Reading and writing archives in Free Pascal

Michaël Van Canneyt

July 26, 2009

Abstract

Often one needs to access or create archives in an application: Creating backups or reading backups is just one of the reasons. Free Pascal by default has native support for several popular archive formats: zip, bzip2, tar and gzip. Often these archives are also encoded or encrypted: support for encryption is also present. This article presents an overview of the possibilities.

1 Introduction

Every now and then, a programmer is faced with the task to produce some kind of backup or to open some kind of archive in his program. Many kinds of archives are available, and meanwhile many compression methods exist. Free Pascal has support for some of the more popular formats:

zip Support for reading and writing zip files (using the deflate algorithm).

gzip Support for reading and writing .gz files (using the deflate algorithm).

bzip2 Support for reading .bz2 files (using the BWT algorithm).

tar Support for reading and writing .tar files (standard Posix Tape ARchive format).

A third-party package exists which can handle 7z archives, but this is not distributed with Free Pascal.

Additionally, Free Pascal has built-in support for some encoding and encyption algorithms:

Blowfish encryption and decryption.

Base64 encoding and decoding: MIME encoding.

Ascii85 encoding and decoding: Ascii85 encoding is found in PDF and Postscript files.

IDEA encoding and decoding (IDEA is used in PGP).

calculate MD5 checksums.

Unix crypt (one way encryption using the DES algorithm).

Not all of these are usable for encryption of files, but they are frequently encountered when dealing with archives.

In the following paragraphs, we'll describe how these archives can be accessed.

2 A note on architecture

The Free Pascal support for all these encoding, encryption and compression formats is not unified in a single library or component with a common API, such as the (open source) Abbrevia suite from TurboPower. However, the mechanism used is roughly the same in all cases and is centered around the use of the `TStream` class.

For decoding or uncompression algorithms, the data to be decoded or decompressed is assumed to be available in a stream. This stream is used as the source of a second stream, from which the decoded or decompressed data can be read: the read operation decodes or decompresses the data in the source stream on the fly. Roughly, this would look as follows:

```
Var
  Source : TStream;
  Data : TDecodingStream;
  Buf : Array[1..SomeSize] of byte;
begin
  Source:=GetSourceStream;
  Data:=TDecodingStream.Create(Source);
  Data.Read(Buf,SizeOf(Buf));
end;
```

After the read operation, the data in the `buf` buffer is decoded or decompressed. The Read operation will read as much data from the Source stream as is needed to fill the `buf` buffer.

Conversely, the encoding or compression algorithms uses also 2 streams. First, a destination stream is created: this is the location where the compressed or encoded data should be written. Secondly, a compression/encoding stream is created which uses the first stream as output. Any data that is written to the second stream will be compressed or encoded on the fly. This could look as follows:

```
Var
  Dest : TStream;
  Data : TEncodingStream;
  Buf : Array[1..SomeSize] of byte;
begin
  Dest:=CreateDestStream;
  Data:=TEncodingStream.Create(Dest);
  Data.Write(Buf,SizeOf(Buf));
end;
```

The write operation on `Data` will take the data from `buf`, perform any needed compression or encoding, and will then write the data to the `Dest` stream.

This mechanism can be used to create chains:

```
Var
  Dest : TStream;
  Encode : TEncodingStream;
  Comp :TCompressionStream;
  Buf : Array[1..SomeSize] of byte;
begin
  Dest:=CreateDestStream;
  Encode:=TEncodingStream.Create(Dest);
  Comp:=TCompressionStream.Create(Encode);
  Comp.Write(Buf,SizeOf(Buf));
end;
```

In this scenario, the data is compressed as it is written to the `Comp` scenario, and then encoded (for example UUEncoded) as it is written to the `Encode` stream. Finally it ends up in `Dest`, which can be e.g. a file stream.

3 Deflate algorithm and gzip

The gzip program (commonly found on unices) produces a .gz file from any file you feed to it. It uses the deflate algorithm for this, and Free Pascal supports compression and decompression using this algorithm. The zstream unit contains 3 classes for this:

TCompressionStream This TStream descendent compresses (using deflate) any data written to it, and writes it to a destination stream. It cannot read data, only write.

TDeCompressionStream This TStream descendent takes a source stream, and whenever data is read from the stream, it reads data from the source stream and decompresses it (using inflate) on the fly. It only reads data

TGZFileStream This TStream descendent can be used to read a .gz file. This is a file written using the deflate algorithm, but which has some extra data prepended to it. This class takes care of the extra data.

The data written by a `TCompressionStream` can only be read by a `TDeCompressionStream` class. If the produced file needs to be read by any other software, it is advisable to use the TGZFileStream class.

Usage is very simple, and follows the general architecture as outlined above. The following very simple program compresses a file using the deflate algorithm:

```
program deflate;

uses Classes, ZStream;

Var
  Src,Dest : TFileStream;
  FN : String;
  Comp : TCompressionStream;

begin
  FN:=ParamStr(1);
  Src:=TFileStream.Create(FN,fmOpenRead);
  try
    FN:=FN+'.z';
    Dest:=TFileStream.Create(FN,fmCreate);
    Comp:=TCompressionStream.Create(clDefault,Dest);
    try
      Comp.SourceOwner:=True;
      Comp.CopyFrom(Src,0);
    finally
      Comp.Free;
    end;
  finally
    Src.Free;
  end;
end.
```

The constructor `Create` of the `TCompressionStream` has 2 mandatory arguments: the first is the compression level (one of `clnone`, `clfastest`, `cldefault` or `clmax`) and the second is the destination stream. The `OwnerSource` property tells the compression stream that it owns the destination stream: when it is freed, it should also free the `Dest` stream.

The standard `TStream.CopyFrom` then copies the contents of the `Src` stream to the `Dest` stream, compressing it on the fly.

To inflate the files created using the above program, the following 'inflate' program can be used:

```
program inflate;

uses SysUtils,Classes, ZStream;

Var
  Src,Dest : TFileStream;
  FN : String;
  DeComp : TDeCompressionStream;
  Buf : Array[1..1024] of byte;
  Count:  Integer;

begin
  FN:=ParamStr(1);
  Src:=TFileStream.Create(FN,fmOpenRead);
  DeComp:=TDeCompressionStream.Create(Src);
  try
    DeComp.SourceOwner:=True;
    FN:=ChangeFileExt(FN,'');
    Dest:=TFileStream.Create(FN,fmCreate);
    try
      Repeat
        Count:=DeComp.Read(Buf,SizeOf(Buf));
        Dest.Write(Buf,Count);
      Until (Count<SizeOf(Buf));
    finally
      Dest.Free;
    end;
  finally
    DeComp.Free;
  end;
end.
```

The general structure is the same as the deflate program, only the roles of the `Src` and `Dest` streams are reversed. Since it is impossible to determine the size of the stream to be decompressed, the `CopyFrom` method cannot be used, and therefor a loop is constructed which reads data from the decompression stream and writes it to the destination stream. The loop ends when no more data can be read from the decompression stream.

The `TGZFileStream` takes a different approach, it behaves more like a regular `TFileStream`. It's constructor takes as parameters the name of a file to open or create, and a file mode, which is one of `gzopenread` or `gzopenwrite`: the former for reading the file, the latter for creating one. Other than that, it is like any other stream. The following program will compress a file to a .gz file:

```
program pgzip;

uses SysUtils,Classes, ZStream;

Var
  FN : String;
  Dest : TGZFileStream;
  Src : TFileStream;

begin
  FN:=ParamStr(1);
  Src:=TFileStream.Create(FN,fmOpenRead);
  try
    FN:=FN+'.gz';
    Dest:=TGZFileStream.Create(FN,gzOpenWrite);
    try
      Dest.CopyFrom(Src,0);
    finally
      Dest.Free;
    end;
  finally
    Src.Free;
  end;
end.
```

The structure is similar to the deflate program, except that only 2 streams are needed: a `TGZFileStream` instance and a `TFilestream` instance. A decompression program can be made just like the inflate program: the sources of a pgunzip program are on the disc accompagnying this issue.

The support for bzip2 archives is currently limited to decompression only: the bzip2stream unit contains a class `TDecompressBzip2Stream` which decompresses a source stream just like the `TDeCompressionStream` does for the deflate algorithm. The sources of a pbunzip2 program are on the disc accompagnying this issue.

Note that version 2.2.4 of FPC contains only an 'bzip2' unit which contains an old TP-style stream object. The bzip2stream unit appeared only in the recent 2.3.1 versions of FPC, therefor it has been included with the sources of the pbunzip2 program.

When files are sent over internet, they are most often encoded in base64-encoding (or MIME encoding). This is used to convert a binary file to a text format (it uses only 7-bit characters). The base64 unit offers 2 components to deal with such files: `TBase64DecodingStream` for decoding, and `TBase64EncodingStream` for encoding. The base64 directory contains 2 programs (encodebase64 and decodebase64) that use these streams to encode/decode a file in base64 format: the structure of these programs is exactly the same as the examples given earlier, therefor the source is not presented here.

Note that many tools which handle base64 encoding (such as uuencode or mpack) write header lines such as the following before the actual data:

```
begin-base64 644 bzip2stream.o
f0VMRgIBAQAAAAAAAAAAAAAAEAPgABAAAAAAAAAAAAAAAAAAAAAAAAAAALhLAAAA
```

These first lines contain some metadata: they should be discarded from the input given to `TBase64DecodingStream`.

Similar to the Base64 encoding, the ASCII85 encoding encodes arbitrary data in a human-readable format which uses only readable characters to encode the data. The output files

generated by the ASCII85 encoding are smaller than the base64 encoding. The PDF and PostScript formats use it for embedded data. The ascii85 unit implements a `TASCII85DecoderStream` class which works exactly like the `TBase64DecodingStream`, and a `TASCII85EncoderStream` which works like the `TBase64EncodingStream`: the ascii85 directory contains 2 small programs that demonstrate the use of these components. Since the FPC 2.2.4 release contains only the decoder, the version of the ascii85 unit that is distributed with FPC 2.3.1, is also included in this directory.

4 Encryption using Blowfish

The BlowFish algorithm is a popular encryption algorithm, which can be used to encrypt data. The blowfish unit contains 2 classes: `TBlowFishEncryptStream` and `TBlowFishDecriptStream` which encrypt or decrypt an input stream using the blowfish algorithm. The blowfish directory contains 2 example programs, which encrypt or decrypt a file, their structure is the same as all other examples. Additionally, a small demo application exists (demoblowfish) which shows how to encrypt a piece of text, and create a human-readable version of the encrypted text.

The demo application contains 2 edits: one for the text to encrypt, another for the key phrase. A memo to show the result and a button to perform the encryption are also added, and last but not least, a checkbox is added which, when checked, will force the application to perform the decryption as well.

The OnClick event of the button performs the following code:

```
procedure TMainForm.BEncryptClick(Sender: TObject);

Var
  O,K,S,R,Msg : String;

begin
  O:=EText.Text;
  K:=EKey.Text;
  S:=DoEncrypt(O,K);
  MEncrypted.Lines.text:=S;
  If CBCheck.Checked then
    begin
    R:=DoDecrypt(S,K);
    If (O<>R) then
      Msg:=Format('Decryption failed: "%s" <> "%s"',[O,R])
    else
      Msg:='Decryption succesful !';
    ShowMessage(Msg);
    end;
end;
```

The real work is done in the `DoEncrypt` function:

```
Function DoEncrypt(Const AText, AKey : String) : String;
Var
  Src,Dest : TStringStream;
  Enc : TBlowFishEncryptStream;
  B : TBase64EncodingStream;
  K : TBlowFishKey;
```

```
  KL : Integer;
  T : String;

begin
  B:=Nil;
  T:=AKey;
  KL:=Length(T);
  If KL>56 then
    KL:=56;
  Move(T[1],K,KL);
  Dest:=TStringStream.Create('');
  try
    B:=TBase64EncodingStream.Create(Dest);
    try
      Enc:=TBlowFishEncryptStream.Create(K,KL,B);
      try
        Enc.Write(AText[1],Length(AText));
      finally
        Enc.Free;
      end;
    finally
      B.Free;
    end;
    Result:=Dest.DataString;
  finally
    Dest.Free;
  end;
end;
```

This function shows the chaining of streams, and shows how to construct a blowfish key from a passphrase: a blowfish key consists of max 56 bytes: the first 56 (or less) characters of the passphrase are copied to it. After that, the output is set up: a `TStringStream` instance (`Dest`), which is used as output for a `TBase64EncodingStream` instance (`B`). The `TBlowFishEncryptStream` stream is created using the base 64 encoding stream as output. When the data is written to the encoding stream, the encoded data ends up in `Dest`, where it is collected for the result. The reverse operation is implemented along similar lines, and is shown in figure 1 on page 8.

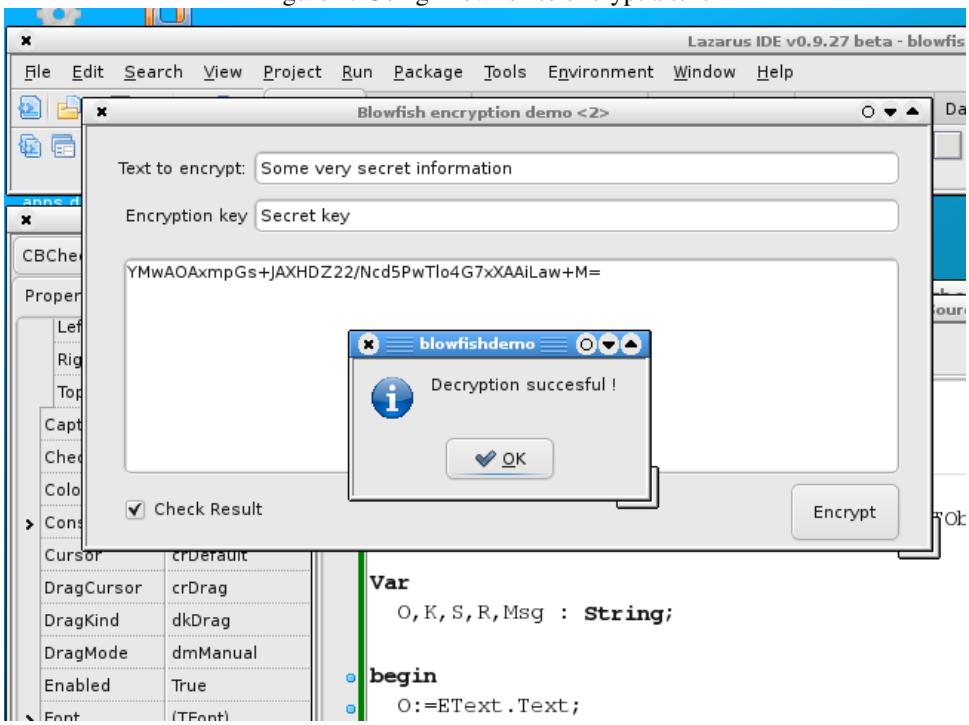
The IDEA encryption algorithm and streams function along similar lines: the IDEA directory contains the same encoding and decoding programs as the blowfish directory, and will not be presented here.

Both the Blowfish and IDEA encryption algorithms require the use of an encryption key. The sample code above demonstrates how to create such a key. The implementations of the IDEA and Blowfish streams have been enhanced in version 2.3.1 with overloaded versions of the constructor which accept a string as a key. These constructors then create the key structures from the passed string.

5 Zip files

The examples till now showed how to encrypt, compress or otherwise transform a single file. By contrast, Zip files or TAR files contain multiple files: the stream-based paradigm will not work for such files. Indeed, the mechanism used to read or create a zip file is very different.

Figure 1: Using Blowfish to encrypt a text



The zipper unit contains 2 classes: `TZipper` and `TUnzipper` which can be used to create zip files and examine or extract files from a zip file, respectively. The `TZipper` class does not support modifying existing zip files, just creating new ones from zero: this allows for a simpler implementation and interface.

The `TUnzipper` component has a simple interface:

```
TUnZipper = Class(TObject)
  Procedure UnZipAllFiles(AFileName : String);
  Procedure UnZipAllFiles; virtual;
  Procedure UnZipFiles(AFileName : String; FileList : TStrings);
  Procedure UnZipFiles(FileList : TStrings);
  Procedure Clear;
  Procedure Examine;
  Property FileName : String;
  Property OutputPath : String;
  Property Files : TStrings;
  Property Entries : TFullZipFileEntries;
end;
```

The `UnzipAllFiles` will simply extract all files from the archive `AFileName` (or the archive specified in the `FileName` property. The files are extracted relative to the `OutputDir` directory. The `UnzipFiles` call takes a list of filenames: the filenames must match the names of the files inside the archive. Optionally, this call also accepts an archive filename.

The `Examine` call will examine the archive in `FileName`, and will populate the `Files` and `Entries` properties: the former is a list of all filenames in the archive, the latter is a collection which contains all available information about the files in the archive. Its items are declared as follows:

```
TZipFileEntry = Class(TCollectionItem)
  Property ArchiveFileName : String;
  Property DiskFileName : String;
  Property Size : Integer;
  Property DateTime : TDateTime;
  property OS: Byte;
  property Attributes: LongInt;
end;

TFullZipFileEntry = Class(TZipFileEntry)
  Property CompressMethod : Word;
  Property CompressedSize : LongInt;
  property CRC32: LongWord;
end;
```

The names of the properties speak for themselves. The `TZipFileEntry` class is also used by the `TZipper` class.

Armed with these classes, we can make a .ZIP archive viewing application in no time. All we need is a Listview to display the contents, and the `TUnZipper` class; Adding some actions to open an archive and linking them to a main menu is standard procedure. The File-Open menu will open a dialog to select an archive, and then the `OpenArchive` method is called:

```
procedure TMainForm.OpenArchive(Const AFileName : String);

begin
  ClearData;
  FZip:=TUnZipper.Create;
  FZip.FileName:=AFileName;
  FZip.Examine;
  ShowEntries;
  Caption:=Format(SViewingFile,[AFileName]);
end;
```

The `ClearData` procedure clears the listview and destroys any previous instance of `TUnZipper`. Then a new instance is created and its `FileName` property is set to the selected filename. Calling the `Examine` method will fill the `Entries` collection, which is subsequently shown in the following method:

```
procedure TMainForm.ShowEntries;

Var
  I : Integer;
  LI : TListItem;
begin
  LVZIP.Items.BeginUpdate;
  try
    For I:=0 to FZip.Entries.Count-1 do
      begin
      LI:=LVZip.Items.Add;
      ShowEntry(LI,FZip.Entries[i]);
      end;
  finally
    LVZIP.Items.EndUpdate;
```

```
  end;
end;
```

The above is a simple loop over all items in the archive. Note that the loop is surrounded by a `BeginUpdate`..`EndUpdate` pair, which will avoid a repaint of the listview each time an item is added. The real work is done in the `ShowEntry` method, which converts the properties of the `TFullZipFileEntry` entry to a listview item:

```
procedure TMainForm.ShowEntry(LI : TListItem; Z : TFullZipFileEntry);

Var
  S : String;

begin
  LI.Caption:=ExtractFileName(Z.ArchiveFileName);
  LI.Data:=Z;
  With LI.SubItems do
    begin
    BeginUpdate;
    try
      S:=ExtractFileExt(Z.ArchiveFileName);
      If (S='') then // Directory entry
        S:=Z.ArchiveFileName;
      Add(S);
      Add(IntToStr(Z.Size));
      Add(DateTimeToStr(Z.DateTime));
      Add(IntTostr(Z.CompressedSize));
      If (Z.Size=0) then
        Add('0')
      else
        Add(Format('%5.2f',[Z.CompressedSize/Z.Size*100]));
      Add(ExtractFilePath(Z.ArchiveFileName));
    finally
      EndUpdate;
    end;
    end;
end;
```

Again, there is no special magic in this routine, except that the item (`Z`) is stored in the `Data` pointer of the listitem: this allows to access the collectionitem when the listview item is selected. Some special care must be taken to show directory entries correctly.

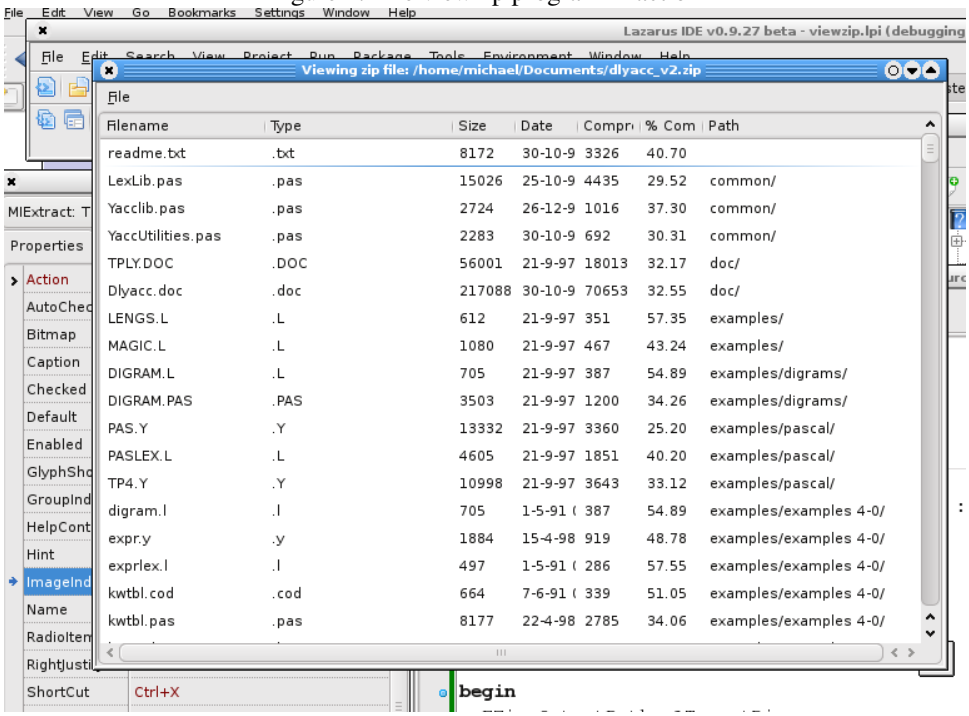
The listview is set to allow multi-selection of the items. An 'Extract' menu item is also added: it pops up a directory selection dialog, and then executes the following code:

```
procedure TMainForm.ExtractToDir(ATargetDir : String);

Var
  L : TStrings;
  I : Integer;
  Z : TFullZipFileEntry;

begin
  FZip.OutputPath:=ATargetDir;
  L:=TStringList.Create;
```

Figure 2: The viewzip program in action




```
  try
    For I:=0 to LVZip.Items.Count-1 do
      begin
      If LVZip.Items[i].Selected then
        begin
        Z:=TFullZipFileEntry(LVZip.Items[i].Data);
        L.Add(Z.ArchiveFileName);
        end;
      end;
    FZip.UnZipFiles(L);
  finally
    L.Free;
  end;
end;
```

A list of filenames (`L`) is constructed: the `TFullZipFileEntry` instance associated with each listview item is used to get the correct filename. After that, the list is passed to the `UnzipFiles` call.

That's all there is to it: it takes very little code to make a simple but functioning archive extraction program. It can be seen in action in figure 2 on page 11

Conversely, creating a zip archive is also deceptively simple. The `TZipper` offers a versatile interface for this:

```
TZipper=Class(TObject)
  Procedure ZipAllFiles; virtual;
  Procedure ZipFiles(FileList : TStrings);
  Procedure ZipFiles(Entries : TZipFileEntries);
  Procedure ZipFiles(AFileName : String; FileList : TStrings);
```

```
  Procedure ZipFiles(AFileName : String; Entries : TZipFileEntries);
  Procedure Clear;
Public
  Property BufferSize : LongWord;
  Property FileName : String;
  Property InMemSize : Integer;
  Property Files : TStrings;
  Property Entries : TZipFileEntries;
```

The `ZipAllFiles` call creates a zip archive in the file indicated by the `FileName` property, it uses all files in the varFiles property. The `ZipFiles` call explicitly passes on a filelist, or a collection of TZipFileEntry instances and optionally an archive filename is passed. Some properties exist to influence the behaviour of the compression: The `BufferSize` property determines what size the compression buffer should be (a reasonable default value is used). The `InMemSize` property determines till what size the files should remain in memory: files larger than the indicated size will be compressed in a temporary file on disk.

The following very simple command-line program acts as the zip command-line program: The first command-line argument is the name of a zip file to be created, and the other arguments are names of files to add to the archive:

```
program createzip;

uses
  Classes, SysUtils, Zipper;

Var
  Zip : TZipper;
  I : Integer;

begin
  If (ParamCount<2) then
    begin
    Writeln('Usage: createzip zipfilename filename1 [filename2]...');
    Halt(1);
    end;
  Zip:=TZipper.Create;
  try
    Zip.FileName:=Paramstr(1);
    For I:=2 to ParamCount do
      Zip.Files.Add(Paramstr(i));
    Zip.ZipAllFiles;
  finally
    Zip.Free;
  end;
end.
```

This small program is a fully functional zip archive creation program. The following command-line session shows it's functionality:

```
home: >createzip test.zip *.o
home: >unzip -l test.zip
Archive:  test.zip
  Length     Date   Time    Name
```

```
--------    ----   ----    ----
   10616  07-26-09 19:31   createzip.o
   53328  07-26-09 18:50   frmmain.o
  205184  07-26-09 18:55   viewzip.o
  163068  07-26-09 19:18   zipper.o
--------                  -------
  432196                  4 files
```

The `unzip` program is the standard infozip command-line program.

The `TZipper` class also has the possibility to create zip files from in-memory data, however it would go beyond the scope of this contribution to examine that functionality. The zipper unit in version 2.3.1 has some fixes for handling directories, file attributes, empty files and symbolic links. It is therefor is included in the sources accompagnying this article.

6 Tar files

The libtar unit contains the necessary classes to read or write .tar archives. As the zipper unit, it has 2 classes: one to read files (`TTarArchive`), the other to create them (`TTarWriter`). The interface is quite different to the zipper classes:

```
TTarArchive=Class(TObject)
Constructor Create (Stream : TStream)
Constructor Create (Filename : String);
Procedure Reset;
Function FindNext (Var DirRec : TTarDirRec) :Boolean;
Procedure ReadFile (Buffer : Pointer);
Procedure ReadFile (Stream : TStream);
Procedure ReadFile (Filename : String);
Function ReadFile : String;
end;
```

The constructor must be passed the filename of the archive or a stream with the contents of the archive. The `FindNext` call can be used to search the archive for the next file entry: if the call returns `True`, the `DirRec` parameter will contain all information of the found file entry. If the call returns `False`, the end of the archive is reached. The various `ReadFile` calls extract the last found entry to a memory buffer, stream instance or file on disk. The `Reset` call repositions the file position on the first entry in the archive.

This interface can be used to create a .tar viewing program, just like the zip viewing program. The main difference is the loop to fill the listview:

```
procedure TMainForm.ShowEntries;

Var
  I : Integer;
  LI : TListItem;
  D : TTarDirRec;

begin
  LVTar.Items.BeginUpdate;
  try
    While FTar.FindNext(D) do
      begin
```

```
      LI:=LVTar.Items.Add;
      ShowEntry(LI,D);
      end;
  finally
    LVTar.Items.EndUpdate;
  end;
end;
```

As can be seen, the loop is very simple. The `ShowEntry` is sligtly more complicated, as it must manually create a copy of the `TTarDirRec` record on the heap. The pointer to this record is stored in the `Data` pointer of the list item:

```
procedure TMainForm.ShowEntry(LI : TListItem; Const D : TTarDirRec);

Var
  S : String;
  E : PTarDirRec;

begin
  S:=ExtractFileName(D.Name);
  If (S='') then // Directory entry
    S:=D.Name;
  LI.Caption:=S;
  With LI.SubItems do
    begin
    BeginUpdate;
    try
      Add(ExtractFileExt(D.Name));
      Add(IntToStr(D.Size));
      Add(DateTimeToStr(D.DateTime));
      Add(ExtractFilePath(D.Name));
    finally
      EndUpdate;
    end;
    end;
  New(E);
  E^:=D;
  LI.Data:=E;
end;
```

To extract files from the Tar archive, the `ExtractFiles` is again used:

```
procedure TMainForm.ExtractToDir(ATargetDir : String);

Var
  L : TStrings;
  I : Integer;
  PD : PTarDirRec;
  D : TTarDirRec;

begin
  L:=TStringList.Create;
  try
    For I:=0 to LVTar.Items.Count-1 do
```

```
      begin
      If LVTar.Items[i].Selected then
        begin
        PD:=PTarDirRec(LVTar.Items[i].Data);
        L.Add(PD^.Name);
        end;
      end;
    FTar.Reset;
    While (L.Count>0) and FTar.FindNext(D) do
      begin
      I:=L.IndexOf(D.Name);
      If (I<>-1) then
        begin
        ExtractFile(ATargetDir,D);
        L.Delete(I);
        end;
      end;
  finally
    L.Free;
  end;
end;
```

The first part is simply collecting the filenames to be extracted. The second part does the actual extraction: The filepointer of the archive is put at the start of the archive, and all entries are scanned: as soon as a matching entry is found, it is extracted, and removed from the list of files to extract: as a consequence, the loop stops as soon as all files are extracted.

The `ExtractFile` checks the type of the entry and takes appropriate action depending on the type of entry:

```
procedure TMainForm.ExtractFile(Const ATargetDir : String; Const D : TTarDirRec);

Var
  ADir : String;
  AFileName : String;

begin
  AFileName:=ATargetDir+D.Name;
  if (D.FileType=ftNormal) then
    ADir:=ExtractFilePath(AFileName)
  else if (D.FileType=ftDirectory) then
    ADir:=AFileName;
  If Not ForceDirectories(ADir) then
    Raise Exception.CreateFmt(SErrCreatingDir,[ADir]);
  If (D.FileType=FTNormal) then
    FTar.ReadFile(AFileName);
end;
```

Contrary to the `TUnzipper` component, the `TTarArchive` class does not handle creation of directory entries or creation of directory parts of the files, so the `ExtractFile` routine must take care of this. Note that this routine does not handle symbolic links, file ownership or file mode: however, thise things can easily be added.

Finally, the `TTarWriter` class can be used to create archives. It's interface is quite simple:

```
TTarWriter = Class
  Constructor Create(TargetStream   : TStream);
  Constructor Create(TargetFilename : STRING; Mode : INTEGER = fmCreate);
  Procedure AddFile(Filename : STRING;  TarFilename : STRING = '');
  Procedure Finalize;
end;
```

Similar calls exist to create symbolic links, directory entries and so on. The createtar program demonstrates how this can be used to create a tar archive:

```
program createtar;

uses
  Classes, SysUtils, libtar;

Var
  Tar : TTarWriter;
  I : Integer;

begin
  If (ParamCount<2) then
    begin
    Writeln('Usage: createtar zipfilename filename1 [filename2]...');
    Halt(1);
    end;
  Tar:=TTarWriter.Create(Paramstr(1));
  try
    For I:=2 to ParamCount do
      Tar.AddFile(Paramstr(i));
    Tar.Finalize;
  finally
    Tar.Free;
  end;
end.
```

Note that this again does not take care of file ownership or permissions, but this is again easily added.

7 conclusion

Free Pascal/Lazarus is distributed by default with enough routines to handle most common archiving or encryption needs. The routines are basic and a common structure is not present, but unless one is writing a archive handling application such as winzip, winrar or ark, this is not really a necessity. The list of available algorithms grows steadily, but if the required algorithm is not yet included by default in Free Pascal, there are plenty of Object Pascal implementations available that handle almost any existing algorithm.