

Keeping Hold of Your Things

by
Joanna Carter

Part 3

In the preceding articles in this series, we have looked at creating a class for maintaining a schema for a custom dataset, and then we started to construct the dataset itself. In this third article we come to the abstract methods of TDataSet that have to be implemented in order to fulfil our half of the conversation initiated by Delphi: were we not to override these methods we would end up with compile-time warnings and run-time errors!

```
protected
...
// abstract methods - these MUST be
// implemented
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;
```

Now let's go through the implementation for these methods in detail:

```
function TBinaryTable.AllocRecordBuffer: PChar;
begin
    Result := AllocMem(fRecordBufferSize);
end;
```

Delphi requires us to tell it just how much memory is required for a record of physical data; plus any calculated fields; plus the size of the TRecInfo structure; plus the size of the bookmark data. We are also responsible for allocating this memory. The fRecordBufferSize parameter is set in the InitBufferPointers method discussed in the previous article.

If you are creating a TDataSet derivative to translate data from a non Delphi format, e.g. Btrieve, then the size of the record buffer should be that of the physical record data as stored in its native format. Translation of data from native to Delphi types takes place in the GetFieldData and SetFieldData methods; which we will discuss later.

Of course, Delphi expects you to release it as well:

```
procedure TBinaryTable.FreeRecordBuffer(var
    Buffer: PChar);
begin
    FreeMem(Buffer);
end;
```

Finders Keepers

Now we come to the part of our conversation that seems to cause the most problems if not handled correctly... Bookmarks.

The first methods in this section that require implementing are GetBookmarkData and SetBookmarkData:

```
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;
```

GetBookmarkData is called by TDataSet.GetBookmark and the Data parameter is actually a TBookmark – otherwise known as a Pointer. What this method achieves is to copy the data that makes up a bookmark from the part of the record buffer that starts at fBookmarkOfs into the Data parameter; something that Delphi cannot do until it knows just how far along the data you have decided to set the bookmark. The reverse applies to SetBookmarkData.

The next methods are easy to confuse with GetBookmarkData and SetBookmarkData. A BookmarkFlag in fact, has nothing to do with the idea of finding where in the data file you have come from: it is actually stored in the TRecInfo structure and is of type TBookmarkFlag which is defined as an enumerated type of: bfCurrent, bfBOF, bfEOF and bfInserted. Apart from setting the BookmarkFlag to bfCurrent in the GetRecord method, I would recommend you leave Delphi to get on with looking after this item all on its own.

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

procedure TBinaryTable.SetBookmarkFlag(Buffer:
    PChar; Value: TBookmarkFlag);
begin
    PRecInfo(Buffer + fRecInfoOfs)^.BookmarkFlag
        := Value;
end;
```

Once again, all that is going on in these two methods is; us letting Delphi know, where in a record buffer it can find the information it requires.

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

    // or generically:
    with PForeignBookmark(Bookmark)^ do
    begin
        Move physical table to RecPtr;
    end;
end;
```

Gotcha!

The GetRecord method is the point in our conversation where our derived TDataset actually gets the data from the physical file on the disk. The main example we are describing involves accessing data stored in a file in a ‘Delphi-compatible’ format. We will, however, also describe how this should be implemented for non-Delphi format data.

First, I will discuss this method as it would be used if you were accessing Delphi typed data

```
function TBinaryTable.GetRecord(Buffer:
    PChar; GetMode: TGetMode;
    DoCheck: Boolean): TGetResult;
begin
```

TGetMode is defined as (gmCurrent, gmNext, gmPrior) and TGetResult is defined as (grOK, grBOF, grEOF, grError)

GetRecord is called by the following TDataset methods: TDataset.GetNextRecord, TDataset.GetPriorRecord and TDataset.Resync.

This first section determines whether we have fallen off the ends of the file or if there are any records at all: if everything is Ok then it determines the offset of the prior or next record: remember, we are using fRecordCount and fCurrentRecord as a simple means of positioning the cursor in our own format of file.

```
if fRecordCount < 1 then
    Result := grEOF //Don't bother to do anything else
else
begin
    Result := grOk; // preset result to grOk
    case GetMode of
        gmNext:
            if fCurrentRecord >= (fRecordCount -
                1) then
                Result := grEOF
            else
                Inc(fCurrentRecord);
        gmPrior:
            if fCurrentRecord <= 0 then
                Result := grBOF
            else
                Dec(fCurrentRecord);
        gmCurrent:
            if (fCurrentRecord >= fRecordCount) or
                (fCurrentRecord < 0) then
                Result := grError;
    end;
```

Now that we have determined that there is some valid data to read from the file, we can use the fCurrentRecord to position the physical file cursor and retrieve the data.

```
case Result of
    grOk:
    begin
        fStream.Position := fRecordSize *
            fCurrentRecord;
        fStream.ReadBuffer(Buffer^,
            fRecordSize);
```

Now we can let TDataset do the next bit; the OnCalcFields event will be called during the GetCalcFields method and the result of any data assigned to the calculated fields will be handled in and out of the buffer entirely by Delphi.

```
GetCalcFields(Buffer);
```

Then it is our turn again: this time to set up the BookmarkFlag; followed by initialising a Bookmark to the offset of the current record in the physical file.

```

PRecInfo(Buffer + fRecInfoOfs)^.BookmarkFlag :=
    bfCurrent;

with PBinaryBookmark(Buffer + fBookmarkOfs)^ do
begin
    RecPtr := fCurrentRecord;
end;
end;

```

Finally we handle the DoCheck parameter which is our cue to raise an exception if we have found an error condition and DoCheck is true

```

grError:
    if DoCheck then
        raise EBinaryTableError.Create('GetRecord:
            Invalid record');
    end;
end;
end;

```

Natives or Foreigners

Now that we can read in Delphi types, let us look at an example that is accessing data in another, possibly non-standard format. This example uses a mixture of Delphi and pseudocode to illustrate the techniques required and is based on a database that does not know that it is at Bof or Eof until you try to get the record before or after the first/last. To cope with this, fCurrentRecord is of type TFilePos and is defined as (fpBofCrack, fpBof, fpRecord, fpEof, fpEofCrack):

```

function TForeignTable.GetRecord(Buffer:
    PChar; GetMode: TGetMode;
    DoCheck: Boolean): TGetResult;
Var
    TempBuffer: PChar;
begin
    if there are no records then
        Result := grEOF
    else
        begin
            Result := grOk;
            case GetMode of
            gmNext:
                begin
                    initialise TempBuffer;
                    if fCurrentRecord = fpBofCrack then
                        begin
                            get first record into TempBuffer;
                            if successful then
                                begin
                                    Result := grOk;
                                    fCurrentRecord := fpBof;
                                end;
                            end;
                        else
                            begin
                                get next record into TempBuffer;
                                if successful then
                                    begin

```

```

                Result := grOk;
                fCurrentRecord := fpRecord;
            end;
        end;
        if at Eof then
            begin
                Result := grEOF;
                fCurrentRecord := fpEof;
            end;
        end;
        gmPrior:
            begin
                initialise TempBuffer;
                if fCurrentRecord = fpEofCrack then
                    begin
                        get last record into TempBuffer;
                        if successful then
                            begin
                                Result := grOk;
                                fCurrentRecord := fpEof;
                            end;
                        end;
                    else
                        begin
                            get previous record into TempBuffer;
                            if successful then
                                begin
                                    Result := grOk;
                                    fCurrentRecord := fpRecord;
                                end;
                            end;
                        if at Bof then
                            begin
                                Result := grBOF;
                                fCurrentRecord := fpBof;
                            end;
                        end;
                    gmCurrent:
                        try
                            initialise TempBuffer;
                            if fCurrentRecord = fpEofCrack then
                                get last record into TempBuffer
                            else
                                get current record into TempBuffer;
                            fCurrentRecord := fpRecord;
                            Result := grOk;
                        except
                            Result := grError;
                        end;
                    end;

                    if Result = grOk then
                        Move(TempBuffer^, Buffer^, fRecordSize)
                    else
                        begin
                            if (Result = grError) and DoCheck then
                                raise EForeignTableError.Create(ErrorString);
                            end;

                            GetCalcFields(Buffer);

                            PRecInfo(Buffer + fRecInfoOfs)^.BookmarkFlag :=
                                bfCurrent;

                            with PPtrBookmark(Buffer + fBookmarkOfs)^ do
                                begin
                                    initialise TempBuffer;
                                    get bookmark supplied by database
                                        into TempBuffer;
                                    Move(TempBuffer^, RecPtr, BookmarkSize)
                                end;
                            end;
                        end;
                    end;
                end;

```

As you can see, we are simply finding the offset of the required field in the record buffer and then copying the data into the field buffer that is provided in the parameter list. The Boolean bit at the end is connected with null values in calculated fields; I must admit I just copied this bit without question from Steve Troxell's article and it works.

```

procedure TBinaryTable.SetFieldData(Field:
    TField; Buffer: Pointer);
var
    Ptr: PChar;
    Offset: Integer;
begin
    if Field.FieldNo >= 0 then
        begin
            Ptr := ActiveBuffer;
            Offset := Integer(fFieldOffset[Field.FieldNo - 1]);
            Inc(Ptr, Offset);
            FillChar(Ptr^, Field.DataSize, 0);
            if Assigned(Buffer) then
                Move(Buffer^, Ptr^, Field.DataSize);
        end
    else
        begin
            Offset := fCalcFieldsOfs + Field.Offset;
            Boolean(CalcBuffer[Offset]) := not
                Assigned(Buffer);
            if Assigned(Buffer) then
                Move(Buffer^, CalcBuffer[Offset + 1],
                    Field.DataSize);
        end;
    end;

    if not (State in [dsCalcFields]) then
        DataEvent(deFieldChange, LongInt(Field));
end;

```

Once again, SetFieldData finds the offset of the required field in the appropriate record buffer and then moves the data from the field buffer into its correct place in the record buffer. Finally DataEvent is called to force the dataset to re-read the record buffer and update any visual controls.

More Fun with Foreigners

As you can see, dealing with native Delphi data is relatively easy, so I thought I would also show you some idea of how to translate non-Delphi data as well. The technique shown here relies on a small hierarchy of data translation classes that I have designed: they all derive from TBufferTranslator which allows us to use polymorphism to make the code easier both to read and to maintain. I use a header class to keep details of the foreign fields: sizes, locations, etc. in a collection of FieldDef objects. Each FieldDef also knows the appropriate class of converter to

use for translating the data for its related field. I will discuss the TBufferTranslator in the next article but, for now, the only relevant methods used are ReadForeignBuffer and WriteDelphiBuffer.

```

function TForeignTable.GetFieldData(Field:
    TField; Buffer: Pointer): Boolean;
var
    Ptr: PChar;
    Offset: Integer;
    Translator: TBufferTranslator;
    FieldClass: TBufferTranslatorClass;
begin
    if State = dsCalcFields then
        Ptr := CalcBuffer
    else
        Ptr := ActiveBuffer;

    if (Field.FieldNo >= 0) then
        begin
            Offset :=
                fHeader.FieldDef[Field.FieldNo -
                    1].FieldOffset;
            Inc(Ptr, Offset);

```

We obtain the offset of the field in the buffer from the appropriate FieldDef and then use the Translator from that FieldDef to convert the data into a Delphi format appropriate for the field that is passed in. Notice that the constructor for TBufferTranslator is overloaded to allow us to use the name Create both with and without a size parameter:

```

FieldClass := fHeader.FieldDef[Field.FieldNo -
    1].TranslatorClass;
if FieldClass = TBtrString then
    Translator :=
        FieldClass.Create(fHeader.FieldDef[Field.FieldNo
            - 1].FieldSize)
else
    Translator := FieldClass.Create;

with Translator do
    try
        ReadForeignBuffer(Ptr);
        WriteDelphiBuffer(Buffer);
    finally
        Free;
    end;
Result := True;
end
else

```

... handle any calculated fields in exactly the same way as the previous example as they are set up and managed by Delphi in native format.

Finally, finally here is the code for SetFieldData, where once again you can see the use of the TBufferTranslator class to change the format of the data in the reverse direction, this time using ReadDelphiBuffer and WriteForeignBuffer.

```

procedure TForeignTable.SetFieldData(Field:
    TField; Buffer: Pointer);
var
    Ptr: PChar;
    Offset: Integer;
    FieldClass: TBufferTranslatorClass;
    Translator: TBufferTranslator;
begin
    if Field.FieldNo >= 0 then
        // field is not calculated
        begin
            Ptr := ActiveBuffer;
            Offset := fHeader.FieldDef[Field.FieldNo
                - 1].FieldOffset;
            Inc(Ptr, Offset);
            FillChar(Ptr^, Field.DataSize, 0);

            if Assigned(Buffer) then
                begin
                    FieldClass := fHeader.FieldDef[Field.FieldNo
                        - 1].TranslatorClass;
                    if FieldClass = TBtrString then
                        Translator :=
                            FieldClass.Create(fHeader.FieldDef[Field.FieldNo
                                - 1].FieldSize)
                    else
                        Translator := FieldClass.Create;

                    with Translator do
                        try
                            ReadDelphiBuffer(Buffer);
                            WriteForeignBuffer(Ptr);
                        finally
                            Free;
                        end;
                    end;
                end;
            end
        else
            ...

```

Copyright © 1999 Joanna Carter

First published in [UK-BUG](#) News July/August 1999.

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

Conclusion

In this article we have looked at overriding some of the abstract methods of TDataset necessary to fulfil our half of the conversation Delphi wants to have with us about how to read our own type of data.

We started off by allocating enough memory to hold all the information we need for each record. Then we looked at handling Bookmarks and BookmarkFlags; we found that these do different jobs but are both stored in the record buffer. Getting the data from the physical file is the responsibility of the GetRecord method, but when it comes to getting the data to and from the fields and ultimately any data-aware controls we need to use the GetFieldData and SetFieldData methods. Converting non-Delphi data is handled in the GetFieldData and SetFieldData methods using a TBufferTranslator class hierarchy to improve readability and maintenance of code.

Next time I will conclude looking at TBinaryTable by running through the remaining abstract methods and showing you the class hierarchy used for translating the field data.