Creating TTables And TFields At Runtime

by Bob Swart

We all know how to drop a TTable component onto a form, connect it to a table and open it to get access to the data. But what if we don't have or don't want a form? What if there is nothing to drop a table onto, not even a Data Module or Web Module? In that case we need to create the table dynamically, and the fields of the table, including the calculated and lookup fields, which is what this article is all about.

Creating Tables On Disk

First, of course we all know TableBob, my Table-2-HTML or Table-2-Source Wizard (if not, check out Issue 31 for the article and surf to www.drbob42.com/tools/tablebob.htm for the latest version). TableBob turns the BIOLIFE.DB table into source code to regenerate the table. Listing 1 shows the source code produced by TableBob for BIOLIFE.DB.

We see that we can use a call to CreateTable to create an entire table dynamically, even including indexes. Personally, I find this ability to create physical tables very helpful when deploying applications, so I don't need to ship empty tables (or worse, tables still filled with your test data). I've also used these techniques to build a self-maintaining internet guestbook.

In theory, it's even possible to create referential integrity relations, but you need some low-level BDE calls that I could never entirely get to work (email me at bob@bolesian.nl if you can show me how, and I'll include the information in an updated version of this article, with the appropriate credits, of course).

Obtaining Field Information

Once we have a table on disk, we can again create a TTable component, point it to the table on disk, and read information from the table (or write new data to it). For a given, supposedly unknown, table, we can even use the information from the table itself to list the number of fields, their type, etc (Listing 2).

Note that just like in the first listing, we're using the FieldDefs array property of the TTable, and we didn't even need to open the table to get our hands on the field definitions, which are shown in Listing 3 for the BIOLIFE.DB table.

Of course we need to open the table if we actually want to read some information from it, or edit or insert/append data. After we've opened the table, we can use the Fields array property to get to the fields' DisplayNames, EditMask (if any) and, most important probably, the DisplayText with the value of the actual field in the current record. Note that all this information was not available when the table was still closed, as FieldDefs only contains field definition information, not the actual table contents and field 'action' information we can find in the Fields property (Listing 4).

If we run this program on -BIOLIFE.DB, we get the output in Listing 5. Note that the FieldNames are empty; we must get to the DisplayName to get the name of the field. Also note that the DisplayText prints (MEMO) for memo fields, and (GRAPHICS) for graphic fields. This is exactly the way they appear in a TDBGrid or Delphi's Web Modules. To get the

```
program BIOLIFE;
uses DB, DBTables;
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.
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
program Analyse1;
{$APPTYPE CONSOLE}
uses DB, DBTables;
var  i: Integer;
begin
  with TTable.Create(nil) do
  try
    DatabaseName := 'DBDEMOS';
    TableName := 'BIOLIFE.DB';
    FieldDefs.Update; { get FieldDefs without Opening table itself }
    writeln;
    for i:=0 to Pred(FieldDefs.Count) do begin
      write('Field ',i,': ',FieldDefs[i].Name);
      write(' - ',FieldDefs[i].FieldClass.ClassName);
      if FieldDefs[i].DataType = ftString then
        write('[',FieldDefs[i].Size,']');
      writeln
    end
  finally
    Free
  end
end.
```

true contents of these fields, we need to use the Value property or the explicit AsString property, which would yield the following enhanced version of the program so far:

```
if Fields[i].DataType = ftMemo then
  write(' - ',Fields[i].Value)
else
  write(' - ',Fields[i].DisplayText);
```

And this time, we indeed get the full contents of the memo field inside the table.

Creating Field Components

Now that we can obtain field type and value information, it's time to put a little bit more structure to it. For the given BIOLIFE.DB example table, we should know by now the types of each of the eight fields. So, why not simply declare those eight specific fields and assign them to the table at runtime? This would be equivalent to a right-click on the Fields Editor and Add all fields by the way.

The source snippet in Listing 6 will create the first TField component, a TFloatField to be specific, pointing to field Species No in the Table. Unfortunately, an unexpected BDE exception will be raised and reported on standard output:

```
Exception EDatabaseError in
  module ANALYSE4.EXE at 000315A2.
   Field name missing.
```

The only way to get rid of this exception is to make sure the FieldName property is assigned before the DataSet property gets assigned, as follows:

```
SpeciesNo :=
  TFloatField.Create(Table);
SpeciesNo.FieldName := 'Species No';
SpeciesNo.DataSet := Table;
```

After we change the order of the DataSet and FieldName assignments, we get the result we want for this first field of the BIOLIFE table: Species No: 90020. And this is only the start, of course. Once we have a TxxxField component, we have easy access to every property and method of that particular field component (see the help for a complete list).

```
Field 0: Species No - TFloatField
Field 1: Category - TStringField[15]
Field 2: Common_Name - TStringField[30]
Field 3: Species Name - TStringField[40]
Field 4: Length (cm) - TFloatField
Field 5: Length_In - TFloatField
Field 6: Notes - TMemoField
Field 7: Graphic - TGraphicField
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
program Analyse2;
{$APPTYPE CONSOLE}
uses DBTables;
var  i: Integer;
begin
  with TTable.Create(nil) do
  try
    DatabaseName := 'DBDEMOS';
    TableName := 'BIOLIFE.DB';
    Open; { Open table to get actual Fields information }
    writeln;
    for i:=0 to Pred(FieldCount) do begin
      write('Field ',i,': (',Fields[i].Name,')');
      write(' displays "',Fields[i].DisplayName,'"');
      write(' - ',Fields[i].DisplayText);
      writeln
    end;
    Close
  finally
    Free
  end
end.
```

```
Field 0: () displays "Species No" - 90020
Field 1: () displays "Category" - Triggerfish
Field 2: () displays "Common_Name" - Clown Triggerfish
Field 3: () displays "Species Name" - Ballistoides conspicillum
Field 4: () displays "Length (cm)" - 50
Field 5: () displays "Length_In" - 19.6850393700787
Field 6: () displays "Notes" - (MEMO)
Field 7: () displays "Graphic" - (GRAPHIC)
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
program Analyse4;
{$APPTYPE CONSOLE}
uses DB, DBTables;
var
  Table: TTable;
  SpeciesNo: TFloatField;
begin
  Table := TTable.Create(nil);
  try
    Table.DatabaseName := 'DBDEMOS';
    Table.TableName := 'BIOLIFE.DB';
    SpeciesNo := TFloatField.Create(Table);
    SpeciesNo.DataSet := Table;
    SpeciesNo.FieldName := 'Species No';
    Table.Open;
    writeln(SpeciesNo.DisplayName,': ',SpeciesNo.DisplayText);
    Table.Close
  finally
    SpeciesNo.Free;
    Table.Free
  end
end.
```

Dynamic Calculated Fields

One of the great benefits of the Delphi IDE when working with TTable components, is the Fields Editor. The best place to add all fields you want to make visible to your application, add new fields, define lookup or calculated fields. You can even use the Fields Editor to drag and drop fields on your Form. But right now, I'm more interested in the ability to create lookup or calculated fields.

To dynamically create a Calculated Field, we start just like any other dynamic field, by creating an instance of a TField component (or a specialised type of field, such as a TStringField or TIntegerField). For example, let's create a calculated field The Answer of type TFloatField, for the BIOLIFE.DB table.

As you'll see in the source code (Listing 7), we do not only need to create a field of 'kind' fkCalculate, we also need to create an OnCalcFields event of type

```
procedure(DataSet: TDataSet)
  of object;
```

meaning that it can't be a regular procedure, but it must be a class method. So, in our small example, we need to create a special class TBTable, which I derived from TTable (we need a TTable anyway), that contains the CalcFields method that gets assigned to the OnCalcFields event handler of the dynamic table. Using a TTable derived class instead of a TDummy class to host the CalcFields method means we won't introduce a class we don't really need.

Other than that, it's hardly different from a regular fkData field. We must assign the FieldName and FieldKind before we can assign the DataSet (or we get the BDE exception we saw earlier). And of course, we get the expected result:

```
The Answer: 42
```

```
program Analyse5;
{$APPTYPE CONSOLE}
uses  DB, DBTables;
const CalcFieldName = 'The Answer';
type
  TBTable = class(TTable)
    procedure CalcFields(DataSet: TDataSet);
  end;
  procedure TBTable.CalcFields(DataSet: TDataSet);
  begin
    DataSet[CalcFieldName] := 42 { or some real calculation, of course }
  end;
var
  Table: TTable;
  CalcField: TFloatField;
begin
  Table := TTable.Create(nil);
  try
    Table.DatabaseName := 'DBDEMOS';
    Table.TableName := 'BIOLIFE.DB';
    CalcField := TFloatField.Create(Table);
    CalcField.FieldName := CalcFieldName;
    CalcField.FieldKind := fkCalculated; { default - fkData }
    CalcField.DataSet := Table;
    Table.OnCalcFields := Table.CalcFields;
    Table.Open;
    writeln(CalcField.DisplayName,': ',CalcField.DisplayText);
    Table.Close
  finally
    CalcField.Free;
    Table.Free
  end
end.
```

➤ *Listing 7*

Dynamic Lookup Fields

For lookup fields, it gets a bit more complicated, as we need another DataSet to look the data up in, as well as another Field component, as will become clear shortly. Also, four additional properties of the 'lookup' TField component must be set in order to function as a lookup field, namely: KeyFields, LookUpDataset, LookUpKeyFields and LookUpResultField. As an example, let's look at the source code (see Listing 8) to create a lookup field at runtime using the two DBDEMOS tables CUSTOMER and ORDERS.

Rather than going through the trial and error I had to face, let's just concentrate on the fact that the KeyField of the LookupField must exist as a separate field component as well. Otherwise, a BDE exception *field CustNo not found* will be raised, which is quite odd, since field CustNo is both part of the CUSTOMER table and the ORDERS table (so it took a while before I found a way to get rid of this exception).

Apart from that little unexpected problem (and I wonder why we can't get on without it), we need to define a KeyField to connect to a LookupKeyField from a LookupDataSet, and once they connect, we can return a LookupResultField, which will indeed be the actual (lookup) value for our lookup FieldName.

Conclusion

In this article, we've seen how to create dynamic table and dynamic fields, including calculated fields and lookup fields. We encountered a number of peculiar exceptions, and learned a few new tricks along the way (at least I did).

The techniques discussed in this article will of course be particularly helpful when writing non-visual applications or DLLs for the internet that don't use any Data Modules or Web Modules.

Bob Swart (aka Dr.Bob, visit www.drbob42.com) is a professional knowledge engineer technical consultant using Delphi, C++Builder and JBuilder for Bolesian. In his spare time, Bob likes to watch videos of Star Trek Voyager and Deep Space Nine with his 4.5-year-old son Erik Mark Pascal and his 2-year-old daughter Natasha Louise Delphine.

➤ *Listing 8*

```
program Analyse6;
{$APPTYPE CONSOLE}
uses DB, DBTables;
var
  Customer,Orders: TTable;
  LookupField: TStringField;
  CustNo: TFloatField;
begin
  Customer := TTable.Create(nil);
  Orders := TTable.Create(nil);
  try
    Customer.DatabaseName := 'DBDEMOS';
    Customer.TableName := 'CUSTOMER.DB';
    Customer.Open;
    Orders.DatabaseName := 'DBDEMOS';
    Orders.TableName := 'ORDERS.DB';
    CustNo := TFloatField.Create(Orders);
    CustNo.FieldName := 'CustNo';
    CustNo.DataSet := Orders;
    LookupField := TStringField.Create(Orders);
    LookupField.FieldName := 'Customer Company';
    LookupField.FieldKind := fkLookup; { default - fkData }
    LookupField.DataSet := Orders;
    LookupField.KeyFields := 'CustNo';
    LookupField.LookupDataSet := Customer;
    LookupField.LookupKeyFields := 'CustNo';
    LookupField.LookupResultField := 'Company';
    Orders.Open;
    writeln(LookupField.DisplayName,':
      (',CustNo.DisplayText,') ', LookupField.DisplayText);
    Orders.Close;
    Customer.Close
  finally
    LookupField.Free;
    CustNo.Free;
    Orders.Free;
    Customer.Free
  end
end.
```