

Keeping Hold of Your Things

by
Joanna Carter

Part 2

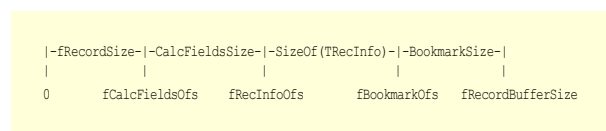
Last time we looked at creating a class for managing a schema file that contains the field definitions for the table that we want to connect to. As promised, this time, I am going to look at the rather complex process of deriving a class from TDataset.

As I looked at the source code for TDataset, I realised that what Delphi is actually doing is to provide one side of a 'conversation'. If we wish to create a derivation of TDataset, then we are required to fill in the long silences that are left for us. Having said that, there are times in this conversation, where Delphi assumes that we are going to say nothing and can quite happily carry on talking all on its own.

Before we can write the TDataset definition, we had better find out what this conversation is going to be based on. The crux of a TDataset is the passing of data between a binary file on disk and the TField components that are connected to the TDataset. This is accomplished through the management of internal data buffers, one buffer for every row of data that is required to be visible at any one time, plus a couple of extras. This is calculated by a TDataSource querying all of the data-aware components that it is attached to and taking the highest number of rows from a component such as a DBGrid.

I would recommend that you use a PChar to hold these buffers, as the size of the buffer is not known until the table is initialised and it will also allow you to do pointer maths to choose which part of the buffer you wish to read or write.

There is no absolute definition of how a record buffer should be laid out but I would recommend the following:



On the top row of this diagram, CalcFieldsSize and BookmarkSize are both properties that Delphi uses to help calculate the buffer size, CalcFieldsSize is set by Delphi without our interference but BookmarkSize has to be set by us. TRecInfo is a record type that can hold

any extra information that may be needed for each record, such as deleted or other flags, but must at least hold a TBookmarkFlag record type. The choice of TRecInfo as a type name is arbitrary but conventional. The fRecordSize is determined by us after we have worked out how many fields and of what type make up the record.

The second row of the diagram shows a series of values that are kept for ease of manipulating the buffer and are fairly self explanatory.

Finding Where You Left Your Things

```
PRecInfo = ^TRecInfo;
TRecInfo = record
    BookmarkFlag: TBookmarkFlag;
end;
```

As you can see, the contents of the TRecInfo that we are going to define are basic, but don't forget that this is where you should keep any other record specific information such as deleted or null flags. TBookmarkFlag is defined as an enumeration of (bfCurrent, bfBOF, bfEOF, bfInserted). You do not need to, nay must not, do anything with this flag, Delphi makes use of it and expects you to merely point the way to it by overriding the abstract GetBookmarkFlag method. A quick look at typical code for this will explain how you tell Delphi to find the BookmarkFlag that it requires.

```
function TBinaryTable.GetBookmarkFlag(Buffer:
    PChar): TBookmarkFlag;
begin
    Result := PRecInfo(Buffer +
        fRecInfoOfs)^.BookmarkFlag;
end;
```

Having shown a use for fRecInfoOfs, it may be a good idea to mention a method that I have added, simply to set up those offset pointers in the buffer structure diagram:

```

procedure TBinaryTable.InitBufferPointers;
begin
  fCalcFieldsOfs := fRecordSize;
  fRecInfoOfs := fCalcFieldsOfs +
    CalcFieldsSize;
  fBookmarkOfs := fRecInfoOfs +
    SizeOf(TRecInfo);
  fRecordBufferSize := fBookmarkOfs +
    BookmarkSize;
end;

```

There is another item of record level data that Delphi expects to find in the buffer, and that is some form of bookmark, not to be confused with a bookmark flag. If we look at the Delphi code for handling bookmarks it might make things a little clearer:

```

function TDataSet.GetBookmark: TBookmark;
begin
  if BookmarkAvailable then
  begin
    GetMem(Result, FBookmarkSize);
    GetBookmarkData(ActiveBuffer, Result);
  end else
    Result := nil;
end;

procedure TDataSet.GotoBookmark(Bookmark:
  TBookmark);
begin
  if Bookmark <> nil then
  begin
    CheckBrowseMode;
    DoBeforeScroll;
    InternalGotoBookmark(Bookmark);
    Resync([rmExact, rmCenter]);
    DoAfterScroll;
  end;
end;

```

These first two methods refer to a TBookmark which is actually just a pointer that can hold any information that you feel necessary to enable the TDataSet to find its way back to where you got the bookmark from. In our case all we want to do is to use the integer value of the record's position in the table.

```

PBinaryBookmark = ^TBinaryBookmark;
TBinaryBookmark = record
  RecPtr: Integer;
end;

```

Then we need to define two more methods that Delphi will call to retrieve or set a bookmark in a particular buffer:

```

procedure TBinaryTable.GetBookmarkData(Buffer:
  PChar; Data: Pointer);
begin
  Move(Buffer[fBookmarkOfs], Data^,
    SizeOf(TBinaryBookmark));
end;

```

```

procedure TBinaryTable.SetBookmarkData(Buffer:
  PChar; Data: Pointer);
begin
  Move(Data^, Buffer[fBookmarkOfs],
    SizeOf(TBinaryBookmark));
end;

```

The Data parameter is the actual TBookmark or pointer and all that we are doing is copying the bytes in that pointer to and from the buffer. Don't forget, all we are doing here is supplying the information to Delphi in a form that is required, further 'conversations' will make use of what Delphi does with this.

TDataSet also provides a Bookmark property that is defined as being of type TBookmarkStr. This is actually just a string and is provided for a user of the TDataSet to store and return to bookmarks when using the TDataSet as part of a program. If we look at how Delphi makes bookmarks available to applications through the Bookmark property, you will see why we need to write the TBinaryTable methods that dovetail into the TDataSet code:

```

function TDataSet.BookmarkAvailable:
  Boolean;
begin
  Result := (State in [dsBrowse, dsEdit,
    dsInsert]) and not IsEmpty
    and (GetBookmarkFlag(ActiveBuffer) =
    bfCurrent);
end;

function TDataSet.GetBookmarkStr:
  TBookmarkStr;
begin
  if BookmarkAvailable then
  begin
    SetLength(Result, BookmarkSize);
    GetBookmarkData(ActiveBuffer,
      Pointer(Result));
  end else
    Result := '';
end;

procedure TDataSet.SetBookmarkStr(const
  Value: TBookmarkStr);
begin
  GotoBookmark(Pointer(Value));
end;

```

As you can see, bookmarks are a small but important part of a TDataSet and are both used internally and made available externally. In case you were wondering how you get the TDataSet to go back to a bookmark, here is the code that we are going to use in our TBinaryTable:

```

procedure TBinaryTable.InternalGotoBookmark(Bookmark:
    Pointer);
begin
    with PBinaryBookmark(Bookmark) ^ do
        begin
            fCurrentRecord := RecPtr;
        end;
    end;
end;

```

This then allows us to multiply the fCurrentRecord value by the record length to get the offset of the record in the binary data file.

Anatomy of a Thing

Having discussed the structure of the data buffer that will be used by the TBinaryTable, we will now start to look at what else needs to be modified in order to derive TBinaryTable from TDataSet. Let us start by going through the class declaration in small sections, starting with the buffer bits that we have already discussed:

```

TBinaryTable = class(TDataSet)
private
    fRecordSize: Integer;
    // size of the actual data
    fCalcFieldsOfs: Integer;
    // offset of Calculated Fields
    fRecInfoOfs: Integer;
    // offset of RecInfo
    fBookmarkOfs: Integer;
    // offset of Bookmark
    fRecordBufferSize: Integer;
    // size of data + Calc + RecInfo + Bookmark

```

Next we need a few fields to keep track of some more of our custom bits:

```

    fStream: TStream;
    // physical table
    fDatabaseName: string;
    fTableName: TFileName;
    fFileName: string;
    // fDatabaseName + fTableName
    fRecordCount: Integer;
    fCurrentRecord: Integer;
    fCursorOpen: Boolean;
    fFieldOffset: TList;
    function FieldDefsStored: Boolean;

```

Most of these are self explanatory but I will just mention the fFieldOffset. This is a TList that normally stores pointers, but in this instance we are going to cast an integer that holds the offset value of a field in the buffer to a pointer. As both an integer and a pointer are four bytes long, this will work and when we want the integer back, all we have to do is to cast it back.

The final item in the private section of the TBinaryTable is a function that determines whether the FieldDefs property is stored in the .DFM file.

```

protected
    procedure ClearCalcFields(Buffer:
        PChar); override;
    procedure OpenCursor(InfoQuery:
        Boolean); override;
    function GetRecordCount: Integer;
        override;
    function GetRecNo: Integer; override;
    procedure SetRecNo(Value: Integer);
        override;

```

The first few methods in the protected section of the class are marked as virtual in TDataSet and range in basic implementation from empty methods to default behaviour. You only need to override those for which the default behaviour is not adequate. However the following section of protected methods are all, not just virtual but abstract methods. In other words, if you don't override them and write an implementation, Delphi has absolutely no idea how to handle them.

```

    function AllocRecordBuffer: PChar;
        override;
    procedure FreeRecordBuffer(var Buffer:
        PChar); override;
    procedure GetBookmarkData(Buffer: PChar;
        Data: Pointer); override;
    function GetBookmarkFlag(Buffer: PChar):
        TBookmarkFlag; override;
    function GetRecord(Buffer: PChar; GetMode:
        TGetMode; DoCheck: Boolean):
        TGetResult; override;
    function GetRecordSize: Word; override;
    procedure InternalAddRecord(Buffer: Pointer;
        Append: Boolean); override;
    procedure InternalClose; override;
    procedure InternalDelete; override;
    procedure InternalFirst; override;
    procedure InternalGotoBookmark(Bookmark:
        Pointer); override;
    procedure InternalHandleException;
        override;
    procedure InternalInitFieldDefs; override;
    procedure InternalInitRecord(Buffer:
        PChar); override;
    procedure InternalLast; override;
    procedure InternalOpen; override;
    procedure InternalPost; override;
    procedure InternalSetToRecord(Buffer:
        PChar); override;
    function IsCursorOpen: Boolean; override;
    procedure SetBookmarkFlag(Buffer: PChar;
        Value: TBookmarkFlag); override;
    procedure SetBookmarkData(Buffer: PChar;
        Data: Pointer); override;
    procedure SetFieldData(Field: TField;
        Buffer: Pointer); override;

```

These are essentially the missing parts of the 'conversation' between the TBinaryTable class and Delphi. We will go on to discuss these in further detail at a later stage.

```

procedure InitBufferPointers; virtual;
procedure SetDatabaseName(const Value:
    string); virtual;
procedure SetTableName(const Value:
    TFileName);

```

These final protected methods are utility methods that I have created to handle various internal functionality.

```

public
    procedure CreateTable; virtual;
    function GetFieldData(Field: TField; Buffer:
        Pointer): Boolean; override;

```

We are going to change very few public methods as most of the functionality offered is not needed in this particular version. I have implemented `CreateTable` as this allows a user to call this method to write an empty data file based purely on the `FieldDefs` that have been created at design time. `GetFieldData` is a method that allows a user to provide a buffer into which the internal binary field data can be copied and examined outside of the `TBinaryTable`.

```

published
    property DatabaseName: string
        read fDatabaseName
        write SetDatabaseName;
    property TableName: TFileName
        read FTableName
        write SetTableName;

```

The published section consists of two parts; the first is two additional properties for which we will later design property editors. The remainder of the class declaration is taken up by re-declaring properties, mostly events, that are only declared in the protected section of `TDataset`.

```

property Active;
property FieldDefs stored FieldDefsStored;
property BeforeOpen;
property AfterOpen;
property BeforeClose;
property AfterClose;
property BeforeInsert;
property AfterInsert;
property BeforeEdit;
property AfterEdit;
property BeforePost;
property AfterPost;
property BeforeCancel;
property AfterCancel;
property BeforeDelete;
property AfterDelete;
property BeforeScroll;
property AfterScroll;
property OnCalcFields;

```

```

    property OnDeleteError;
    property OnEditError;
    property OnFilterRecord;
    property OnNewRecord;
    property OnPostError;
end;

```

The Shoulder Bone's Connected to The...

So now we have a complete class declaration, all that's left now is to put some flesh on this skeleton. As this article is getting quite long enough already (and Joanna P. our beloved editor is screaming for copy!) I will just go through the first few non-abstract protected methods.

```

implementation
uses
    SchemaFile, Dialogs, Controls, Forms,
    TypInfo;

{ TBinaryTable }

procedure TBinaryTable.ClearCalcFields(Buffer:
    PChar);
begin
    FillChar(Buffer[fCalcFieldsOfs],
        CalcFieldsSize, 0);
end;

```

This method is called in several places by `TDataset` and normally does absolutely nothing, but I felt that it was good practice to clear the `CalcFields` buffer that Delphi manages by setting all the bytes to \$0. Note the use of the `fCalcFieldsOfs` field and the `CalcFieldsSize` property, the setting up of which I will discuss later.

```

procedure TBinaryTable.OpenCursor(InfoQuery:
    Boolean);
begin
    try
        if fDatabaseName = '' then
            DatabaseError(SDatabaseNameMissing);

        inherited OpenCursor(InfoQuery);
    except
        raise;
    end;
end;

```

The `OpenCursor` method is called by `TDataset` and allows you to insert any extra code that you might deem necessary around the opening of the table. In this case I have checked for the existence of a database name and used a constant defined in `BDEConst` to raise an appropriate exception in case of failure. The inherited `TDataset` version of `OpenCursor` that is called reads as follows:

```

procedure TDataSet.OpenCursor(InfoQuery:
    Boolean);
begin
    if InfoQuery then
        InternalInitFieldDefs
    else
        DoInternalOpen;
end;

```

If you want controls such as DBGrid to give you a working scroll bar that is proportional then the following methods will have to be overridden:

```

function TBinaryTable.GetRecordCount:
    Longint;
begin
    Result := inherited GetRecordCount;
    try
        CheckActive;
        Result := fRecordCount;
    except
        Result := -1;
    end;
end;

```

CheckActive does just about what it says and if it fails to find the dataset is active, raises an exception, leaving the Result as -1. If GetRecordCount and GetRecNo are not overridden then DBGrid scroll bars will not appear, as the default behaviour for both these methods is to return -1. DBGrid uses these values to set up the ScrollRange and ScrollPos properties.

```

function TBinaryTable.GetRecNo: Longint;
begin
    UpdateCursorPos;
    if fCurrentRecord < 0 then
        Result := 1
    else
        Result := fCurrentRecord + 1;
end;

```

UpdateCursorPos ensures that the internal buffer is pointing to a valid record. In our implementation, we return the value of fCurrentRecord + 1 as fCurrentRecord is 0 based and we want record numbers to appear to start from 1.

```

procedure TBinaryTable.SetRecNo(Value:
    Integer);
begin
    CheckBrowseMode;
    if (Value > 1) and (Value <=
        fRecordCount) then
    begin
        fCurrentRecord := Value - 1;
        Resync([rmCenter]);
    end;
end;

```

If you have a valid system of record or key numbers then SetRecNo allows you to position the cursor to a particular record or key number. In our case we have kludged a record numbering system to prove how it could work. CheckBrowseMode looks to see if the dataset is editing and if so, posts any changed data, returning the dataset state to dsBrowse. Then we set fCurrentRecord to the appropriate value in preparation for when TDataSet will require it to retrieve data for that record. Resync([rmCenter]) is responsible for telling any multi-row data aware controls that they are to redraw themselves with the selected record halfway down the display.

Conclusion

We have looked at the first stages of creating a TDataSet derivative. The TDataSet class defines one side of a conversation; it is up to us to write our side of that conversation. We have defined the internal buffer layout and discovered how to implement both bookmarks and bookmark flags, essential parts of the conversation. We have designed the basic additional types that are needed by TBinaryTable and then listed those parts of the class that are going to need modifying in some way. Then we went on to start writing the implementation of the non-abstract methods that are going to be used in our TBinaryTable.

Further monster building will ensue!

Copyright © 1999 Joanna Carter

First published in [UK-BUG](#) News May/June 1999.

Not to be reproduced without consent; please contact [Joanna Carter](#)