

# Analysing DataSets (in Delphi and Kylix)

by Bob Swart

One of the good things of being part of a Delphi Solutions Centre (we call it DOC), is that people will call you for help whenever they need you. Sometimes, this means taking over some existing half-finished application with no documentation, but which has been in use for years already, so *“please don’t break it, but while you’re at it, could you make it internet aware as well?”*

And when I look at the pile of source code and try to make sense of it, the first place where I usually start is at the base - the database, that is. If you’re lucky, there’s a datamodel documented somewhere (if you’re really lucky it’s not out of date), but if you’re unlucky, the original project has been written by someone who had never heard of the concept of a data module (or the fact that you shouldn’t put your tables on all your forms, make them all autogenerated, and wonder why updates sometimes fail).

Although the Database Explorer and certainly SQL Explorer can show a lot of detailed information of all these tables, I often want to have a document (on paper) that contains an up-to-date overview of the different tables, their fields, fieldtypes, etc. Not only for situations like I described above, but also for my own projects (if only to check if the final implementation corresponds to the initial specification).

For those purposes, I’ve written a little tool that can analyse an entire BDE Alias. I’ve been using this tool for a while now, and will show it in the first part of this article. However, as I’ve just received Kylix, I expect slowly to migrate most of my BDE related tools and projects to dbExpress (using Kylix and the upcoming Delphi 6). And the second part of this article will show what I needed to do in order to make the BDE Alias analyser work with dbExpress (something that those of you who attend my *From Delphi to Kylix and Back* pre-conference tutorial at DCon will also see in detail).

## BDE Alias Analyser

But first of all, let’s consider my BDE Alias analyser. The purpose of this little tool (which can be seen in the listing opposite), is to walk through all tables in a given alias, and produce some detailed information for each of these tables. The helpful method that gives us all tablenamees for a given alias can be obtained by calling the GetTableNames method from an active Session component (very BDE specific, indeed), as follows:

Once we have a list of TableNames, we need to use those to open specific TTable components (using both the given AliasName and TableName) and list relevant information for fields, such as Name, Type and whether or not a value is required. We can also use the IndexDefs to list the known indices (like the primary index).

```
{ $APPTYPE CONSOLE }
program analias;
uses
  Classes, DB, DBTables;
var
```

```
    i, j: Integer;
    TableNames: TStringList;
begin
    TableNames := TStringList.Create;
    with TSession.Create(nil) do
    try
        AutoSessionName := True;
        GetTableNames(ParamStr(1), '', True, False,
                    TableNames);

    finally
        Free
    end;
    with TTable.Create(nil) do
    try
        DatabaseName := ParamStr(1);
        for j:=0 to Pred(TableNames.Count) do
        begin
            writeln;
            TableName := TableNames[j];
            Open;
            writeln(TableName[j], ' ', RecordCount:4);
            for i:=1 to Length(TableName[j])+5 do
                write(' ');

            writeln;
            writeln('Field'#9'Name'#9'Type'#9'Req');
            for i:=0 to Pred(FieldCount) do
            begin
                write(i,#9);
                write(Fields[i].DisplayName,#9);
                write(FieldDefs[i].FieldClass.ClassName);
                if FieldDefs[i].DataType = ftString
then
                    write([' ',FieldDefs[i].Size,']');
                    if FieldDefs[i].Required then
                        write('#9'Yes');

                writeln
            end;
            IndexDefs.Update;
            if IndexDefs.Count > 0 then
            begin
                writeln;
                writeln('Index'#9'Name'#9'Fields')
            end;
            for i:=0 to Pred(IndexDefs.Count) do
            begin
                write(i,#9);
                write(IndexDefs[i].DisplayName,#9);
                write(IndexDefs[i].Fields);
                writeln
            end;
            writeln;
            Close
        end
    finally
        Free
    end
end.
end.
```

Note that the code from this listing uses tabs as delimiter between the different values (or columns). This is done so I can import the resulting file inside Windows, and convert the text to a Winword table, which results in a nice and readable report. As an example, the table BIOLIFE.DB from Alias DBDEMOS is analysed and reported as follows (note that I’ve converted this text to a table already):

```

biolife.db 28
=====
Field Name   Type   Req
0   Species No   TFloatField
1   Category     TStringField[15]
2   Common_Name  TStringField[30]
3   Species Name TStringField[40]
4   Length (cm)  TFloatField
5   Length_In    TFloatField
6   Notes        TMemoField
7   Graphic      TGraphicField

Index Name   Fields
0   <Primary>   Species No

```

By the way, you can extend the code from the listing with additional features that can report even more information, like range (min value, max value), etc. but the information printed above is usually enough for my first analysis anyway.

## dbExpress Analyser

As you probably know, Kylix is shipping (I recently received my copy of Kylix Server Developer edition) and - what you may or may not know - Kylix is not using the BDE, but uses something called dbExpress for data access. And so will Delphi 6, so most of what I'm saying in the remainder of this article will probably also work with the forthcoming Delphi 6 (at least, that's what I've been told).

Anyway, to move the code in listing 1 from Windows to Linux, I usually FTP it to my Linux machine, and then open the project in Kylix. I won't bother you here with details regarding .res, .opt, .cfg and .dfm files, we only have to worry about a single analias.dpr file here, which can be moved over and loaded in Kylix without too much trouble.

Of course, loading in Kylix and recompiling with Kylix are two different matters. And since the original application is very BDE specific (TSession and TTable), we need to look at dbExpress alternatives instead (like SQLConnection and SQLTable). But first, we need to replace the DBTables unit with SqlExpr. Now, we can replace the TSession with TSQLConnection. The AutoSessionName property makes no sense here, but we can set the ConnectionName to initialise the TSQLConnection component (and can make it active by setting Connected to true).

Once we have a list of TableNames, we can again loop through it, and create TSQLTable components. The only difference with a BDE TTable is that we no longer need to mention a DatabaseName, but an SQLConnection instead (which must point to an existing SQLConnection instance, just as we should have used a TDatabase component in the BDE example, which would have been much better).

```

{$APPTYPE CONSOLE}
program analias;
uses
  Classes, DB, SqlExpr;
var
  i, j: Integer;
  TableNames: TStringList;
  SQLConnection1: TSQLConnection;
begin
  TableNames := TStringList.Create;
  SQLConnection1 := TSQLConnection.Create(nil);
  with SQLConnection1 do
  begin
    LoadParamsOnConnect := True;

```

```

    ConnectionName := ParamStr(1);
    Connected := True;
    GetTableNames(TableNames);
  end;
  with TSQLTable.Create(nil) do
  try
    SQLConnection := SQLConnection1;
    for j:=0 to Pred(TableNames.Count) do
    begin
      writeln;
      TableName := TableNames[j];
      Open;
      writeln(TableName[j], ' ', RecordCount:4);
      for i:=1 to Length(TableNames[j])+5 do
        write('=');
      writeln;
      writeln('Field'#9'Name'#9'Type'#9'Req');
      for i:=0 to Pred(FieldCount) do
      begin
        write(i,#9);
        write(Fields[i].DisplayName,#9);
        write(FieldDefs[i].FieldClass.ClassName);
        if FieldDefs[i].DataType = ftString
      then
        write([' ',FieldDefs[i].Size,']');
        if FieldDefs[i].Required then
          write('#9'Yes');
        writeln
      end;
      IndexDefs.Update;
      if IndexDefs.Count > 0 then
      begin
        writeln;
        writeln('Index'#9'Name'#9'Fields')
      end;
      for i:=0 to Pred(IndexDefs.Count) do
      begin
        write(i,#9);
        write(IndexDefs[i].DisplayName,#9);
        write(IndexDefs[i].Fields);
        writeln
      end;
      writeln;
      Close
    end
  finally
    Free;
    SQLConnection1.Free
  end
end.

```

Note that LoadParamsOnConnect which we must set to True, and is needed to make sure that the SQLConnection component will load the DriverName and Params automatically when you set (or change) the value of ConnectionName. In this particular case, ConnectionName should be defined in your file "dbxdrivers" (as described in the Kylix Developer's Guide page 19-2).

As you can see, it took just over 10 lines of code to change one unit, two components and a few properties, and now this application compiles with Kylix. And is expected to work with Delphi 6 as well in the future. For more Delphi to Kylix (and back) migration issues, please attend the pre-conference tutorial at DCon 2001, or visit my website on a regular basis, as I plan to move some more of my tools over to Linux. *Stay tuned...*



**Bob Swart** (aka *Dr. Bob* - [www.drbob42.com](http://www.drbob42.com)) is an @-Consultant for Everest Delphi OplossingsCentrum and has spoken at conferences all over the world. He's also a free-lance technical author and has written chapters for several Delphi and C++Builder books.