List <> Nil then begin
      for Idx := 0 to List.Count - 1 do begin
        p := List.Items [Idx];
        FreeMem(p, GetDirEntrySize(PDirEntry(p)^));
      end;
      List.Free;
      List := Nil;
    end;
  end;
begin
  FreeList(Dir);
  if pTail <> Nil then begin
    FreeMem(pTail, sizeof(TailRec) +
      PTailRec(pTail)^.BannerLength);
    pTail := Nil;
  end;

{ Continued on page 40... }
```

➤ *Facing page:*
Listing 1 continued

`SortStyle` is set to `SCompressRatio`, the full floating point value is used, so that the sort order matches other programs such as WinZip.

ZIP File Structure

Now you understand how to use the `TZipFile` class, it's time to look at how it works. However, before we can do that, you need to get some idea of the internal layout of a ZIP file. Most file formats start off with a header which contains pointers to other information within the file: the icon and bitmap files I talked about a couple of months ago are typical examples. Surprisingly, ZIP files aren't like this. Instead, the most important data structure, the central directory, is located towards the end of the file and is followed by an "End of Central Directory Record." It's the End of Central Directory Record (normally referred to as the "tail" and equivalent to the `TailRec` data structure in my code) which must be located before the contents of the file can be enumerated.

One of the great things about ZIP files is that you can arbitrarily add files to an existing ZIP archive, or remove files just as easily. If the directory was placed at the beginning of the ZIP file, adding a new file would increase the size of the directory, thus necessitating that everything else "shuffle down" towards the bottom of the file. The same argument applies to the removal of entries, causing a "shuffle up." By locating the directory towards the end of the file, this shuffle is largely avoided.

Notice that I've been careful to say "towards the end of the file" rather than "at the end of the file." Again, this is a performance optimisation devised by PKWare, the original creators of the ZIP format. Suppose you add a few small files to a ZIP file: the PKZIP program will typically increase the size of the directory slightly and write the new files after the central directory and tail. However, if it detects that the tail and central directory are going to end up more than 64Kb from the end of the file, some shuffling gets performed and the tail and central directory are once more moved to the end of the file. The overall strategy is to minimise disk-intensive shuffles most of the time.

Central file header signature	4 bytes (0x02014b50)
Version made by	2 bytes
Version needed to extract	2 bytes
General purpose bit flag	2 bytes
Compression method	2 bytes
Last mod file time	2 bytes
Last mod file date	2 bytes
CRC-32	4 bytes
Compressed size	4 bytes
Uncompressed size	4 bytes
Filename length	2 bytes
Extra field length	2 bytes
File comment length	2 bytes
Disk number start	2 bytes
Internal file attributes	2 bytes
External file attributes	4 bytes
Relative offset of local header	4 bytes
Filename	variable size
Extra field	variable size
File comment	variable size

➤ *Table 3: Central directory entry structure*

With the benefit of hindsight, this all seems rather bizarre. A fairly obvious question is why on earth weren't the first four bytes of the file used as a pointer to the central directory, wherever it might wind up after each operation? Nowadays, that would be an eminently sensible way of doing things, but you've got to bear in mind that this file format evolved at a time when some PCs only had floppy disk drives! Floppy disk seek time (the time needed to move the read/write head from track to track) was, and still is, horrendously slow. PKWare designed the file format in order to minimise the number of large scale seeks required in the normal course of events. In fact, the PKWare application note even suggests that they were concerned about compatibility with non-seekable output devices.

The bottom line is this: a legal ZIP program will contain a "tail" record within 64Kb of the end of the file and we have to search for it, backwards from the file end. Just in case the ZIP file isn't kosher, I actually search the entire file rather than the last 64Kb, only giving up if the tail cannot be located. As you'll see from the code, the `TailRec` structure contains a field called `DirOffset` which tells us where the central directory is located. The central directory consists of a contiguous sequence of directory entries, of type `DirEntry`. However, these directory entries are all variable sized: the size of each entry depends on the length of the file name, the length of the file comment (if any) and the amount of "extra information" associated with the file. This extra information field is a way of associating arbitrary information with each ZIP file entry; it's used by some ZIP-compatible software to store operating system dependent information which won't fit into the ordinary directory entry. The overall structure of a central directory entry (which is taken from PKWare's application note) is shown in Table 3.

How It Works

Armed with the above information, we can examine the various routines in Listing 1. When you create a new `TZipFile` object and associate it with a ZIP file, the first routine of any interest that gets

```
{ Continued from page 38... }

  if fd <> -1 then begin
    _lclose(fd);
    fd := -1;
  end;
end;
function TZipFile.GetSigOffset(Signature: LongInt): LongInt;
const
  InBufferSize = 8192;            { for sig searching }
var
  buff: PChar;
  fs, pos: LongInt;
  bp, bytesread: Integer;
begin
  GetMem(buff, InBufferSize);
  try
    fs := _llseek(fd, 0, 2);
    if fs <= InBuffersize then
      pos := 0
    else
      pos := fs - InBufferSize;
    _llseek(fd, pos, 0);
    { Get initial buffer content }
    _lread(fd, buff, InBufferSize);
    bp := FindSig(fd, buff, InBufferSize, pos, Signature);
    { This is the main search loop... }
    while (bp < 0) and (pos > 0) do begin
      Move(buff, buff [InBufferSize - 4], 4);
      Dec(pos, InBufferSize - 4);
      if pos < 0 then pos := 0;
      _llseek(fd, pos, 0);
      bytesRead := _lread(fd, buff, InBufferSize - 4);
      if bytesRead < InBufferSize - 4 then
        Move(buff [InBufferSize - 4],
          buff [BytesRead], 4);
      if bytesRead > 0 then begin
        Inc(bytesRead, 4);
        bp :=
          FindSig(fd, buff, bytesRead, pos, Signature);
      end;
    end;
    if bp < 0 then
      GetSigOffset := -1
    else
      GetSigOffset := pos + bp;
  finally
    FreeMem(buff, InBufferSize);
  end;
end;
function TZipFile.GetFilesCount: Integer;
begin
  if Dir = Nil then
    raise EZipErr.Create('No ZIP file specified');
  Result := Dir.Count;
end;
function TZipFile.GetFileName(Index: Integer): String;
begin
  Result := ExtractFileName(GetFullName(Index));
  if fLowerCaseNames then
    Result := LowerCase(Result);
end;
function TZipFile.GetPathName(Index: Integer): String;
begin
  Result := ExtractFilePath(GetFullName(Index));
end;
function TZipFile.GetFullName(Index: Integer): String;
begin
  Result := DirGetFullName(GetDirectoryEntry(Index));
end;
function TZipFile.GetDateTime(Index: Integer): TDateTime;
var pde: PDirEntry;
begin
  Result := 0;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := FileDateToDateTime(pde^.DateTime);
end;
function TZipFile.GetEncrypted(Index: Integer): Boolean;
var pde: PDirEntry;
begin
  Result := False;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := (pde^.GenBits and 1) <> 0;
end;
function TZipFile.GetCompressionRatio(Index: Integer):
Integer;
begin
  Result := Round(DirGetCompRatio(
    GetDirectoryEntry(Index)));
end;
function TZipFile.GetCompressedSize(
  Index: Integer): LongInt;
var pde: PDirEntry;
begin
  Result := 0;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := pde^.CompressedSize;
end;
function TZipFile.GetOriginalSize(Index: Integer): LongInt;
var pde: PDirEntry;
begin
  Result := 0;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := pde^.OriginalSize;
end;
function TZipFile.GetCompressMethod(
  Index: Integer): CompressType;
var pde: PDirEntry;
begin
  Result := Stored;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := CompressType(pde^.CompressMethod);
end;
function TZipFile.GetDiskNumber(Index: Integer): Integer;
var pde: PDirEntry;
begin
  Result := 1;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := pde^.DiskNumStart;
end;
function TZipFile.GetCrc32(Index: Integer): LongInt;
var pde: PDirEntry;
begin
  Result := 0;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := pde^.crc32;
end;
function TZipFile.GetCommentLength(Index: Integer): Word;
var pde: PDirEntry;
begin
  Result := 0;
  pde := GetDirectoryEntry(Index);
  if pde <> Nil then
    Result := pde^.CommentLen;
end;
function TZipFile.GetCompressMethodName(
  Index: Integer): String;
var typ: CompressType;
begin
  typ := GetCompressMethod(Index);
  case typ of
    Stored:             Result := 'Stored';
    Shrunk:             Result := 'Shrunk';
    Reduce1..Reduce4:   Result := 'Reduced';
    Imploded:           Result := 'Imploded';
    Deflated:           Result := 'Deflated';
  else
    Result := Format('Unknown(%d)', [Ord(typ)]);
  end;
end;
end.
```

➤ *Listing 1, concluded*

called is `TZipFile.SetZipName`. This calls `UnloadDirectory` to dispose of any previous directory context and then tries to open the file, generating an exception if the file couldn't be found. As a philosophical aside, let me just say that I don't believe a low-level 'file format interface class' such as `TZipFile` should have any sort of user interface. It doesn't prompt for files, it doesn't put up dialogs to indicate errors, it is completely faceless. The user interface is the sole responsibility of the calling application. Instead, `TZipFile` responds to error conditions by generating exceptions, which can be intercepted and handled appropriately by the application itself.

Once the file is open, `GetSigOffset` is called to find the location of the tail record within the ZIP file. The tail can be identified because it contains a special four byte signature in the first four bytes; more on this in a moment. If no tail can be found an exception is raised to indicate that it's not a valid ZIP file. If the tail is found, the complete tail record is read into memory and

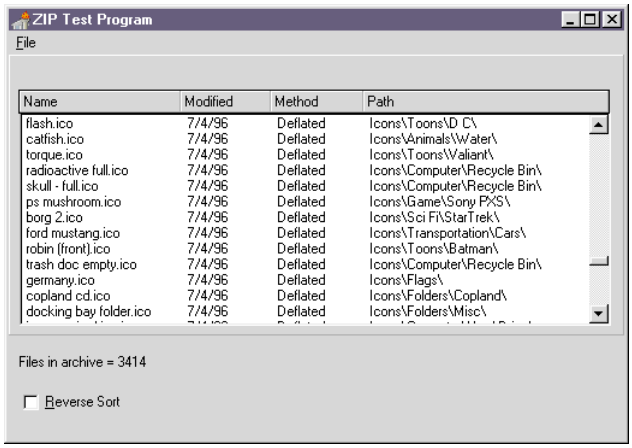
the LoadDirectory method is called to load the ZIP directory proper.

The GetSigOffset routine is responsible for scanning backwards through the ZIP file, looking for the tail signature. It may look a little more complicated than you'd expect, that's because the file is examined in blocks of approximately 8Kb and it's possible that the four byte signature might straddle two of these blocks, causing it to be missed if we simply examined each block individually. Within the GetSigOffset routine, FindSig is called to look for the signature within the current buffer. There is perhaps some argument for rewriting FindSig in assembler code to speed up the signature search, but in practice it wouldn't be worthwhile unless you were dealing with a very large ZIP file that didn't include the tail within the last 64Kb of the file.

If a signature is detected another routine, IsValidTailPos, is called to validate the tail position. This is a very important and subtle point. As you'll no doubt appreciate, it's perfectly legitimate to store one ZIP file inside another ZIP file, a child inside a parent, so to speak. Because ZIP files are already highly compressed, a typical ZIP program will try to compress the child ZIP file, fail, and default to merely 'storing' (CompressMethod = Stored) the child inside the parent file. This means that the enclosing ZIP will contain an uncompressed copy of the child. Now imagine what would happen if our signature scanning routine picked up the tail signature of the child instead of the signature of the parent. I can guarantee you that this not only can happen, but sooner or later it will: the voice of bitter experience! If this problem goes undetected, then deeply bad things will occur! That's the purpose of the IsValidTailPos routine. It ensures that the detected tail signature really is the tail signature of the outermost, enclosing ZIP file.

Once we've got the tail, the rest is easy. LoadDirectory is called to read the central directory into memory, one entry at a time, and store it in the Dir TList variable. This routine calls another small function, GetDirEntrySize, to calculate the total size of each directory entry, allowing for all the variable-sized information that I've previously mentioned. There's another subtle point here: if you look at a ZIP file containing folder names, you'll find that it contains an empty, zero-length entry for each folder name within the archive. I believe that these entries are used to store information relating to the folders themselves. For example, if you want a ZIP file to reconstruct a directory tree, it should ideally recreate each directory with the original modification date and time information. However, I felt that these blank entries were potentially confusing to end-users, and so my TZipFile component silently hides them. This is done by the NonBlankEntry routine inside LoadDirEntry. You'll find that my assessment of the number of files in an archive agrees with WinZip, which also hides blank entries for the same reason.

➤ *Here's the sample program running: this is a seriously large ZIP file, over 3Mb in size and nearly 3,500 entries, but TZipFile doesn't seem to have any problems digesting it!*



Let's Get Sorted...

The final thing I want to discuss is the operation of the sorting code. When the ZIP directory is read into memory, the Dir variable is set up to contain a list of pointers to the individual directory entries. In principle, one could sort the directory entries by merely resorting the entries in this TList variable. However, if we did that, then we wouldn't easily be able to reverse the operation. In order to get back to the sRaw state, we'd have to reread the directory information from disk, or perhaps allocate a large buffer to hold the entire, unprocessed central directory information. This isn't a very elegant solution.

In order to solve this problem, I introduced an additional TList variable called SortMap. As with Dir, SortMap contains a list of pointers to the directory entries. When a sorting operation takes place, the contents of Dir never change, it's the sort map that gets modified. To return to the raw, unsorted state, all that's necessary is to copy all the entries from Dir into their corresponding positions in SortMap.

All good programmers know that QuickSort is the best general purpose sorting algorithm. If you're unconvinced, try running Delphi's THREADS demo program and you'll soon get the message! The algorithm I use here is a standard recursive QuickSort that operates on the SortMap array.

You might wonder why I didn't just call the QuickSort-based Sort method which is now supported by TList. There are at least two good reasons. Firstly, as mentioned earlier, I plan to port this code back to 16-bit Delphi, which does not support sorting in TLists but only in TStringLists.

Secondly, the Sort method in recent TLists has been badly implemented in an inflexible manner. The application-supplied comparison function only takes two parameters, the items to be compared. This means that without resorting to global variables, or some unpleasant hack, it's impossible for the comparison function to access all the information it needs to make the comparison. In

the present case, we'd either have to use separate comparison functions for each distinct sort method, or else copy `fSort` into a global variable so that it could be accessed by the comparator.

If Borland had defined `TListSortCompare` as a `function of object` rather than a plain vanilla function, then it would have been possible to make the comparison function a method of the `TZipFile` class. In the event, I have rolled the QuickSort functionality into the class while (hopefully!) preserving backward compatibility with 16-bit Delphi. See how kind I am to you!

Next Month...

That's it for this month. Next month, I'll present the code for an all-singin', all-dancin' file find utility which will include the ability to search inside ZIP files.

The code for `TZipFile` is included on this month's disk, along with a simple little test program so you can try it out. You can see the program running in Figure 1.

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