Table Source Generator Expert

by Bob Swart

In this article we'll develop a variant of the nice and visual database (or table) form expert. Instead of a visual form, this expert generates a unit which creates an empty database table, using the structure of an existing table.

Visual

Delphi is a visual RAD tool, which means that Delphi users probably do everything visually. From designing forms and resources to designing tables with the Borland Database Desktop or the Database Explorer. However, amidst this visual violence, we often forget that a form, designed with the visual Database Form Expert, needs the actual table, created with the visual Database Desktop, to be present when the `TTable` component is set to `Active`. If the table is not found, a BDE exception will be raised. Which means that most applications need to ship with the required tables and index files. This is of course all right in those cases we're shipping an application that needs data to be present in the tables, but it's actually less useful to ship a phonebook application with nothing but empty databases, ready to be filled by the end-user, but present nonetheless. To avoid having to include empty tables, one could of course hand-write code to create the table on the fly when needed, using the `CreateTable` method of the `TTable` component. Unfortunately, it takes quite some code to re-create a table. And if your application uses lots of tables, you need to write code to create each of them. And why should we? We've already created them once visually, so why not use that information to re-create them when needed?

To solve this problem once and for all (once I got sick and tired of writing the same kind of table creating code again), I've written a little Table Source Generator Expert, that given a table, extracts the structural information from the table (fields names and types and indexes) and generates code to re-create the same, empty, table and index files. The expert part should be no surprise for readers of my *Under Construction* column, and looks like Listing 1.

```
Type
  TTableSrcExpert = class(TIExpert)
  public
    { Expert Style }
    function GetStyle: TExpertStyle; override;
    { Expert Strings }
    function GetIDString: String; override;
    function GetName: String; override;
  {$IFDEF WIN32}
    function GetAuthor: String; override;
  {$ENDIF}
    function GetMenuText: String; override;
    function GetState: TExpertState; override;
    { Launch the Expert }
    procedure Execute; override;
  end {TDataSrcExpert};
```

➤ *Listing 1*

```
procedure TTableSrcExpert.Execute;
var f: System.Text;
    i: Integer;
begin
  try
    with TTable.Create(nil) do
    try
      with TOpenDialog.Create(nil) do
      try
        Title := GetName; { name of Expert as OpenDialog caption }
        Filter := 'DB Files (*.db)|*.db';
        Options := Options + [ofShowHelp, ofPathMustExist, ofFileMustExist];
        if Execute then begin { not a showmodal! }
          DatabaseName := ExtractFilePath(FileName);
          TableName := ExtractFileName(FileName)
        end
      finally
        Free
      end;
```

➤ *Listing 2*

Most methods are straightforward (and full source code is on the disk, as usual), so we skip right to the procedure `Execute` where the name of the table is asked, followed by the extraction of the table, fields and index structure information and the generation of ObjectPascal source code. Let's look at the source code for the `Execute` procedure one step at a time.

The first part of the `Execute` procedure (Listing 2) is using an `TOpenDialog` to ask for the specific tablename to convert to source code. Note that we're looking for Paradox .DB files by default. The filepath is used as `DatabaseName`, while the filename is used as `TableName`. We're not interested in Aliases here. A dynamic `TTable` is created with the `DatabaseName` and `TableName`, and we're ready to get the information from it we want.

The second part (Listing 3) starts the code generation with the first few lines of the unit. The unitname is the same as the tablename, and the unit filename is the same as the table filename with the .PAS extension instead of .DB. Note again that we're expecting Paradox table formats again (but you are free to modify this to support other formats as well). We haven't used any information from the `Table` yet, by the way.

The third part (Listing 4) gets interesting. Here we start using information from the Table to define the fields. We can either Open the Table, or call `FieldDefs.Update` to obtain the field structural information. We've

```
{generate the first part of the unit source}
  System.Assign(f,ChangeFileExt(TableName,'.PAS'));
  System.Rewrite(f);
  writeln(f,'unit ',ChangeFileExt(TableName,''),';');
  writeln(f,'interface');
  writeln(f);
  writeln(f,'  procedure Create',ChangeFileExt(TableName,''),';');
  writeln(f);
  writeln(f,'implementation');
  writeln(f,'uses DB, DBTables;');
  writeln(f);
  writeln(f,'  procedure Create',ChangeFileExt(TableName,''),';');
  writeln(f,'  begin');
  writeln(f,'    with TTable.Create(nil) do');
  writeln(f,'    try');
  writeln(f,'      Active := False;');
  writeln(f,'      TableType := ttParadox;');
  writeln(f,'      TableName := ''',TableName,''';');
```

➤ *Listing 3*

```
FieldDefs.Update { get info without opening the database };
writeln(f,'      with FieldDefs do');
writeln(f,'      begin');
writeln(f,'        Clear;');
for i:=0 to Pred(FieldDefs.Count) do begin
  writeln(f,' ':8,'Add(''',FieldDefs[i].Name,''', ',
 {$IFDEF Win32}
  GetEnumName(TypeInfo(TFieldType),  Ord(FieldDefs[i].DataType)),
 {$ELSE}
  GetEnumName(TypeInfo(TFieldType),  Ord(FieldDefs[i].DataType))^,
 {$ENDIF}
    ', ',FieldDefs[i].Size,', ', FieldDefs[i].Required,');')
end;
writeln(f,'      end;');
```

➤ *Listing 4*

```
{$IFDEF Win32}
 GetEnumName(TypeInfo(TFieldType), Ord(FieldDefs[i].DataType))
{$ELSE}
 GetEnumName(TypeInfo(TFieldType), Ord(FieldDefs[i].DataType))^
{$ENDIF}
```

➤ *Listing 5*

done the latter here, so we don't have to actually open the table (saves time and space). `FieldDefs.Count` identifies the number of fields in the Table (counting from 0 upwards), and `FieldDefs` can be used as array property of type `TFieldDefs` when it comes to the values of `FieldDefs[].Name`, `FieldDefs[].DataType`, `FieldDefs[].Size` and `FieldDefs[].Required`. The `Name` part is the easy one. To get the `Field Type`, we need to get to the `DataType`, which is of type `TFieldType`, and use RTTI to obtain the actual string representation of the actual value. Since `RTTI` changed representation from Delphi 1 to 2, we need an `IFDEF` here to distinguish between the two implementations.

The only difference between the 16- and 32-bits version is that the `GetEnumName` returns a pointer to a string in the 16-bits version, and a plain string in the 32-bits version. So we need an extra ^ dereference symbol. That's all.

Now that we have all the fields, names, size (only needed for memo and blobs) and required information, it's time to look for any indexes that are defined with this table. To do that, we can use a similar approach, by calling `IndexDefs.Update`, (Listing 6).

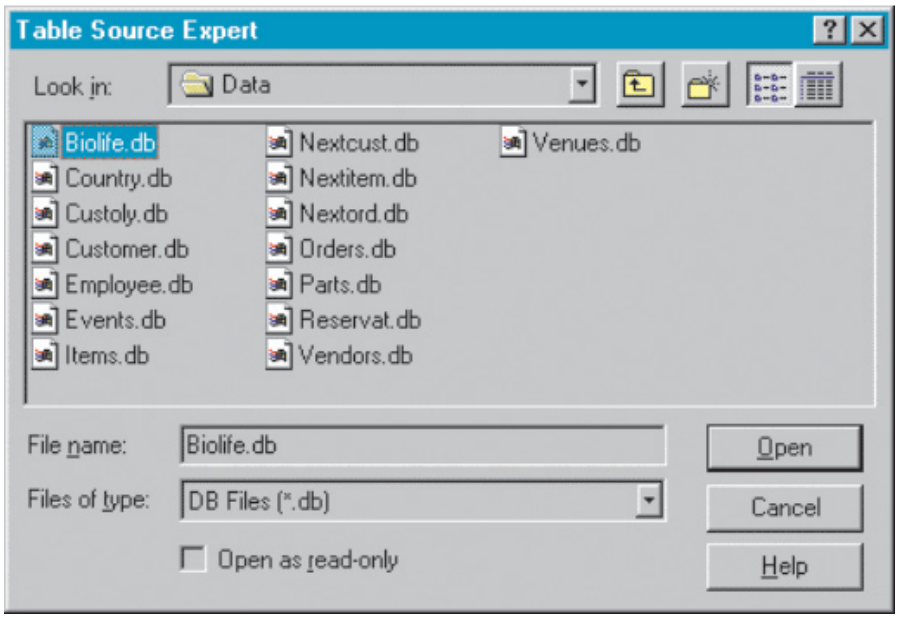
The `IndexDefs` property has a `Name`, list of `Fields` but also a set of `options`. To print a string representing a set of options cannot be done using RTTI, since more than one actual option can be selected. A separate function `OptionNames` had to be written to look for the presence of each option, and appending it to the `Result String` as shown in Listing 7.

And finally, after the fields and index information has been re-created, it's time to write the call `CreateTable`, and finish the rest of the unit (Listing 8).

Now that we have an actual `TTableSrcExpert`, we can install it in the Delphi IDE by installing the unit `TABLESRC` as if it were a new component. However, there's also another way to utilise an expert: by simply creating it and calling the `Execute` method ourselves. This results in a very tiny program of only 9 lines of code (recompile using the command-line compilers) that results in a handy program to convert .DB and .DBF tables into units that can re-create those tables, (Listing 9).

If we compile the program above and run it (in Windows 95), we get a `FileOpen Dialog` that asks for a database filename. Let's select the BIOLIFE.DB and see what happens (Figure 1).

➤ *Figure 1*



The resulting unit BIOLIFE can be used to re-create an empty table called BIOLIFE but with exactly the same structure (see Listing 10).

To test the just generated unit BIOLIFE, all we need to do is write another very tiny test program (this time a Win32-only console application), and call the CreateBIOLIFE function. Be sure not to run this program in the same directory where the current BIOLIFE table resides, because it will overwrite the yable without questioning! Normally, you'd want to check for the existence of the table (using FileExists) first before attempting to re-create it, of course.

```
{$APPTYPE CONSOLE}
uses
  BIOLIFE;
begin
  CreateBIOLIFE;
end.
```

While it may seem trivial (or at least less useful) to be able to re-create the BIOLIFE table, consider what it may do to your applications. Just image not having to write table-creation code any more. From now on, we only need to visually define our tables, and then run some experts to create visual forms and non-visual table re-creation code. The (empty) tables no longer need to ship alongside your application. That means fewer files to install, less disk space. Less development time. At least for some of my projects I have managed to save myself enough time to justify the work invested in this expert already!

Full source code for the Table Source Generator Expert and application is available on this month's disk. If you have any questions or comments, just ask. Also, as an extra bonus, I've included a copy of my updated TableBob Wizard, which can re-generate a table (in Paradox format) and optionally copy one or more fields from all the records in the source table (Access, Paradox, dBASE, etc.) to the newly generated Paradox table. It can also convert a table to HTML webpages, so this should save you even more time!

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian (at www.bolesian.com), a freelance technical author for *The Delphi Magazine*, co-author of *The Revolutionary Guide to Delphi 2* and the electronic knowledge base *Delphi Internet Solutions*.

➤ Listing 6

```
IndexDefs.Update { get info without opening the database };
writeln(f,'      with IndexDefs do');
writeln(f,'      begin');
writeln(f,'        Clear;');
for i:=0 to Pred(IndexDefs.Count) do begin
  writeln(f,' ':8,'Add(''',IndexDefs[i].Name,''', ''', IndexDefs[i].Fields,
    ''', ', OptionNames(IndexDefs[i].Options),');')
end;
writeln(f,'      end;');
```

➤ Listing 9

```
uses TableSrc;
begin
  with TTableSrcExpert.Create do
  try
    Execute
  finally
    Free
  end
end.
```

➤ Listing 7

```
function OptionNames(IndexOptions: TIndexOptions): String;
begin
  Result := '[ ';
  if ixPrimary in IndexOptions then
    Result := Result + 'ixPrimary,';
  if ixUnique in IndexOptions then
    Result := Result + 'ixUnique,';
  if ixDescending in IndexOptions then
    Result := Result + 'ixDescending,';
  if ixCaseInsensitive in IndexOptions then
    Result := Result + 'ixCaseInsensitive,';
  Delete(Result,Length(Result),1);
  Result := Result + ']'
end {OptionNames};
```

➤ Listing 8

```
        writeln(f,'      CreateTable');
        writeln(f,'    finally');
        writeln(f,'      Free');
        writeln(f,'    end');
        writeln(f,'  end {Create',
          ChangeFileExt(TableName,''),'};');
        writeln(f);
        writeln(f,'end.');
        System.Close(f)
      finally
        Free
      end
    except
      HandleException
    end
  end {Execute};
end.
```

➤ Listing 10

```
unit BIOLIFE;
interface
procedure CreateBIOLIFE;
implementation
uses
  DB, DBTables;
procedure CreateBIOLIFE;
begin
  with TTable.Create(nil) do
  try
    Active := False;
    TableType := ttParadox;
    TableName := 'BIOLIFE.DB';
    with FieldDefs do begin
      Clear;
      Add('Species No', ftFloat, 0, FALSE);
      Add('Category', ftString, 15, FALSE);
      Add('Common_Name', ftString, 30, FALSE);
      Add('Species Name', ftString, 40, FALSE);
      Add('Length (cm)', ftFloat, 0, FALSE);
      Add('Length_In', ftFloat, 0, FALSE);
      Add('Notes', ftMemo, 50, FALSE);
      Add('Graphic', ftGraphic, 0, FALSE);
    end;
    with IndexDefs do begin
      Clear;
      Add('', 'Species No', [ ixPrimary,ixUnique]);
    end;
    CreateTable
  finally
    Free
  end
end {CreateBIOLIFE};
end.
```