Lazarus Fishfacts: Paradox support for Lazarus/Free Pascal

Michaël Van Canneyt

January 29, 2008

Abstract

Delphi sill ships with the BDE engine and it's support for Paradox files. The use of the pxlib library now enables the use of Paradox files in Free Pascal and Lazarus.

1 Introduction

A regular question to the Free Pascal and Lazarus developers is: does Free Pascal/Lazarus support handling of Paradox files? Finally, this question can be answered in the affirmative.

Support for Paradox files has been in the Delphi IDE since the very beginning, using the BDE (Borland Database Engine), making it a popular format for many desktop applications. Indeed, it's FishFacts demo database was often used to showcase the database connectivity possibilities.

Finally, Free Pascal and Lazarus now also support handling of Paradox files: Thanks to the open source pxlib library (a C library), paradox files can now be handled: both the low-level pxlib API and a TDataset descendent have been implemented and can be installed in the Lazarus IDE. This means that people who have Paradox-based applications can think about moving to the cross-platform Lazarus environment: Even if the paradox files themselves are no longer wanted, the paradox support can still come in handy to move the Paradox data to a newer database using Lazarus/Free Pascal. A good idea in itself, because Paradox is no longer maintained. Worse still, it's file format was never properly documented.

Obviously, the lack of proper documentation also means that there may be Paradox files around that are not handled well by pxlib. At least the Borland FishFacts database (which showcases memos and graphic blobs) can be opened and viewed with the TParadox component presented here; Also on 64-bit systems.

2 Installation

To install the paradox support, a version of the pxlib library must be present on the system. This library is not shipped with Free Pascal or Lazarus, but can be downloaded freely from

```
http://pxlib.sourceforge.net/
```

Unpacking the library should not be a problem, and a simple

```
./configure
make all
make install
```

should configure, build and install the library on any unix system, including mingw for Windows. (Microsoft Visual studio is also reported to work). For 32-bit windows, a pre-built dll is available as well.

The Free Pascal unit is called (logically) pxlib. It is a simple translation of the C header files; The `TDataset` descendent is called `TParadox` and is in a unit `paradox`. They are currently only in subversion and the daily source snapshots of FPC, but a copy has been provided on the CD accompagnying this issue.

A Lazarus package (lazparadox) has been created which registers the TParadox component in the IDE, and puts it in the component palette. There is also a small demo application available.

3   Using the low-level API

The low-level API of the pxlibrary is quite straightforward and simple. The `pxlib` unit does not link the pxlib library statically, so the first thing to do is to call `loadpxlib`:

```
LoadPXLib(pxLibrarName);
```

This will load the PX library. The `pxLibraryName` is a pre-defined constant with the default library name of pxlib for the platform the program is compiled for. When the library is no longer needed, it can be unloaded with `FreePXLib` (this function is called automatically by the finalization section of the pxlib unit).

After loading the library, it must be initialized with the `PX_Boot` call. Similarly, it must be shutdown with the `PX_Shutdown` call prior to unloading it.

A paradox file is represented by the `PPX_Doc` type. It is a pointer to an (opaque) record, and must be allocated with the `PX_New` function. Once the record is reserved in memory in this way, a paradox file can be really opened with the `PX_open_file` call.

Putting all this together, the following minimalistic program will open and close a paradox file:

```
 Var
  Doc : PPX_Doc;

begin
  LoadPXlib(pxlibraryname);
  PX_Boot;
  try
    Doc:=px_new();
    Try
      px_open_file(Doc,ParamStr(1));
      try
         // Do things.
      Finally
        PX_close(Doc);
      end;
    Finally
      PX_Delete(Doc);
    end;
  finally
    PX_Shutdown;
  end;
```

```
end.
```

As can be seen, the file is closed with `PX_Close` and the record is disposed of with `PX_Delete`; Note that `PX_Delete` will not delete the file from disk; It just removes the file's representation from memory.

Once the paradox file was opened, it's internal structure can be examined: the `PX_Get_Num_fields` call returns the number of fields in the file. The `PX_Get_Fields` call will retrieve a pointer to a series of C records describing the fields in the paradox file. Each record has a couple of fields describing a fields in a paradox record:

px_fname the name of the field.

px_flen the declared length of the field.

px_ftype the type of the field.

px_fdc number of decimals in a BCD field.

This means that the structure of a paradox file can be displayed with the following routine:

```
procedure DumpInfo (Doc : PPX_Doc);

Var
  I : Integer;
  S : String;
  pxf : Ppxfield_t;

begin
  I:=1;
  pxf:=PX_get_fields(Doc);
  While I<=PX_get_num_fields(Doc) do
    begin
    Write('Field ',I:3,': ',strpas(pxf^.px_fname):18,' : ');
    S:='';
    Case (pxf^.px_ftype) of
      pxfAlpha:    S:=Format('char(%d)',[pxf^.px_flen]);
      pxfDate:     S:=Format('date(%d)',[pxf^.px_flen]);
      pxfShort:    S:=Format('int(%d)',[pxf^.px_flen]);
      pxfLong:     S:=Format('int(%d)',[pxf^.px_flen]);
      pxfCurrency: S:=Format('currency(%d)',[pxf^.px_flen]);
      pxfNumber:   S:=Format('double(%d)',[pxf^.px_flen]);
      pxfLogical:  S:=Format('boolean(%d)',[pxf^.px_flen]);
      pxfMemoBLOb: S:=Format('memoblob(%d)',[pxf^.px_flen]);
      pxfBLOb:     S:=Format('blob(%d)',[pxf^.px_flen]);
      pxfFmtMemoBLOb:
                 S:=Format('fmtmemoblob(%d)',[pxf^.px_flen]);
      pxfOLE:      S:=Format('ole(%d)',[pxf^.px_flen]);
      pxfGraphic:  S:=Format('graphic(%d)',[pxf^.px_flen]);
      pxfTime:     S:=Format('time(%d)',[pxf^.px_flen]);
      pxfTimestamp: S:=Format('timestamp(%d)',[pxf^.px_flen]);
      pxfAutoInc:  S:=Format('autoinc(%d)',[pxf^.px_flen]);
      pxfBCD:      S:=Format('decimal(%d,%d)',[pxf^.px_flen*2,
                                                 pxf^.px_fdc]);
      pxfBytes:    S:=Format('bytes(%d)',[pxf^.px_flen]);
    else
```

```
          S:=Format('Unknnown type (%d) (%d)',[pxf^.px_ftype,
                                                pxf^.px_flen]);
        end;
      Writeln(S);
      Inc(I);
      Inc(pxf);
      end;
end;
```

Note the pre-defined constants for each of the known field types.

A paradox file is a file with a sequential structure. This means that records in the file are numbered, starting at record number zero. Reading a file is therefore a matter of retrieving the needed record, and extracting the data from a record. The number of records in a file can be retrieved with the `PX_get_num_fields` function. The size of a single record can be retrieved with the `PX_get_recordsize` function: this size can be used to reserve a memory buffer for the contents of a record, which can subsequently be retrieved with the `PX_get_record` function.

To scan over the records in the paradox file, these 3 functions can be used to compose an algorithm like the following:

```
Procedure ScanRecords(Doc : PPX_Doc);

var
  I : Integer;
  buf : Pchar;

begin
  I:=0;
  Buf:=GetMem(PX_get_recordSize(Doc));
  For I:=0 to px_get_num_records(Doc)-1 do
    begin
    PX_get_record(Doc,I, Buf);
    // Do something with buf
    end;
end;
```

The buffer contains the fields in paradox format, one after the other. To access the value, the correct offset in the buffer must be obtained: the offset is simply the sum of the lengths of the preceding fields. Dumping the field contents can therefor be done as follows:

```
Procedure DumpRecordContents(Doc : PPX_Doc; Buf : PChar);

Var
  I,flen : Integer;
  fbuf : PChar;
  pxf : Ppxfield_t;

begin
  pxf:=PX_get_fields(Doc);
  fbuf:=Buf;
  For I:=0 to PX_get_num_fields(Doc)-1 do
    begin
    flen:=pxf^.px_flen;
```

```
      DumpField(Doc,FBuf,pxf);
      Inc(fbuf,Flen);
      Inc(Pxf);
      end;
end;
```

In each iteration, the fbuf pointer is shifted with the lengh of the field that was just dumped.

The contents of each field is placed in the buffer in a specific manner: how this is done depends on the type of the field. pxlib provides functions to retrieve each of the possible field types, these functions are called `PX_get_data_XYZ`, where XYZ is the name of the type to retrieve. Each of these functions is documented, and has it's own semantics. It would lead too far to cover all functions.

The `DumpField` function will therefor be a function that scans the type of the field to dump, and calls the appropriate function to retrieve the field's data from the buffer:

```
Procedure DumpField(Doc : PPX_Doc; FBuf : PChar; pxf : PPxField_t);

Var
  Flen : Integer;
  s : string;
  value : Pchar;
  longv : clong;
  y,m,d : cint;

begin
 flen:=pxf^.px_flen;
 Case (pxf^.px_ftype) of
    pxfAlpha:
      if PX_get_data_alpha(Doc,fbuf,flen,@value)>0 then
        begin
        S:=Strpas(value);
        doc^.free(doc,value);
        end;
    pxfDate:
       if PX_get_data_long(Doc,fbuf,flen,@longv)>0 then
          begin
          PX_SdnToGregorian(longv+1721425,@Y,@M,@D);
          S:=DateToStr(EncodeDate(Y,M,D));
          end;
    pxfAutoInc,
    pxfLong:
      if (PX_get_data_long(Doc,fbuf,flen,@longv)>0) then
        S:=IntToStr(Longv);
  end;
  WriteLn(strpas(pxf^.px_fname):18,' = ',S);
end;
```

For the pxfAlpha (string) type field, the **pxlib** library expects a pointer to the address of a PChar value. On return of the `PX_get_data_alpha`, the PChar will point to a null-terminated buffer of characters. As can be seen, the buffer is converted simply to a string. Since **pxlib** has allocated this buffer, it must be freed with a call to `PPX_Doc.Free`. The date in a paradox buffer is encoded in a special way, and the `PX_SdnToGregorian` function can be used to convert this to a year/month/day triplet. A longint is stored as-is, and the **PX_get_data_long** function will return the value directly in it's last argument.

Note that these functions all accept a pointer to the exact location of the field in the buffer: no checks are done to assure that these locations are correct. It is therefore possible to call e.g. `PX_get_data_alpha` with the location of an integer field, which will yield garbage, of course.

There are more types than shown here, but the complete program can be found on the CD accompagnying this issue: all possible types are treated there.

4 TParadox: A TDataset descendent

All this handling of buffers and memory management is of course tedious and error-prone. That is why a `TDataset` descendent was created, called `TParadox` which presents the contents of the file in a more known and usable format. Besides all the well-known methods from `TDataset`, it offers some additional properties:

LibraryName the name of the pxlibrary to use. Initially this is set to the default value for the platform for which the component is compiled. The library is loaded when the first paradox file is opened.

FileName the name of the paradox file to open.

BlobFileName the name of the file containing the BLOB data for the paradox file. If none is specified, the component tests for the existence of a file with extension .mb (or .MB). If it is found, then it is opened as well and the `BlobFileName` is set to the name of the found file.

TableName after the file is opened, this property is filled with the table name as stored in the paradox file (it need not be the same as the file name).

targetencoding encoding to be used when reading data from the file. If this property is left empty, it will be filled with a default value after opening the file.

inputencoding encoding to be used when writing data to the file. If this property is left empty, it will be filled with a default value after opening the file.

Filter a filter expression (a string). If set, and `Filtered` is set to `True`, only records matching the criterium will be shown.

Filtered If set to `True`, the filter in the `Filter` property is applied.

When reading character data from the file, the data is converted from the encoding used in the file to the encoding specified in the `TargetEncoding` property: this is handled transparantly by the pxlib library.

Use of the component in the lazarus IDE is extremely simple: drop it on a Lazarus form, attach a `TDatasource` to it, and attach several DB-Aware components to it. Set the `FileName` property, and set the Active property to True.

A sample program is included on the CD-ROM accompagnying this issue. It allows to select a filename using a `TFileNameEdit`, and shows the contents of the file in a DBGrid. If the file is determined to be the 'biolife.db' file that comes with the BDE sample data, then the DBMemo and TImage on the form are hooked up to the description of the fish, and the bitmap of the fish:

```
procedure TMainForm.Button1Click(Sender: TObject);
begin
  FPX.FileName:=FEPX.FileName;
```

Figure 1: The FishFacts demo file opened in a Lazarus program
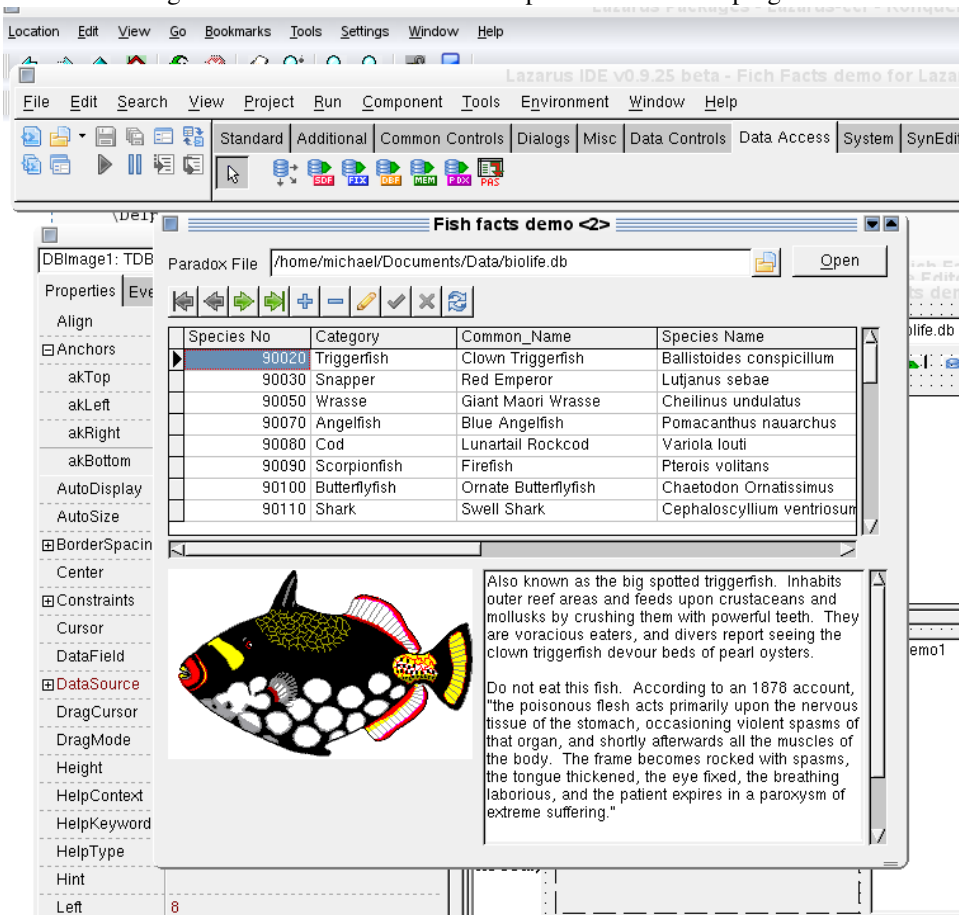

```
  FPX.Open;
  If FPX.FindField('Notes')<>Nil then
    DBMemo1.DataField:='Notes';
  If FPX.FindField('Graphic')<>Nil then
    DBImage1.DataField:='Graphic';
end;
```

The result can be seen in figure 1 on page 7, showing that blob and graphic data are supported for Paradox files.

5   Conclusion

While it is unlikely that anyone will start new development using Paradox files as a database back-end, supporting Paradox tables still has it's uses: simply viewing or converting legacy data in Paradox format is now possible using the Free Pascal and Lazarus toolchain, with no more effort than is needed to view data in any other format: the ever-increasing number of supported databases using a unified TDataset approach is just one of the many strong points of the Lazarus/Free Pascal tandem.