Surviving Client/Server: Custom Dataset Components, 1

by Steve Troxell

With the release of Delphi 3, Borland has redesigned much of the database VCL to make it easier to develop database applications that are not reliant on the BDE. Previously, BDE interface calls were embedded throughout the VCL, forcing third-party developers to write "BDE replacements" which involved designing an interface that emulated the BDE API, replacing one or more of Delphi's database units with custom versions, and rebuilding the component library to force the database components to access the new database code rather than Delphi's default BDE-oriented code.

With Delphi 3, all the BDE interface calls have been pulled out of the `TDataSet` class and encapsulated in a new layer called `TBDEDataSet`. In addition, all the BDE-specific classes have been removed from the `DB` unit and placed in `DBTables` so that the `DB` unit can be used within an application without automatically pulling in the BDE. `TDataSet` is now fully abstracted such that there are no dependencies on any particular database API or file format. By deriving a new class descending from `TDataSet`, we can provide methods specific to any file architecture we like.

The aim of this article is to walk through the process of creating a custom `TDataSet` descendant for a non-BDE file architecture. Along the way we will point out exactly what `TDataSet` methods we should override and why. To avoid getting bogged down in the specifics of any particular database API, we will use a simple untyped binary file architecture using Delphi's standard file I/O functions. Our data file format is just a file of fixed length records, easily accessible using Delphi's `Seek`, `BlockRead` and `BlockWrite` procedures. The test table we will use for our examples has the structure shown in Listing 1. To handle deleted records, the first byte of each record is a deleted flag. If this byte is non-zero the record is deleted. For simplicity we won't try to reclaim deleted record space for newly inserted records.

TDataSet

`TDataSet` manages a number of activities for us: calling event handlers, buffering records, handling high-level dataset navigation, etc. This frees us from worrying about most aspects of datasets except the specific physical implementation of our database. `TDataSet` defines a plethora of abstract methods which we must override in a descendant component to handle these physical details.

`TDataSet` buffers records internally and will hold as many records from the dataset as are visible on the screen at one time (like in a grid for example). What we need to keep in mind about this is that when handling any particular record buffer, the record in question may not necessarily align with the current cursor position in the physical file. For example, if a 10 row grid is filled with the first 10 records from a table the current physical cursor position is most likely to rest on the 11th record in the table. However, the user may scroll freely among the visible records in the grid without changing the physical cursor position.

Basic File I/O

The first obvious bits of functionality we will need are the basics of opening, closing, reading, and navigating our file. Listing 2 shows a first pass at our custom `TMyDataSet` component and the abstract methods we'll cover in this article.

Our dataset component includes a `TableName` property to allow us to specify the name of the external file we wish to open. Normally, you would find this at a `TTable` level, but for now, we need a way to identify the external file.

Also, since our data file does not store its record definition internally, `TDataSet` has no way of determining the fields or record length of our file (which we will take care of next month). Our application will have to provide the record definition (Listing 1) and we redefine the `RecordSize` property to make it read/write so our application can specify the record length for us. These are the two requirements of the calling program to open a table with our basic dataset component: provide a filename and a record length before calling `Open`.

The `InternalOpen` and `InternalClose` methods are where we need to open and close our file. Listing 3 shows our implementation. We set the `FCursorOpen` flag field which we use to implement the `IsCursorOpen` abstract function. `TDataSet` uses this internally while managing its record buffers.

`TDataSet`'s internal record buffers contain more information than just the record data. We will be adding info to support bookmarks, update status, etc and need to account for this in the buffer size. Within `InternalOpen` we calculate the record buffer size to be the size of the physical record plus the extra space needed for our

➤ *Listing 1*

```
type
  PTestRec = ^TTestRec;
  TTestRec = record
    DelFlag: Byte;
    EmpNo: SmallInt;
    FirstName: string[15];
    LastName: string[20];
    HireDate: TDateTime;
    DeptNo: string[3];
    Salary: Double;
  end;
```

```
type
  PExtraRecInfo = ^TExtraRecInfo;
  TExtraRecInfo = record
    RecordNumber: LongInt;
    BookmarkFlag: TBookmarkFlag;
  end;
  TBookmarkInfo = LongInt;
  TMyDataSet = class(TDataSet)
  private
    FBookmarkOffset: LongInt;  { Offset to bookmark data in recbuf }
    FCursorOpen: Boolean;      { True if cursor is open }
    FInternalFile: file;       { File variable }
    FRecSize: Word;            { Physical size of record }
    FRecBufSize: Word;         { Total size of recbuf }
    FExtraRecInfoOffset: Word; { Offset to extra rec info in recbuf }
    FTableName: TFileName;     { External filename to open }
  protected
    { basic file reading and navigation }
    function AllocRecordBuffer: PChar; override;
    procedure FreeRecordBuffer(var Buffer: PChar); override;
    function GetCurrentRecord(Buffer: PChar): Boolean; override;
    function GetRecord(Buffer: PChar; GetMode: TGetMode; DoCheck: Boolean):
      TGetResult; override;
    function GetRecordCount: Integer; override;
    function GetRecordSize: Word; override;
    function GetRecNo: Integer; override;
    procedure InternalClose; override;
    procedure InternalFirst; override;
    procedure InternalLast; override;
    procedure InternalOpen; override;
    function IsCursorOpen: Boolean; override;
    { bookmarks }
    function BookmarkValid(Bookmark: TBookmark): Boolean; override;
    function CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
      override;
    procedure GetBookmarkData(Buffer: PChar; Data: Pointer); override;
    function GetBookmarkFlag(Buffer: PChar): TBookmarkFlag; override;
    procedure SetBookmarkData(Buffer: PChar; Data: Pointer); override;
    procedure SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag); override;
    procedure InternalGotoBookmark(Bookmark: Pointer); override;
    procedure InternalSetToRecord(Buffer: PChar); override;
    { basic file modification }
    procedure InternalInitRecord(Buffer: PChar); override;
    procedure InternalEdit; override;
    procedure InternalDelete; override;
    procedure InternalPost; override;
   public
    {TDataSet properties}
    property RecordSize: Word read GetRecordSize write FRecSize;
    {descendant properties}
    property TableName: TFileName read FTableName write FTableName;
 end;
```

➤ *Listing 2*

```
procedure TMyDataSet.InternalOpen;
begin
  FRecBufSize := FRecSize + SizeOf(TExtraRecInfo);
  FExtraRecInfoOffset := FRecSize;
  AssignFile(FInternalFile, FTableName);
  Reset(FInternalFile, 1);   { Open a file of bytes }
  FCursorOpen := True;
end;
function TMyDataSet.AllocRecordBuffer: PChar;
begin
  Result := StrAlloc(FRecBufSize);
end;
procedure TMyDataSet.FreeRecordBuffer(var Buffer: PChar);
begin
  StrDispose(Buffer);
end;
procedure TMyDataSet.InternalClose;
begin
  CloseFile(FInternalFile);
  FCursorOpen := False;
end;
function TMyDataSet.IsCursorOpen: Boolean;
begin
  Result := FCursorOpen;
end;
```

➤ *Listing 3*

additional info. `FRecBufSize` always refers to the size of the record buffer and will always be larger than the size of the physical record (in `FRecSize`). Also, `FExtraRecInfoOffset` always points to the start of our "extra" record information in the buffer.

In the course of opening the table, `TDataSet` allocates memory for its internal record buffers using `AllocRecordBuffer`, which we must override since it is up to us to determine how big our records are. We must also override `FreeRecordBuffer` to release this memory.

Reading Records

Next, we'll need to handle a simple loop through all the records in the table. Fortunately, all record retrieval is encapsulated in one method: `GetRecord`. `GetRecord` is passed a pointer to the internal record buffer, and a flag indicating whether the current, next or previous record is desired. For the moment we are only concerned with the next record. The return value of this function indicates success, EOF, BOF or error. This return condition is `TDataSet`'s sole means of determining BOF and EOF conditions on the result set. The physical file's EOF indicator is not used directly by `TDataSet`.

Listing 4 shows our partial implementation for `GetRecord`. We loop through the file until we find the next undeleted record (or EOF). Because records are buffered internally, we cannot use the physical file pointer to reliably report the sequence number for any given record. We might request the record number for a buffered record while the physical file pointer pointed to a completely different record. Therefore, as we read in each record from the file, we store its record number right in the record buffer within the "extra record info" area we set aside. Now whenever we are referring to a given record, we always have its number regardless of the current state of the physical file pointer.

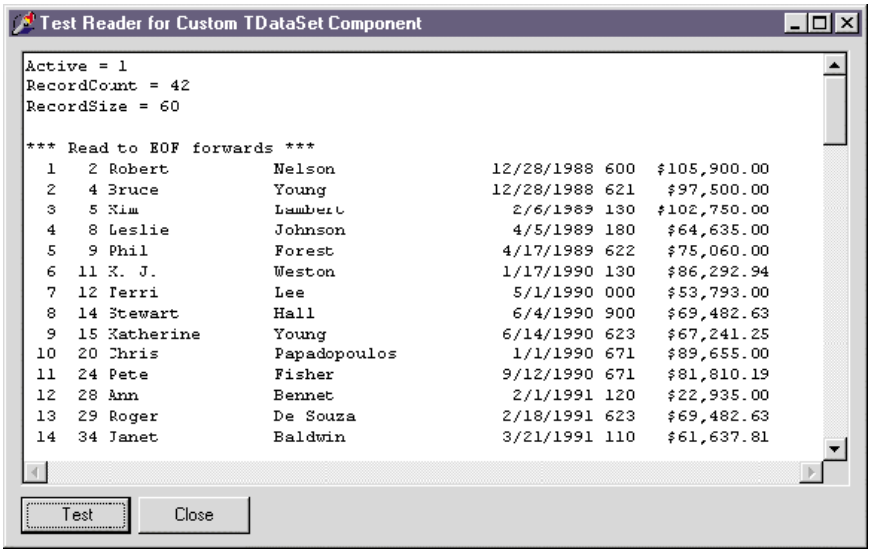
Finally, we need a way to access the record data. Eventually we will implement complete `TFieldDef` components for each field in the record, but for the moment we'll just read the raw record buffer. `TDataSet.ActiveBuffer` always points to the current record buffer and is maintained automatically.

Eventually, our internal buffers will contain extra information for bookmarks, update status, etc, so the internal buffer will be larger than the actual record data. We need to implement the abstract

method GetRecordSize to return the actual size of the record data instead of the size of the internal buffer (see Listing 5). This method supports the TDataSet.RecordSize property.

We should also override the GetCurrentRecord method to fill an application supplied buffer with the contents of the current record. This is simply a matter of testing for an empty dataset (with the IsEmpty internal method) and copying the record data from ActiveBuffer as shown in Listing 5. Note that we use the physical record size rather than the buffer size, because the calling application will not be aware of the extra buffer information we are going to tack on later.

We are now capable of supporting the simple table traversal code shown in Listing 6, with its output shown in Figure 1. This routine simply opens the table, steps forward through the table until EOF, displays the contents of each record and closes the table. Remember, our TTestRec record definition (Listing 1) is defined within the application itself and we are accessing the raw record data with GetCurrentRecord.



➤ *Figure 1*

The implementation of the dataset's simple RecordCount and RecNo properties is also in Listing 5.

First and Last Methods

Now we need to embellish our dataset component to support the First, Last and Prior dataset methods. TDataSet calls InternalFirst to set the file pointer to the beginning of file, and then calls GetRecord to read the next record from the current file position. Likewise, TDataSet calls InternalLast to set the file pointer to the end of file, then calls GetRecord to read the previous record from the current file position. Our implementations are shown in Listing 7. For InternalLast, we set the file pointer to one record beyond the physical end of file, forcing the system Eof function to return True.

Reading Prior Records

Now we must expand GetRecord to handle reading the record before the current file position. Listing 7 shows our expanded GetRecord method. Reading prior records becomes a bit tricky when we have to account for BOF and EOF conditions. In general, when we read the next record of a file, we read the record from the current file position and advance the file position to the end of the record we just read (the beginning of the subsequent record). When reading a prior record, we must move the file pointer backwards by *two* records

➤ *Listing 4*

```
{Note: TGetMode and TGetResult are defined in the DB unit}
function TMyDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
begin
  Result := grOk;
  case GetMode of
    gmNext:
      { read next record, skipping deleted records }
      repeat
        if System.Eof(FInternalFile) then
          Result := grEOF
        else
          BlockRead(FInternalFile, Buffer^, FRecSize);
      until (Result <> grOk) or (Byte(Buffer^) = 0);
    else
      Result := grError;
end;
  { Store record number in the buffer }
  if Result = grOk then
    with PExtraRecInfo(Buffer + FExtraRecInfoOffset)^ do
      RecordNumber := (FilePos(FInternalFile) div FRecSize) - 1;
end;
```

➤ *Listing 5*

```
function TMyDataSet.GetRecordSize: Word;
begin
   Result := FRecSize;
end;
function TMyDataSet.GetCurrentRecord(Buffer: PChar): Boolean;
begin
  Result := False;
  if not IsEmpty then begin
    Result := True;
    Move(ActiveBuffer^, Buffer^, RecordSize);
  end;
end;
function TMyDataSet.GetRecordCount: Integer;
begin
  Result := FileSize(FInternalFile) div FRecSize;
end;
function TMyDataSet.GetRecNo: Integer;
begin
  { Because of Delphi's internal record buffering, we must read the stored record
    number, not the current physical file position }
  Result := PExtraRecInfo(ActiveBuffer + FExtraRecInfoOffset)^.RecordNumber;
end;
```

```
procedure TForm1.DumpCurrentRec1;
var RecBuffer: TTestRec;
begin
  if MyDataSet.GetCurrentRecord(@RecBuffer) then with RecBuffer do
    Memo1.Lines.Add(Format('%3d %3d %-15s %-20s %10s %3s %12m',
      [MyDataSet.RecNo, EmpNo, FirstName, LastName, DateToStr(HireDate),
      DeptNo, Salary]));
end;
procedure TForm1.btnTestClick(Sender: TObject);
begin
  with MyDataSet do begin
    TableName := 'TEST1.DAT';
    RecordSize := SizeOf(TTestRec);
    Open;
    try
      Memo1.Lines.Add('Active = ' + IntToStr(Ord(Active)));
      Memo1.Lines.Add('RecordCount = ' + IntToStr(RecordCount));
      Memo1.Lines.Add('RecordSize = ' + IntToStr(RecordSize));
      Memo1.Lines.Add('');
      Memo1.Lines.Add('*** Read to EOF forwards ***');
      while not Eof do begin
        DumpCurrentRec1;
        Next;
      end;
    finally
      Close;
    end;
  end;
end;
```

➤ *Listing 6*

```
procedure TMyDataSet.InternalFirst;
begin
  Seek(FInternalFile, 0);
end;
procedure TMyDataSet.InternalLast;
begin
  { force system eof condition }
  Seek(FInternalFile, FileSize(FInternalFile));
end;
function TMyDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
var FilePosition: LongInt;
begin
  Result := grOk;
  case GetMode of
    gmNext:
      { read next record, skipping deleted records }
      repeat
        if System.Eof(FInternalFile) then
          Result := grEOF
        else
          BlockRead(FInternalFile, Buffer^, FRecSize);
      until (Result <> grOk) or (Byte(Buffer^) = 0);
    gmPrior:
      repeat
        FilePosition := FilePos(FInternalFile);
        if FilePosition < (2 * FRecSize) then
          Result := grBOF
        else begin
          if Eof then
            Seek(FInternalFile, FileSize(FInternalFile) - FRecSize)
          else
            Seek(FInternalFile, FilePosition - (2 * FRecSize));
          BlockRead(FInternalFile, Buffer^, FRecSize);
        end;
      until (Result <> grOk) or (Byte(Buffer^) = 0);
    else
      Result := grError;
  end;
  { Store record number in the buffer }
  if Result = grOk then with PExtraRecInfo(Buffer + FExtraRecInfoOffset)^ do
    RecordNumber := (FilePos(FInternalFile) div FRecSize) - 1;
end;
```

➤ *Listing 7*

to position ourselves at the start of the record *before* the one we just read. Then by reading that record we leave the file pointer at the end of the record we just read.

To account for running into BOF while reading prior records, we must check our current position in the file. If we are currently at BOF, or have just read the first record and are currently pointing to the second record, then there is no "prior" record to fetch and we return a BOF condition.

Accounting for EOF is a little trickier. We might be tempted to use the `SysUtils.Eof` function to detect EOF on our untyped file, then position to the last record and read it. However, after reading the last record, `SysUtils.Eof` again returns true. So fetching the prior record after setting the file pointer to EOF (as with the `Last` method) results in an infinite loop as we keep reading the last record and falling back into the EOF state.

That is why we must be careful to use the `TDataSet.Eof` method to test for EOF. `TDataSet` manages the EOF status internally based on the value returned from `GetRecord` and other means, so we can rely on it to show our logical position in the dataset without falling into the loop produced by the physical EOF. The act of reading the last record doesn't result in an EOF on the dataset; that only occurs after we attempt to read beyond the last record in the table.

With all this in place, we can now support reading backwards through the table with the code shown below and the output shown in Figure 2:

```
Memo1.Lines.Add(
  '*** Read to BOF backwards ***');
Last;
while not Bof do begin
  DumpCurrentRec1;
  Prior;
end;
```

Bookmarking Records

The next layer of dataset navigation we will implement is bookmarking. Bookmarking allows us to mark the current record, move anywhere else in the dataset, then return to the bookmarked record at will. We can have as many bookmarks as we can to store in our application.

Internally, `TDataSet` relies on two pieces of information to implement bookmarks. Each record buffer holds bookmark data and a bookmark flag. The bookmark data is simply the data necessary to return to that record. In our example, we only need to know the record number. In contrast, the BDE requires a bookmark packet returned by the `DbiGetBookmark` interface function. Although we already have a record number field in our "extra record info", to make our example more realistic we will store our bookmark data in such a

way that it is independent of any other piece of information in the buffer.

Since TDataSet can buffer many records internally, the user can request a bookmark for a specific record while the physical cursor position points to a different record altogether. Instead of repositioning the physical cursor to match the buffered record so we may obtain bookmark information (ie so TBDEDataSet can call DbiGetBookmark), TDataSet expects that bookmark data will be fetched as each record is read and stored within the record buffer as extra data. Then when a bookmark is requested, it simply gets the necessary information out of the record buffer rather than making a request of the physical database.

The bookmark flag stored within the record buffer is used internally by TDataSet to handle record positioning. We set it to bfCurrent upon reading the record and let TDataSet handle it from there on.

Implementing Bookmarks

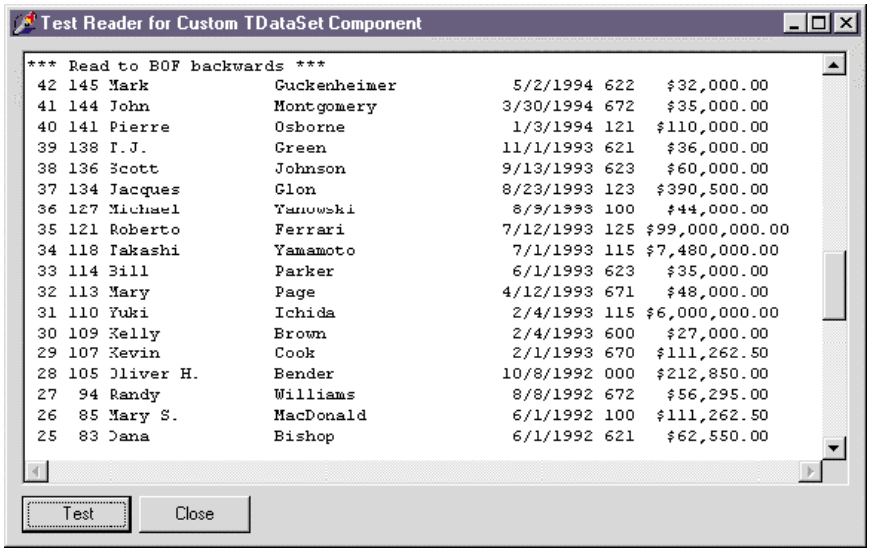
Within the record buffer, the bookmark flag is already accounted for in our "extra record info" packet. To accommodate the bookmark data, we add it onto the end of our record buffer. We need to change InternalOpen as shown in Listing 8. BookmarkSize is a property of TDataSet and our TBookmarkInfo is simply a LongInt to hold the record number.

Finally, we set FBookmarkOffset to point to the start of our bookmark data within the buffer. We also need to change GetRecord to populate the record buffer with bookmark data.

Listing 8 also shows the implementation of the four methods TDataSet uses to access this bookmark information for a record of interest. In each case a pointer to the buffer for the record of interest is passed in and we simple copy the bookmark information in or out of the buffer.

The public GetBookmark and FreeBookmark methods are handled automatically by TDataSet since it is aware of the size of the bookmark data. The GotoBookmark method ultimately calls the abstract method InternalGotoBookmark, which we must override for our specific file structure.

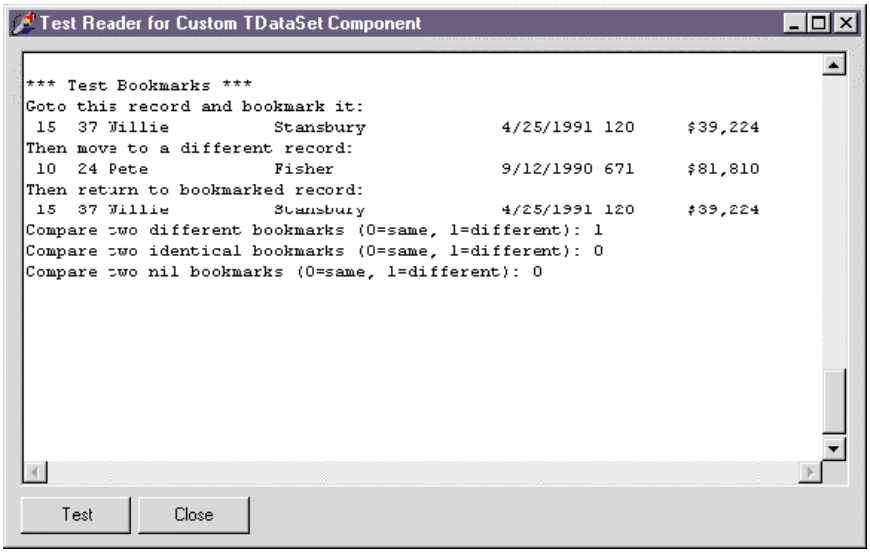
Since our bookmark data consists simply of the record number, we position the physical file pointer to the record of interest. Remember the file pointer always points to the end of the record we just read, so we take that into account when returning to a bookmark. TDataSet then fetches the record data by calling GetRecord and asking for the current record, so our file pointer must be properly positioned at the end of the current record, because GetRecord will back up one record length to reread the current record.



➤ *Figure 2*

➤ *Listing 8*

```
procedure TMyDataSet.InternalOpen;
begin
  BookmarkSize := SizeOf(TBookmarkInfo);
  FRecBufSize := FRecSize + SizeOf(TExtraRecInfo) + BookmarkSize;
  FExtraRecInfoOffset := FRecSize;
  FBookmarkOffset := FExtraRecInfoOffset + SizeOf(TExtraRecInfo);
  AssignFile(FInternalFile, FTableName);
  Reset(FInternalFile, 1);   { Open a file of bytes }
  FCursorOpen := True;
end;
function TMyDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
begin
  {... lines omitted ...}
  { Store record number in the buffer }
  if Result = grOk then begin
    with PExtraRecInfo(Buffer + FExtraRecInfoOffset)^ do begin
      RecordNumber := (FilePos(FInternalFile) div FRecSize) - 1;
      BookmarkFlag := bfCurrent;
      SetBookmarkData(Buffer, @RecordNumber);
    end;
  end;
end;
procedure TMyDataSet.GetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  Move(Buffer[FBookmarkOffset], Data^, BookmarkSize);
end;
function TMyDataSet.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
begin
  Result := PExtraRecInfo(Buffer + FExtraRecInfoOffset).BookmarkFlag;
end;
procedure TMyDataSet.SetBookmarkData(Buffer: PChar; Data: Pointer);
begin
  Move(Data^, Buffer[FBookmarkOffset], BookmarkSize);
end;
procedure TMyDataSet.SetBookmarkFlag(Buffer: PChar; Value: TBookmarkFlag);
begin
  PExtraRecInfo(Buffer + FExtraRecInfoOffset).BookmarkFlag := Value;
end;
procedure TMyDataSet.InternalGotoBookmark(Bookmark: Pointer);
  { position physical file to bookmarked record }
begin
  { Position AFTER the record, as though we just read it }
  Seek(FInternalFile, (TBookmarkInfo(Bookmark^) + 1) * FRecSize);
end;
```



➤ *Figure 3*

```
function TMyDataSet.BookmarkValid(Bookmark: TBookmark): Boolean;
var DelFlag: Byte;
begin
  Result := Assigned(Bookmark) and (TBookmarkInfo(Bookmark^) > 0)
            and (TBookmarkInfo(Bookmark^) <= RecordCount);
  if Result then begin
    CursorPosChanged;  { physical position no longer matches logical position }
    try
      Seek(FInternalFile, TBookmarkInfo(Bookmark^) * FRecSize);
      BlockRead(FInternalFile, DelFlag, 1);
      Result := DelFlag = 0;  { check for a deleted record }
    except
      Result := False;
    end;
  end;
end;
function TMyDataSet.CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
begin
  { bookmarks are equal if they are both nil or they both have the same value }
  if Bookmark1 = Bookmark2 then
    Result := 0
  else begin
    Result := 1;
    if Assigned(Bookmark1) and Assigned(Bookmark2) then
      if TBookmarkInfo(Bookmark1^) = TBookmarkInfo(Bookmark2^) then
        Result := 0;
  end;
end;
function TMyDataSet.BookmarkValid(Bookmark: TBookmark): Boolean;
var DelFlag: Byte;
begin
  Result := Assigned(Bookmark) and (TBookmarkInfo(Bookmark^) > 0)
            and (TBookmarkInfo(Bookmark^) <= RecordCount);
  if Result then begin
    CursorPosChanged;  { physical position no longer matches logical position }
    try
      Seek(FInternalFile, TBookmarkInfo(Bookmark^) * FRecSize);
      BlockRead(FInternalFile, DelFlag, 1);
      Result := DelFlag = 0;  { check for a deleted record }
    except
      Result := False;
    end;
  end;
end;
function TMyDataSet.CompareBookmarks(Bookmark1, Bookmark2: TBookmark): Integer;
begin
  { bookmarks are equal if they are both nil or they both have the same value }
  if Bookmark1 = Bookmark2 then
    Result := 0
  else begin
    Result := 1;
    if Assigned(Bookmark1) and Assigned(Bookmark2) then
      if TBookmarkInfo(Bookmark1^) = TBookmarkInfo(Bookmark2^) then
        Result := 0;
  end;
end;
```

➤ *Listing 9*

There are also two additional methods for bookmarks: `BookmarkValid` and `CompareBookmarks`. Since bookmark data is specific to the database used, we must override these methods and provide our own implementations as shown in Listing 9. Comparing two bookmarks is a straight-forward task. To validate our bookmarks, we simply need to know that it is a valid record number and that it does not point to a deleted record. We must actually go to the physical record and read its deleted flag to check this. Whenever we alter the physical file position such that it is no longer aligned with how `TDataSet` filled its current record buffers, we must invalidate `TDataSet`'s internal tracking of the physical file position by calling the `CursorPosChanged` internal method. `TDataSet` then knows to resynchronize the physical file position.

Our custom dataset component now supports the bookmarking code fragment shown in Listing 10 with output shown in Figure 3.

Deleting Records

Now that we've covered the basics of reading data, let's turn to modifying data content.

Deleting records is simplest, so we'll start there. All we need to do is override the `InternalDelete` abstract method. For our example data file, we simply mark the record as deleted. `TDataSet` aligns the physical file position correctly before calling `InternalDelete` so we are safe in assuming the physical file points to the record to delete. See Listing 11.

The current file position is always at the end of the record we just read, so we back up, rewrite the first byte of the record (the deleted flag), and then position ourselves back at the end of the record we just deleted. Even if the record following the one we just deleted is also deleted, the looping logic in `GetRecord` will make sure we end up positioned on the next undeleted record.

Updating Records

To support editing records we must support the public methods `Edit` and `Post` and we do that by overriding the abstract methods `InternalEdit` and `InternalPost` as shown in Listing 12. Like `Delete`, `TDataSet` ensures that the physical file pointer is positioned correctly before calling these methods.

When the Edit method is called, all we really need to do is refresh our copy of the record. This is also where we might wish to implement pessimistic record locking (which we'll avoid in this issue to avoid extra complicatiuons). Post handles both editing and inserting records, so we must account for both activities. When editing a record, we just position the file pointer to the existing record and rewrite the data from the record buffer. When inserting a record, we will position the file pointer to the end of the file so we may append a new record. A more elaborate scheme would attempt to reclaim deleted record space.

In the calling application, since we are accessing the raw record contents, we simply modify the fields directly in the record buffer as shown below.

➤ *Listing 10*

```
Memo1.Lines.Add('*** Test Bookmarks ***');
First;
MoveBy(10);
Memo1.Lines.Add('Goto this record and bookmark it:');
DumpCurrentRec1;
BookmarkA := GetBookmark;
try
  MoveBy(-5);
  Memo1.Lines.Add('Then move to a different record:');
  DumpCurrentRec1;
  BookmarkB := GetBookmark;
  try
    if BookmarkValid(BookmarkA) then begin
      GotoBookmark(BookmarkA);
      Memo1.Lines.Add('Then return to bookmarked record:');
      DumpCurrentRec1;
    end;
    Memo1.Lines.Add('Compare two different bookmarks (0=same, 1=different): ' +
      IntToStr(CompareBookmarks(BookmarkA, BookmarkB)));
    Memo1.Lines.Add('Compare two identical bookmarks (0=same, 1=different): ' +
      IntToStr(CompareBookmarks(BookmarkA, BookmarkA)));
    Memo1.Lines.Add('Compare two nil bookmarks (0=same, 1=different): ' +
      IntToStr(CompareBookmarks(nil, nil)));
  finally
    FreeBookmark(BookmarkB);
  end;
finally
  FreeBookmark(BookmarkA);
end;
```

➤ *Listing 11*

```
procedure TMyDataSet.InternalDelete;
var
  DelFlag: Byte;
  FilePosition: LongInt;
begin
  FilePosition := FilePos(FInternalFile) - FRecSize;
  Seek(FInternalFile, FilePosition);
  DelFlag := 255;
  BlockWrite(FInternalFile, DelFlag, 1);
  Seek(FInternalFile, FilePosition + FRecSize);
end;
```

➤ *Listing 12*

```
procedure TMyDataSet.InternalEdit;
begin
  { Refresh the current record }
  Seek(FInternalFile, FilePos(FInternalFile) - FRecSize);
  BlockRead(FInternalFile, ActiveBuffer^, FRecSize);
end;
procedure TMyDataSet.InternalPost;
begin
  case State of
    dsEdit:
      begin
        Seek(FInternalFile, FilePos(FInternalFile) - FRecSize);
        BlockWrite(FInternalFile, ActiveBuffer^, FRecSize);
      end;
    dsInsert:
      begin
        Byte(ActiveBuffer^) := 0;  { reset deleted flag }
        Seek(FInternalFile, FileSize(FInternalFile));
        BlockWrite(FInternalFile, ActiveBuffer^, FRecSize);
      end;
  end;
end;
```

```
Last; { modify last record }
Edit;
with PTestRec(ActiveBuffer)^ do
  Salary := Salary + 1000;
Post;
```

Inserting Records

When adding a new record, TDataSet creates a new empty record buffer and calls InternalInitRecord to perform any special initialization we might require. For our example, we're simply going to ensure that the buffer is cleared:

```
procedure
  TMyDataSet.InternalInitRecord(
    Buffer: PChar);
begin
  FillChar(Buffer^,
  FRecBufSize, #0);
end;
```

Since we've already implemented Post for inserting records, we can now support this code fragment:

```
Insert;
with PTestRec(ActiveBuffer)^
  do begin
  EmpNo := 444;
  FirstName := 'NewGuy';
  LastName := 'Inserted';
  HireDate := Date;
  DeptNo := '621';
  Salary := 100000;
end;
Post;
```

Conclusion

We have begun to develop our own custom dataset component and already have a good deal of usable functionality. We can fully navigate and bookmark a table as well as delete, edit, or insert records. Borland finally did a great job in making the database components extensible for any data file format.

Next month, we will continue by adding TField support, indexes, and more.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@turbopower.com or on CompuServe at STroxell.