Beating The System: TAR-geting Linux...

by Dave Jewell

Oh alright, I unreservedly apologise for that dreadful pun! As promised in the last *Beating The System*, this month we're going to dwell on some of the programming issues that surround moving to Kylix and targeting the Linux operating system. To put it another way, up until now, Delphi and C++Builder developers haven't had to worry much about cross-platform portability issues, but once Kylix (or whatever Borland choose to call it!) hits the streets, we'll all be a lot more interested in porting our applications across to the brave new world of Linux, so we may as well start thinking about portability issues right now.

I'm not allowed to mention specific software companies or products, but I can tell you that a surprising number of key Delphi component developers are already lining up behind the Kylix bandwagon, which means that, by the time Kylix actually ships, there will be quite a number of third-party tools and controls available for it, many of which will be familiar to Windows developers. Thus, whether you're an application developer or a component creator, now is the time to get to grips with portability issues, and now is the time to tackle the tricky task of removing all those 'Windowisms' from your code.

Rather than talking entirely in the abstract, I wanted to do something practical, and therefore this discussion of portability issues is interwoven with the development of `TTarFile`, a (mostly!) portable Delphi class which makes it possible to read UNIX-style tar files from your application. But first...

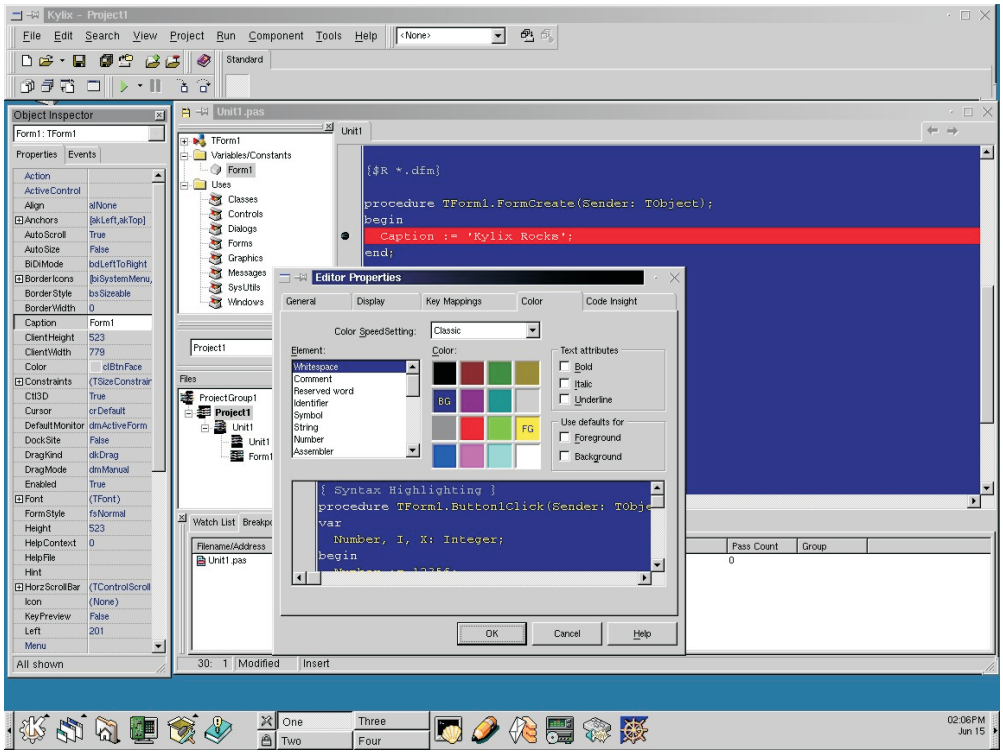
Removing Windowisms

Although I was quite pleased with last month's enhanced group box component, I won't pretend that I made any serious effort to turn it into a portable component. If you've got last month's article in front of you, take a long hard look at the `TCustomGroupBoxEx.PaintCaption` method where you will find a wide variety of misdemeanours from a cross-platform point of view.

To begin with, you'll notice that the code declares two variables of type `TLogFont` and `TTextMetric`, both of which are native Windows API types, this makes them an obvious no-no. The next problem comes a few lines down when I use the `OffsetRect` routine to reposition the rectangle which surrounds the caption text. Again, `OffsetRect` is a native Windows call, which is therefore out. In this particular case, the implementation of `OffsetRect` is so trivial that most programmers will probably roll their own equivalent in order to have the same routine under both Linux and Windows. Indeed, if we're lucky, Borland may even provide an implementation for us.

Things get somewhat dirtier as we delve deeper. Rather than figuring out the exact pixel location of the group box caption, and drawing it at that position, I sneakily relied upon the fact that Windows has a `DrawText` routine which can perform automatic left or right justification and centring of a string within a designated rectangle. This is why I set up the `Flags` variable and then called `DrawText` to do all the real work. Again, this isn't acceptable under Linux because of course `DrawText` is a Windows call.

Having said that, it's now public knowledge that the Linux equivalent of the VCL library is going to sit on top of Qt, the excellent C++ application framework from Troll Tech (recently relocated to www. trolltech.com). If you carefully read through the reference documentation for Qt, you'll find that the class



➤ *Borland have just released some exciting screenshots of an early version of the Kylix development system: in this case running under the KDE window manager.*

```
void QPainter::drawText (int x, int y, int w, int h, int tf, const QString & str,
  int len = -1, QRect * brect=0, char ** internal=0 )
```

➤ *Listing 1*

library often includes method calls which are more or less equivalent to procedural counterparts in Windows. `Draw- Text` is a good example of this, appearing as a method of the `QPainter` class, see Listing 1.

Although it's by no means obvious from this function prototype, the `tf` parameter is actually a mask of various bit-flags broadly equivalent to the formatting flags in the Win32 `DrawText` call, including those all-important text justification options. Thus, if you were porting my enhanced group box control over to Linux, you could always use a little conditional compilation to call either the Win32 `DrawText` or `QPainter::drawText` as appropriate. `QPainter` is the Qt equivalent of a Windows device context, so if you're doing any drawing, you're bound to have a `QPainter` object to hand!

You might argue that calling down to the Qt layer is messy, and that's undeniably true. But of course it's no messier than making Windows API calls from inside your nice, neat, object oriented VCL code. The bottom line is that a certain amount of conditional compilation and special case code is going to be inevitable, but with careful design this can be minimised, and preferably localised into one platform-dependent area of your code. Similar considerations apply to calls such as `SetTextColor`, `SetBkMode`, and so forth. Again, these routines have their equivalent `QPainter` methods in Linux-land.

Enter Delphi 6 And CLX

So does this mean that you've got to mess up your code with lots of conditional compilation statements? Well, not necessarily, no, there is another option. Something else which is now public knowledge (following a recent presentation by Charles Jazdzewski, Delphi's Chief Architect) is the relationship between Delphi 6, Kylix and CLX. CLX (pronounced 'clicks' in case you've not heard of it), is the new cross-platform component library which will replace the VCL library under Linux. I think it stands for **C**omponent **L**ibrary, **X**-Platform, or something like that. The key point here (and it's an exciting one!) is that Borland plan to implement CLX in Delphi 6, the next major release of the development system for Windows.

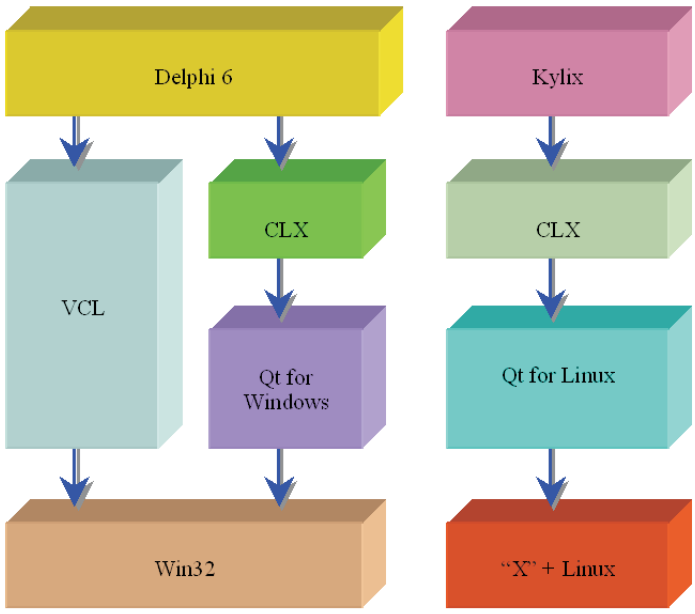
You can see what this means by referring to the overall architecture diagram which I've put together in Figure 1. Kylix developers (and bear in mind that this includes the Linux-hosted C++ development system as well as Object Pascal) will construct applications made up from CLX components and will use the CLX library just as you're used to using the VCL at the present time. They will, however, have the option of bypassing CLX and making 'raw' Qt calls in those cases where CLX doesn't provide the exact functionality that's required.

Strictly speaking, this is unlikely to be necessary, since we're told that the Linux implementation of CLX is going to be a complete 'wrap' of Qt. Hopefully, the `QPainter::drawText` functionality that I've alluded to earlier will be exposed as part of the CLX framework, as a new method of `TCanvas`, or whatever the CLX equivalent of `TCanvas` turns out to be.

On the Windows side, the developer has got two choices. You can either stick with the VCL library, or you can go down the CLX route. Delphi 6 will include a licensed version of Qt for Windows, meaning that it will be possible to create a 'one size fits all' code base that can be compiled under Delphi 6 and under Kylix. This is likely to be the most attractive option for those who want to port their software to Linux, but obviously a certain amount of re-engineering will be needed in making the move from VCL to CLX.

There are certain areas which will clearly require more work than others. For example, even the 'purest' (in the sense of not making any Windows API calls) Delphi program will very likely need modifications in the area of filenames and pathnames. As you may know, Linux doesn't have the concept of drive letters. Instead, the entire Linux file system descends from the root directory

➤ *Figure 1: With Delphi 6, developers will have two possible implementation routes: continue to use the VCL, offering maximum Windows-specific features, or go down the CLX route, which will provide a high degree of source code compatibility with Kylix.*



```
unit TarUnit;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls;
type
  TForm1 = class(TForm)
    Files: TListView;
    Button1: TButton;
    OpenDialog1: TOpenDialog;
    Label1: TLabel;
    SaveDialog1: TSaveDialog;
    procedure FilesDblClick(Sender: TObject);
    procedure Button1Click(Sender: TObject);
  private
  public
  end;
var
  Form1: TForm1;
implementation
{$R *.DFM}
type
  TTarFileEntry = class(TObject)
  private
    name, uid, gid: String;
    mode, size: Integer;
    mtime: TDateTime;
    FileOffset: Integer;
  end;
  TTarFile = class (TObject)
  private
    FileList: TStringList;
  public
    constructor Create (const FileName: String);
    destructor Destroy; override;
  end;
// TTarFile
function OctalToInt (Str: String): Integer;
var
  Idx: Integer;
begin
  Result := 0;  Str := Trim (Str);
  for Idx := 1 to Length (Str) do begin
    if not (Str [Idx] in ['0'..'7']) then Exit;
    Result := (Result shl 3) + Ord (Str [Idx]) - Ord ('0');
  end;
end;
function UnixTimeToFileTime (UnixDateTime: TLargeInteger):
  TDateTime;
var
  Time: Integer;
  LocalTime: TFileTime;
  FileTime: TFileTime absolute UnixDateTime;
begin
  UnixDateTime := (UnixDateTime + 11644473600) * 10000000;
  FileTimeToLocalFileTime (FileTime, LocalTime); // WIN32!!
  FileTimeToDosDateTime (LocalTime, LongRec (Time).Hi,
    LongRec (Time).Lo); // WIN32!!
  Result := FileDateToDateTime (Time);
end;
function PermissionsToStr (Perm: Integer): String;
  function PermFlags (bits: Integer): String;
  begin
    Result := '---';
    if (bits and 4) <> 0 then Result [1] := 'r';
    if (bits and 2) <> 0 then Result [2] := 'w';
    if (bits and 1) <> 0 then Result [3] := 'x';
  end;
begin
  // Display order is owner-group-other
  Result := PermFlags (Perm shr 6) + PermFlags(Perm shr 3)
    + PermFlags (Perm);
end;
constructor TTarFile.Create (const FileName: String);
type
  TarHeader = record
    name: array [0..99] of Char;  // name of the file
    mode: array [0..7] of Char;   // permission bits
    uid: array [0..7] of Char;    // owner - user ID
    gid: array [0..7] of Char;    // owner - group ID
    size: array [0..11] of Char;  // size of this file
    mtime: array [0..11] of Char; // file modification time
    chksum: array [0..7] of Char; // checksum for file header
    linkflag: Char;
    linkname: array [0..99] of Char;
    magic: array [0..7] of Char;
    uname: array [0..31] of Char;
    gname: array [0..31] of Char;
    devmajor: array [0..7] of Char;
    devminor: array [0..7] of Char;
  end;
var
  fs: TFileStream;
  Header: TarHeader;
  NextBlock: Integer;
  entry: TTarFileEntry;
begin
  Inherited Create;
  FileList := TStringList.Create;
  if FileExists (FileName) then begin
    fs := TFileStream.Create (FileName, fmOpenRead);
    try
      while fs.Position < fs.Size do begin
        NextBlock := fs.Position + 512;
        fs.Read (Header, sizeof (Header));
        if Header.name = '' then break;
        entry := TTarFileEntry.Create;
        entry.name  := Header.name;
        entry.mode  := OctalToInt (Header.mode);
        entry.size  := OctalToInt (Header.size);
        entry.mtime := UnixTimeToFileTime(
         OctalToInt(Header.mtime));
        entry.FileOffset := NextBlock;
        if Trim (Header.magic) = 'ustar' then begin
          entry.uid   := Trim (Header.uname);
          entry.gid   := Trim (Header.gname);
        end else begin
          entry.uid   := Trim (Header.uid);
          entry.gid   := Trim (Header.gid);
        end;
        FileList.AddObject (entry.name, entry);
        fs.Position := NextBlock + ((entry.size + 511)
          div 512) * 512;
      end;
      finally
        fs.Free;
      end;
  end;
end;
destructor TTarFile.Destroy;
var Idx: Integer;
begin
  for Idx := FileList.Count - 1 downto 0 do
    FileList.Objects [Idx].Free;
  FileList.Free;
  Inherited Destroy;
end;
procedure TForm1.FilesDblClick(Sender: TObject);
var
  Item: TListItem;
  FileName: String;
  Size, Offset: Integer;
  function DeUnix (const Path: String): String;
  var Idx: Integer;
  begin
    Result := Path;
    for Idx := 1 to Length (Result) do
      if Result [Idx] = '/' then Result [Idx] := '\';
  end;
  procedure ExtractFile (const Archive, Dest: String;
   Offset, Size: Integer);
  var
    sArchive, sDest: TFileStream;
  begin
    sArchive := TFileStream.Create (Archive, fmOpenRead);
    try
      sDest := TFileStream.Create (Dest, fmCreate);
      try
        sArchive.Position := Offset;
        sDest.CopyFrom (sArchive, Size);
      finally
        sDest.Free;
      end;
    finally
      sArchive.Free;
    end;
  end;
begin
  if Files.Items.Count = 0 then
      ShowMessage ('Please open a tar file first')
  else begin
    Item := Files.Selected;
    if Item <> Nil then begin
      Size := StrToInt (Item.SubItems [2]);
      if Size = 0 then
        ShowMessage ('Can only extract physical files')
      else begin
        Offset := StrToInt (Item.SubItems [6]);
        FileName := ExtractFileName (DeUnix (Item.Caption));
        if MessageDlg('Extract ' + FileName + '?',
          mtConfirmation, [mbYes, mbNo], 0) = mrYes
          then begin
          SaveDialog1.FileName := FileName;
          if SaveDialog1.Execute then
            ExtractFile (OpenDialog1.FileName,
              SaveDialog1.FileName, Offset, Size);
        end;
      end;
    end;
  end;
end;
procedure TForm1.Button1Click(Sender: TObject);
var
  Idx: Integer;
  tar: TTarFile;
  Item: TListItem;
  Entry: TTarFileEntry;
begin
  if OpenDialog1.Execute then begin
    tar := TTarFile.Create (OpenDialog1.FileName);
    try
{ CONTINUED ON FACING PAGE }
```

```
{ CONTINUED FROM FACING PAGE}
      Files.Items.Clear;
      for Idx := 0 to tar.FileList.Count - 1 do begin
        Entry := TTarFileEntry (tar.FileList.Objects [Idx]);
        Item := Files.Items.Add;
        Item.Caption := Entry.name;
        Item.SubItems.Add (FormatDateTime ('dd/mm/yyyy',
          Entry.mtime));
        Item.SubItems.Add (FormatDateTime ('hh:mm:ss',
          Entry.mtime));
        Item.SubItems.Add (IntToStr (Entry.size));
        Item.SubItems.Add (PermissionsToStr (Entry.mode));
        Item.SubItems.Add (Entry.uid);
        Item.SubItems.Add (Entry.gid);
        if Entry.size <> 0 then Item.SubItems.Add ('$' +
          IntToHex (Entry.FileOffset, 8));
      end;
    finally
      tar.Free;
      Label1.Visible := Files.Items.Count > 0;
    end;
  end;
end;

end.
```

➤ *Facing page and above: Listing 2.*

named '/', ie a single forward slash. So drive letters are out, and so are backslashes as path separators. A less obvious, but no less important, consideration is the fact that Linux has a case-sensitive file system. Thus, `MyApp.dat` is a completely different file from `Myapp.dat`. Both files can happily coexist in the same directory, and if you don't get your capitalisation right, you'll end up referencing the wrong file. For the same reasons, unit names are case sensitive when referenced from a `uses` clause. I imagine that Borland will provide new implementations of `ExtractFilePath`, `ExtractFileName`, etc, which, as far as possible, do the right thing on either platform.

The TAR File Format

With that as an introduction, let's take a look at the `tar` file format, something that's going to become increasingly relevant to Windows programmers who start venturing into the world of UNIX/Linux development, and need to read Linux-originated file distributions. `tar` is actually a *very* old file format, when I tell you that `tar` stands for 'Tape Archive', you'll realise just how old this format is. That's right folks, we're talking about paper tape readers, paper tape punches and all the fun things that your humble scribe used to play with when I started at university some twenty five years ago.

I chose the `tar` file format for one reason only: it's simple! A tar archive can contain multiple files, but, here's the important bit, none of the data within those files is compressed. You can think of the tar file as a straight concatenation of multiple files, with a special file header preceding the data for each file. Think of it as a ZIP file without data compression, if you like. Interestingly, the UNIX/Linux programming community have a compressor/decompressor utility called gzip which is the reverse of tar: it does provide data compression, but it only works with a single file! Since gzip only works with one file and tar doesn't provide compression, you won't be surprised to hear that Linux programmers routinely use tar to group a set of related files into a single archive, and then use gzip to compress that one file. Sneaky, huh? This is why you often see Linux-related file distributions with file names such as xxxx.tar.gzip. This naming convention indicates that you should decompress first, and then extract the files from the resulting tar file.

At this point, some of you are no doubt thinking that if these Linux gurus are so smart, why don't they just write a decent ZIP utility that stores multiple files into one *compressed* archive? Well, I've got a lot of sympathy with that position and, as it happens, PKZIP compatible utilities with names such as zip and unzip do exist. However, the majority of Linux developers still seem to favour this hybrid use of tar and gzip, and therefore we're stuck with tar files for some time to come. Truth to tell, putting archiving capabilities (file grouping) into one utility and compression/decompression capabilities into another utility is consistent with the classical UNIX philosophy of getting a single program to do one thing and do it well.

Each file in the tar archive is preceded with a record which looks like that shown in Listing 2, the `TarHeader` record which is defined inside the `TTarFile.Create` constructor. The first big surprise here is that all the fields are characters. I suspect that this is a historical hangover from the days of paper tape: some systems were seven bit only whereas others were eight bit and could therefore cope with every possible bit value.

Briefly, the meaning of the various fields is as follows.

The `name` field specifies the name of the file that follows this header. As with WinZip and more modern archive formats, tar is capable of recording directory information, and the header might therefore represent the presence of a directory, rather than an actual file.

The `mode` field is used to represent standard UNIX file permissions for owner, group and other. The lower three bits represent other permissions, the next three bits represent the group permissions and the final three bits represent owner permissions. This yields the equivalent bit-mask declarations shown in Listing 3, although I haven't used these in my code. And (yes, you've guessed!) this field, like all other 'numeric' fields in the header, is actually a number that's been

➤ *Listing 3*

```
const
  PermOtherExec   =   $1;   // other, execute/search
  PermOtherWrite  =   $2;   // other, write
  PermOtherRead   =   $4;   // other, read
  PermGroupExec   =   $8;   // group, execute/search
  PermGroupWrite  =   $10;  // group, write
  PermGroupRead   =   $20;  // group, read
  PermOwnerExec   =   $40;  // owner, execute/search
  PermOwnerWrite  =   $80;  // owner, write
  PermOwnerRead   =   $100; // owner, read
```

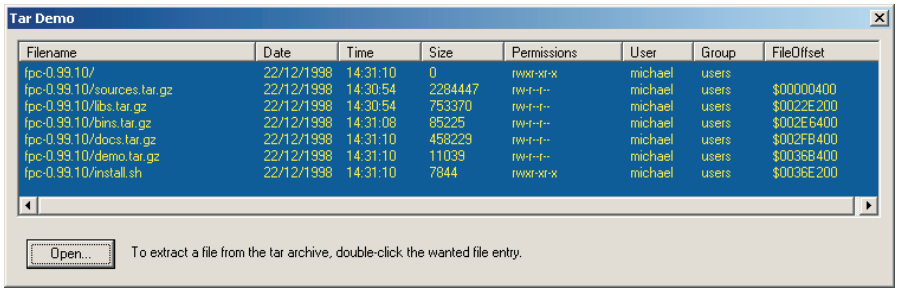
encoded as an *octal* string! Retro computing rules OK!

The `uid` field is an octal string that is a number corresponding to the user ID of the file owner.

The `gid` field is another encoded number that corresponds to the group ID of the file owner.

The `size` field, as you'd expect, is the size of the file. It can encode a total of twelve octal digits, giving us a potential range of 8^{12} which corresponds to a maximum file size of 64Gb. If the tar file entry isn't a physical file (eg a directory entry or a representation of a UNIX-style link) then the `size` field will be zero. One important point that I've not mentioned thus far is the block-structured nature of tar files. You can think of a tar file as being divided up into 512 byte blocks, something else which I suspect is a historical hangover from the paper tape days. These file headers are always aligned on 512-byte boundaries and even if a file occupies a single byte, it will still have 512 bytes allocated to it in the file. Now you see why it's such a good idea to compress a tar file: it's a very inefficient archive format for storing large numbers of small files.

The `mtime` field is the modification date of the file expressed in Coordinated Universal Time: the number of seconds that have elapsed since 1st January 1970. And thereby hangs a tale. As far as I know (please correct me if I'm wrong), Windows doesn't actually have a built-in function for converting Coordinated Universal Time into the more familiar time/date formats used by Delphi. After a certain amount of hunting around (Cathy: this is why the article was late; am I forgiven? *[Dave: have you tried sending her cream cakes? Ed]*) I managed to find an MSDN article which explains how to do this. Unfortunately, it uses a couple of Windows API calls. Arghhhh, horror! This is why I cunningly describe my `TTarFile` class as being 'mostly portable'. Hopefully, Borland will be able to provide a sufficiently rich set of date/time conversion routines in CLX as to eliminate this Win32 dependency. Time will tell.



➤ *Figure 2: My little demo program can display the files in a tar archive and allows you to remove files on an individual basis.*

The `chksum` field is used to provide a checksum. Somewhat counter-intuitively, this is only used to check the integrity of the file header, not of the file itself. Error checking: what error checking? The checksum is calculated by adding together all the bytes that make up the file header.

The remaining fields are, for the most part, not especially relevant when accessing a tar file from a Windows environment. The `linkflag` field is perhaps the most important and indicates whether we're dealing with a directory, a regular file, or whatever. The possible values of this field are described in more detail in the tar.txt file which you'll find on the companion disk. Incidentally, after spending an hour trawling the net, this was the only reasonably complete definition of the tar file format that I was able to find. The `linkname` field and the other fields are only relevant for certain types of files that aren't relevant under Windows. An exception is the `magic` field which, if it contains the special string `ustar`, indicates that the `uname` and `gname` fields are valid, containing an alphanumeric representation of the user and group owner fields which should be used in preference to the previously discussed `uid` and `gid` fields.

A Simple TAR Viewer/Extractor Utility

With this as background, Figure 2 shows a simple program which provides the ability to scan a `tar` file, display a directory listing and extract a specified file from the archive, the complete source for this being shown in Listing 2. I haven't bothered to separate out the `TTarFile` class into a reusable component in its own unit, but this can easily be done if desired. For the purposes of simplicity, I've also restricted this code to reading and extracting: it can't create new `tar` files, nor alter existing ones. It wouldn't be too difficult to add these capabilities but, to be honest, I don't think there's much mileage to be gained from prolonging the life of antique, brain-dead file formats. The emphasis here is on accessing tar files from a Delphi or Kylix Pascal application.

As you can see, the program uses a `TListView` control (yes, this control will undoubtedly be present in Kylix, courtesy of the existing `QListView` control which forms part of the extensive widget collection offered by the Qt class library). The `TTarFile` class is responsible for parsing a `tar` file and converting it into a form which is more easily digestible by Delphi applications, namely a `TStringList` contains a list of all the files, directories, links, etc, contained within the tar file. This class is very simple, with a constructor and destructor being the only introduced methods. You could easily turn it into a drop-in, reusable component, but this would obviously involve extra work. The associated `Objects` property of the string list contains an array of `TTarFileEntry` objects, each of which provides more detailed information on that particular file entry. You'll notice that I haven't bothered copying all the information from the tar header record, it's simply not needed here.

Next, the source code includes a few utility routines. Firstly, there's `OctalToInt` which does exactly

what it says on the tin. This routine is extensively used to convert octal number representations into a plain-vanilla integer. You'll also see that this routine makes use of the `Trim` function. This is actually very important when working with tar file headers because many of those octal strings are padded with spaces. Read the tar.txt file for more information on this.

Next comes the deeply impenetrable `UnixTimeToFileTime` routine. No, I don't have a clue where that number `11644473600` comes from! Suffice to say that it just works. Firstly, the code converts a UNIX date/time field (expressed here as a 64-bit integer) into a `TFileTime` structure as used by the Windows API. Next the `FileTimeToLocalFileTime` routine is used so as to express the date/time with respect to the current locale, and next it is converted to a DOS format date/time before finally massaging it into the much more useful `TDateTime` format which all VCL programmers know and love. I've flagged those two dastardly Win32 API routines for subsequent removal. If anyone knows of a simple, *platform independent* way to do these date/time conversions, then I'd like to hear from you.

The last of our utility functions takes the file permissions field and converts it into the human-readable format familiar to UNIX programmers. The permission bits are expressed in the standard order: owner, group, other. In other words, three permission bits for the owner, followed by three for the group and then three for anyone else. The code works by right-shifting the permissions field and calling `PermFlags` for owner, group and other in turn.

Moving swiftly on, the `TTarFile.Creator` constructor is responsible for opening the tar archive and scanning through it in search of file headers. Firstly, it creates the `FileList` object which is used to store our list of files. Next, it verifies that the archive exists and opens a file stream object in the normal way. At this point, a little validation code (ie, is this really a tar file?) wouldn't go amiss, but this is left as an exercise to the reader! Probably the simplest thing would be to check that the `Header.magic` field is one of the recognised values, as defined in the tar.txt file.

The code then iterates through the file, reading successive file headers. An interesting 'gotcha' of tar files is their ability to include an arbitrary number of zero bytes following the last file's data. Again, this shouldn't surprise anyone who, like your scribe, is old enough to remember the trailer section associated with a paper tape. In order to detect an end of file condition, I check for a file entry which has a null file name. If you don't implement this check, then you'll find that the file list will fill up with a number of 'empty' file entries.

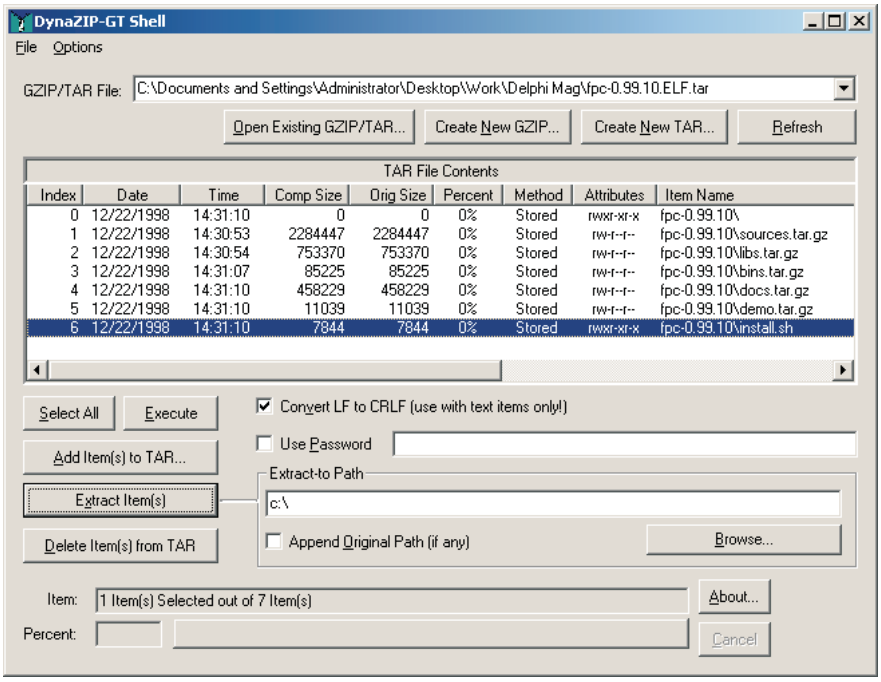
Immediately prior to reading the file header, 512 bytes are added to the current stream position and saved into `NextBlock`. This designates the position of the next file header, assuming that we are

dealing with a zero-length file such as a link or directory entry. If it's not a zero-length file, then `NextBlock` gives us the byte offset into the archive of where the file data can be found. For each file entry, a new `TTarFileEntry` object is created and the various fields initialised. As I mentioned earlier, the `Header.magic` field indicates whether we're dealing with numeric or ASCII user/group file owner information, and the `uid` and `gid` fields are set up accordingly.

If we're dealing with a real file (ie one that occupied more than zero bytes in the tar file!) we have to calculate the position of the next file header. This is done by adding 511 to the file size and then dividing by 512, thus giving the total number of 512-byte blocks occupied by the file. This is then multiplied by 512 and added to the `NextBlock` value so as to give the next file header position. Finally, when everything has been read, the file stream object is destroyed and the routine terminates.

Implementation of the class destructor is trivial, it just walks the file list, destroying all the `TTarFileEntry` objects that were previously created. The `Button1Click` routine corresponds to pressing the `Open` button in the screenshot. This code simply prompts the user for a `tar` file to open using a `TOpenFileDialog` in the usual way, creates a corresponding `TTarFile` object and then loads the associated data into the `TListView` control which is operating in report mode.

Finally, the `FilesDblClick` routine is called whenever an entry in the list view is double clicked. It retrieves the corresponding file size and file offset, together with the name of the file. An error message is displayed if the user clicked a zero-length file corresponding to a directory or link, though I suppose in a full featured visual `tar` program you might want to give the user the option of 'extracting a directory', thereby creating a folder. One wrinkle here is the need to remove those forward slash characters from the pathname, replacing them with backslashes.



➤ *Figure 3: This is the DynaZIP-GT Shell program, part of the recently introduced DynaZIP-GT programmers' toolkit which you can find at www.innermedia.com. This product offers full compatibility with both tar and gzip files.*

If you don't do this, the `ExtractFileName` routine isn't much use. This is accomplished by the `DeUnix` routine.

Finally, the code requests confirmation that you want to extract a file and calls `ExtractFile` to do the business, reopening the archive and using the handy-dandy `CopyFrom` method to move the wanted file data from the source to the destination stream. Easy peasy.

Conclusions

I wouldn't dare pretend that this is a fully-featured tar extractor by any means. You might want to add the ability to extract to the relative path specified by the file entries within the `tar` file, add a facility to perform automatic conversion of LF sequences to CR/LF, and so forth. The goal here is simply to provide a basis for your own tar-related utilities and, at the same time, discuss Kylix portability with some real code. As you can see, I didn't succeed in creating something that was 100% portable, thanks to those two date/time related Windows API calls, but I don't doubt that there'll be some way around that problem once Kylix arrives. Until then, have fun and happy (cross-platform) programming.

Inner Media have recently released DynaZIP-GT, a fully-featured commercial toolkit which can be used to work with both tar and gzip files. It's compatible with Delphi, C++Builder, and in fact most Windows programming environments. You can download a *full* version (no time-bombs, no crippled features) for evaluation purposes from www.innermedia. com, and the distribution includes the useful demo program shown in the screenshot above.

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