Surviving Client/Server: Custom Dataset Components, 2

by Steve Troxell

Delphi 3 includes a substantially revamped database VCL and one of the most notable changes was the total abstraction of the TDataSet class to remove all dependencies on the Borland Database Engine for database access. Delphi developers are now free to create their own custom dataset descendants for whatever data format they like. Last month we began making our own custom dataset class around a simple typed file of records, using Delphi's standard file I/O procedures (Reset, Seek, BlockRead, BlockWrite) as the API for our "database". In our last exciting episode, we were able to do all the basic table navigation commands, bookmark records, and delete, edit, and insert records by writing directly to the record buffer. This month we'll add support for TField components and data-aware controls.

A True Component

The code we developed last month wasn't truly a component in the sense that it could be installed in the Component Palette, dropped on a form, and manipulated through the Object Inspector. This month we'll create a TMyTable non-visual component to emulate as much of Delphi's TTable component as is practical. Our TMyDataSet class will remain a separate entity, encapsulating the data access, while TMyTable provides an interface for that data. Listing 1 shows the code for our TMyTable component.

As you can see, much of our component simply consists of surfacing some methods and properties from TDataSet, with a few minor adaptations for our purposes. We no longer have to preset the table's record length since we will be defining the fields in the table, so the RecordSize property is not published and returns to being a readonly property in TMyDataSet. Listing 2 shows the interface to our TMyDataSet class, with the changes we've made since last month highlighted in red.

```
unit MyTable;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms,
  Dialogs, Db, MyDS;
type
  TMyTable = class(TMyDataSet)
    private
      FReadOnly: Boolean;
    protected
      function GetCanModify: Boolean; override;     { Derived from TDataSet }
    public
      property CanModify: Boolean read GetCanModify; { Derived from TDataSet }
    published
      { Derived from TDataSet }
      property Active;
      property AutoCalcFields;
      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 OnNewRecord;
      property OnPostError;
      { Derived from TMyDataSet }
      property TableName;
      { TMyTable Properties }
      property ReadOnly: Boolean read FReadOnly write FReadOnly;
  end;
procedure Register;
implementation
function TMyTable.GetCanModify: Boolean;
begin
  Result := not FReadOnly;
end;
procedure Register;
begin
  RegisterComponents('Data Access', [TMyTable]);
end;
end.
```

➤ *Listing 1*

The Internal Record Buffer

As we saw last month, TDataSet stores much more than the raw record data in its internal record buffers. We're going to be adding a bit more this month. Figure 1 shows a diagram of the internal record buffer and the relative positions of the various segments. Our TMyDataSet descendant keeps track of each of these segments with offset values stored in the variables shown at the bottom of the diagram.

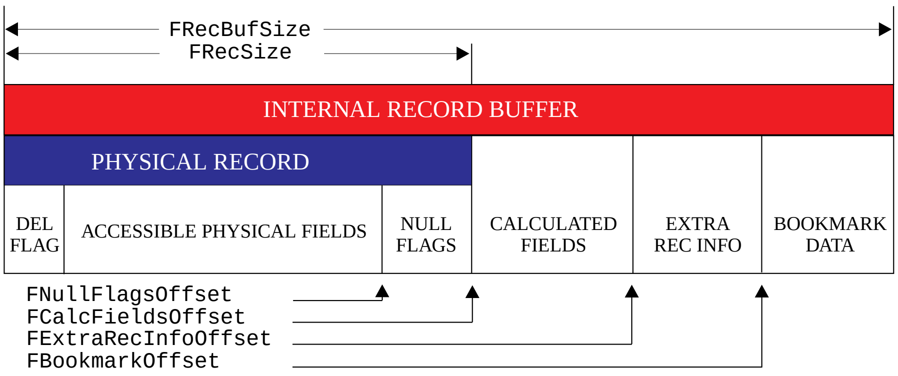
The contents of these data segments in the record buffer were described last month, with the exception of two new ones we are adding this month: null flags and calculated fields.

Null Field Values

In order to take advantage of Delphi's data aware controls, we're going to need to provide

```
TMyDataSet = class(TDataSet)
  private
    FBookmarkOffset: LongInt;  { Offset to bookmark data in recbuf }
    FCalcFieldsOffset: Word;   { Offset to calculated fields data }
    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 }
    FNullFlagsOffset: Word;    { Offset to null flags in recbuf }
  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;
    { field component stuff }
    procedure InternalInitFieldDefs; override;
    function GetFieldData(Field: TField; Buffer: Pointer): Boolean; override;
    procedure SetFieldData(Field: TField; Buffer: Pointer); override;
    procedure InternalAddRecord(Buffer: Pointer; Append: Boolean); override;
    { calculated fields }
    procedure ClearCalcFields(Buffer: PChar); override;
  protected
    FieldOffsets: TList;
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    property RecordSize: Word read GetRecordSize;  { from TDataSet }
    property TableName: TFileName read FTableName write FTableName;
  end;
```

➤ *Listing 2*



➤ *Figure 1*

support for `TField` components in our dataset (plus fields are just a neat thing to have when working with data). Fields in Delphi have built in support for null values and to be truly robust, we should build in support for null values in our component. But since we are using standard Delphi simple datatypes to store values, how do we indicate a null value? We can't simply choose an arbitrary value like 0 or -1 or empty string to indicate null because we would prevent our users from storing a legitimate data value in our table.

To solve this, we will add a `LongInt` to the end of each record; each bit in this region corresponds to a field in the record. If the bit is set, then the field is null. If the bit is not set then the field is not null. Our dataset code will have to take these null flags into account when reading and writing data to the record.

Notice that our 4 bytes of null flags impose an arbitrary limit of 32 fields per record. In reality, you'd have to come up with a more flexible storage scheme, but this serves our purposes for now and keeps the material simple.

Calculated & Lookup Fields

Now that we have an actual `TDataSet` descendant component which we can drop on a form, we automatically inherit the Fields Editor property editor.

The Fields Editor allows us to select a subset of fields from the physical record, rearrange their order in the dataset, and even set certain display attributes for the fields. We can also add entirely new fields to the dataset by defining calculated fields, which are assigned values through the `TDataSet.OnCalcRecord` event-handler. Obviously, we're going to need some place to store the data values for the calculated fields. We set aside a region in the record buffer immediately after the physical record to contain all the calculated fields. Lookup fields are a special form of calculated field and their result values are also stored in this area. Internally, there is no difference in the representation of a lookup field and a straight calculated field.

To support calculated fields, we must override the `ClearCalcFields` method and modify our `GetRecord` method to call `GetCalcFields` to write the field data in the record buffer. Both of these modifications are shown in Listing 3.

External Field Definitions

To support Delphi's `TField` components, we're going to need some definition of fields bound externally with the table itself. To accomplish this, we'll add a dictionary file to go with our data file. Our dictionary will be a simple text file, having the same name as the data file with the extension DIC.

Listing 4 shows the Delphi record definition for our data file (this is how we accessed the fields in the record last month) and also shows the dictionary file contents for the same file.

Each line in the dictionary describes a field in the table and each element of the field description is delimited by a comma. The field name appears first, followed by the field datatype, followed by an optional field size, followed by an optional NULL keyword which tells us whether the field requires a value or can be set to null. The field size in this case only applies to string fields, since the sizes of the other datatypes are fixed by definition (see the help on TFieldDef.Size for other Delphi field types to which Size applies).

Notice that the DelFlag and NullFlags fields are not defined in the DIC file. That's because these fields represent implementation data in the physical record used to manage the table. When DelFlag is nonzero, it denotes a deleted record. The NullFlags field marks null values for the fields in the record. They are not record *data* but record *overhead*. Neither of these fields should be directly accessible to the user of the data. The TTestRec definition describes the physical record structure, while the DIC file definition describes the user-accessible data.

Within TDataSet, there are two collections of fields to be concerned with. The TDataSet.FieldDefs property defines the actual field structure of the table, regardless of how fields are rearranged in the Fields Editor. The TDataSet.Fields property defines the persistent fields setup via the Fields Editor. If no persistent fields have been defined, then Fields is filled from, and matches, FieldDefs.

Our first order of business is to populate FieldDefs with the actual field structure of the table when the table is opened. To do this we override the TDataSet.InternalInitFieldDefs method as shown in Listing 5.

```
procedure TMyDataSet.ClearCalcFields(Buffer: PChar);
begin
  FillChar(Buffer[FCalcFieldsOffset], CalcFieldsSize, 0);
end;
function TMyDataSet.GetRecord(Buffer: PChar; GetMode: TGetMode;
  DoCheck: Boolean): TGetResult;
begin
  {... lines omitted}
  if Result = grOk then begin
    GetCalcFields(Buffer);
    with PExtraRecInfo(Buffer + FExtraRecInfoOffset)^ do begin
      RecordNumber := (FilePos(FInternalFile) div FRecSize) - 1;
      BookmarkFlag := bfCurrent;
      SetBookmarkData(Buffer, @RecordNumber);
    end;
  end;
end;
```

➤ *Listing 3*

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


DIC File Contents:
EmpNo      ,SMALLINT
FirstName  ,STRING     ,15    ,NULL
LastName   ,STRING     ,20
HireDate   ,DATETIME   ,      ,NULL
DeptNo     ,STRING     ,3     ,NULL
Salary     ,DOUBLE
```

➤ *Listing 4*

In the body of this method, we simply open the DIC file and loop through all the records there, building up the FieldDefs structure as we go along. The local procedure ParseDictRec handles interpreting the dictionary record. It first breaks apart the record into its constituent attributes with the GetNextAttribute local function. GetNextAttribute simply parses a comma-separated list of items in a string and returns the leading item in one string and the remaining list in another string. If optional trailing attributes are omitted (like the size attribute), then GetNextAttribute simply returns empty strings.

To create a TFieldDef instance, we need to specify the field number, a datatype from TFieldType, the field's size, and whether a value is required for the field (nulls not allowed). The remainder of ParseDictRec takes the information from the dictionary record and populates the variables FieldName, DataType, Size, and Required as appropriate for the current field. Note that even if the field size is provided in the dictionary, we ignore it except for those datatypes to which it is relevant; in our case only the string fields. We also set ActualSize in all cases to the correct size of the physical field so that we may accumulate the total size of the physical record and compute field offsets. Notice that in the case of strings, Size is the number of characters that may be stored in the string and ActualSize is one byte longer to account for the actual physical storage space consumed by the string.

Back in the main body of InternalInitFieldDef, we use the information set by ParseDictRec to create an instance of TFieldDef and bind it to the TDataSet.FieldDefs property. We also calculate the total size of the physical record as we go in FRecSize, remembering to skip the 1-byte "deleted flag" field at the start of the record and to account for the 4-byte "null flags" field at the end. Notice that before we add the current field's size to FRecSize, it conveniently contains the offset value to the start of the current field in the record buffer. We'll need this information later on when we read and write data from

```
procedure TMyDataSet.InternalInitFieldDefs;
var
  DictFile: TextFile;
  DictRec: ShortString;
  FieldNo: Integer;
  FieldName: ShortString;
  Required: Boolean;
  DataType: TFieldType;
  Size: Word;
  ActualSize: Word;
  procedure GetNextAttribute(Rec: ShortString;
    var Attribute, OutRec: ShortString);
  var I: Integer;
  begin
    I := 1;
    Attribute := '';
    OutRec := '';
    if Rec = '' then Exit;
    while (I <= Length(Rec)) and (Rec[I] <> ',') do begin
      if not (Rec[I] in [' ', #9]) then
        Attribute := Attribute + Rec[I];
      Inc(I);
    end;
    if I < Length(Rec) then
      OutRec := Copy(Rec, I + 1, Length(Rec));
  end;
  procedure ParseDictRec;
  var
    DataTypeStr: ShortString;
    TempSize: Integer;
    Attribute: ShortString;
  begin
    { Get field name }
    GetNextAttribute(DictRec, FieldName, DictRec);
    if FieldName = '' then Exit;
    { Get data type }
    GetNextAttribute(DictRec, DataTypeStr, DictRec);
    { Get size }
    GetNextAttribute(DictRec, Attribute, DictRec);
    TempSize := 0;
    if Attribute <> '' then TempSize := StrToInt(Attribute);
    { Get null/not null }
    GetNextAttribute(DictRec, Attribute, DictRec);
    Attribute := Uppercase(Attribute);
    Required := Attribute <> 'NULL';
    Size := 0;
    ActualSize := 0;
    DataTypeStr := Uppercase(DataTypeStr);
    if DataTypeStr = 'SMALLINT' then begin
      DataType := ftSmallInt;
      ActualSize := SizeOf(SmallInt);
    end else if DataTypeStr = 'INTEGER' then begin
      DataType := ftInteger;
      ActualSize := SizeOf(Integer);
    end else if DataTypeStr = 'WORD' then begin
      DataType := ftWord;
      ActualSize := SizeOf(Word);
    end else if DataTypeStr = 'SINGLE' then begin
      DataType := ftFloat;
      ActualSize := SizeOf(Single);
    end else if DataTypeStr = 'DOUBLE' then begin
      DataType := ftFloat;
      ActualSize := SizeOf(Double);
    end else if DataTypeStr = 'STRING' then begin
      DataType := ftString;
      ActualSize := TempSize + 1;
      Size := TempSize;
    end else if DataTypeStr = 'DATETIME' then begin
      DataType := ftDateTime;
      ActualSize := SizeOf(TDateTime);
    end else
      DataType := ftUnknown;
  end;
begin
  FieldDefs.Clear;
  AssignFile(DictFile, ChangeFileExt(FTableName, '.DIC'));
  Reset(DictFile);
  try
    FRecSize := 1;  {skip the delete flag field}
    FieldNo := 0;
    while not System.Eof(DictFile) do begin
      ReadLn(DictFile, DictRec);
      Inc(FieldNo);
      ParseDictRec;
      if FieldName <> '' then begin
        FieldOffsets.Add(Pointer(FRecSize));
        { store field offset }
        Inc(FRecSize, ActualSize);
        { compute our record size }
        TFieldDef.Create(FieldDefs, FieldName, DataType,
          Size, Required, FieldNo);
      end;
    end;
    FNullFlagsOffset := FRecSize;
    Inc(FRecSize, SizeOf(LongInt));
    { Record size includes null flags space }
  finally
    CloseFile(DictFile);
  end;
end;
```

➤ *Listing 5*

the record buffer so we keep a list of field offsets in our own FieldOffsets list. Many database APIs reference fields by field number rather than offset, so whether you need to keep track of field offsets will depend on your database API.

Once we've handled the last field, FRecSize serves as an offset to the start of the "null flags" field at the end of the record. So we tuck this info away in the FNullFlagsOffset variable and increment FRecSize to account for the size of the "null flags" field.

Now all we need to do is make sure InternalInitFieldDefs gets called properly from our custom dataset component. Delphi calls this method automatically in design-mode when the Fields Editor is invoked, but we have to call it ourselves when the data file is opened in InternalOpen. Shown in Listing 6 is how we've changed InternalOpen and InternalClose since last month.

Remember that in InternalInitFieldDefs we only populate FieldDefs, not Fields. TDataSet populates Fields for us when persistent fields are defined through the Fields Editor.

Our only concern is when no persistent fields have been defined. In that case, the TDataSet property DefaultFields is true and we call TDataSet.CreateFields to have the physical field definitions created in Fields.

Next, we call the TDataSet.BindFields method which scans the fields and does some initialization work for calculated, lookup, and BLOB fields in the dataset. At this point, the size of the space reserved in the record buffer for calculated/lookup fields is known and can be retrieve through the

➤ *Listing 6*

```
procedure TMyDataSet.InternalOpen;
begin
  AssignFile(FInternalFile, FTableName);
  Reset(FInternalFile, 1); { Open a file of bytes }
  FCursorOpen := True;
  InternalInitFieldDefs;              { Populate FieldDefs from external dict }
  if DefaultFields then CreateFields;        { Populate Fields from FieldDefs }
  BindFields(True);
  BookmarkSize := SizeOf(TBookmarkInfo);
  { Compute offsets to various record buffer segments }
  FCalcFieldsOffset := FRecSize;
  FExtraRecInfoOffset := FCalcFieldsOffset + CalcFieldsSize;
  FBookmarkOffset := FExtraRecInfoOffset + SizeOf(TExtraRecInfo);
  FRecBufSize := FBookmarkOffset + BookmarkSize;
end;
procedure TMyDataSet.InternalClose;
begin
  { Destroy the TField components if no persistent fields }
  if DefaultFields then DestroyFields;
  { InternalClose is called by the Fields Editor in design mode, so
    the actual table may not be open. }
  if FCursorOpen then CloseFile(FInternalFile);
  FCursorOpen := False;
end;
```

TDataSet.CalcFieldsSize property. Now all we do is calculate the offsets to the various record buffer segments and, finally, the total size of the record buffer.

Writing Field Data

When we were modifying data last month, we simply pulled the entire physical record buffer into our application, changed data directly in the buffer, then posted the whole buffer back to the table. Now that we have field definitions set up, we're going to have to support data access via the TField components as well. We do this through TDataSet's GetFieldData and SetFieldData methods, which are called directly from the TField.GetData and TField.SetData methods. So anytime a field is read from or written to, we handle the I/O with the record buffer through these two methods.

Listing 7 shows our implementation for SetFieldData. A special consideration must be made for date/time fields which we will discuss after covering the general premise of setting field data in the record buffer.

TDataSet passes in the TField component for the field being modified and a pointer to the buffer containing the new field data. If the desire is to set this field to null, then the buffer pointer is nil. There are several factors to consider when manipulating field data. Is the field part of the physical record? Is it a calculated field? Is it set to null? Depending on the answers to these questions we may need to reference a different segment of the record buffer.

Calculated/lookup fields are identified by having a field number of -1. If we know we have a physical data field, we retrieve its offset from the FieldOffsets list we built in InternalInitFieldDef.

If we are setting the field value to null, then we erase the existing field data and set the appropriate null flag. Erasing the actual field data is not actually necessary but keeps things tidy.

If we are writing actual data in the field, then we simply copy the data from the field buffer into the record buffer at the appropriate offset. Notice we have a special case for string fields. TField handles strings as null-terminated strings, so we have to do a conversion since we are storing normal Pascal length-byte strings. Then we clear the null flag for the field (in case it was null before). Finally we post a field change data event for the dataset, but only for a change in physical fields.

If the field being modified is a calculated field, then it's offset from the start of the calculated fields segment of the record buffer is stored within the TField component. Space for a calculated field is reserved in the calculated fields segment based on the size required for its datatype plus one byte for a null/not null flag. Since we're implementing the access to this field ourselves, we are free to place the one byte null flag either at the start or end of the field data. Since it is simpler to work with it at the start of the field, we'll keep it there.

When the OnCalcFields event-handler is invoked, the dataset's state changes to dsCalcFields. In this mode, we cannot allow regular fields to be written to (that is, regular fields cannot be assigned values within OnCalcFields). Also, the dataset's ActiveBuffer does not necessarily point to the buffer containing the fields we are calculating, so we must ensure that we are using TDataSet.CalcBuffer,

➤ *Listing 7*

```
procedure TMyDataSet.SetFieldData(Field: TField; Buffer: Pointer);
var
  Offset,
  DataSize: Integer;
  StrBuff: ShortString;
  NullFlags: ^LongInt;
  TimeStamp: TTimeStamp; { TTimeStamp is declared in SysUtils }
  DateTime: TDateTime;
begin
  if Field.FieldNo <> -1 then begin   { a physical field }
    { Cannot set fields while in OnCalcFields handler }
    if State = dsCalcFields then DatabaseError(SNotEditing);
    Offset := LongInt(FieldOffsets[Field.FieldNo - 1]);
    DataSize := Field.DataSize;  {?? need this? }
    { Current null flags }
    NullFlags := @ActiveBuffer[FNullFlagsOffset];
    if not Assigned(Buffer) then begin
      { If setting field to null, clear the field data and set the null flag }
      FillChar(ActiveBuffer[Offset], DataSize, #0);
      NullFlags^ := NullFlags^ or (1 shl (Field.FieldNo - 1));
    end else begin
      { Special handing for date/time fields }
      if Field.DataType in [ftDateTime, ftDate, ftTime] then begin
        case Field.DataType of
          ftDate:
            begin
              TimeStamp.Time := 0;
              TimeStamp.Date := TDateTimeRec(Buffer^).Date;
            end;
          ftTime:
            begin
              TimeStamp.Time := TDateTimeRec(Buffer^).Time;
              TimeStamp.Date := DateDelta;
            end;
        else
          try
            TimeStamp := MSecsToTimeStamp(TDateTimeRec(Buffer^).DateTime);
          except
            TimeStamp.Time := 0;
            TimeStamp.Date := 0;
          end;
        end;
        DateTime := TimeStampToDateTime(TimeStamp);
        Move(DateTime, ActiveBuffer[Offset], SizeOf(TDateTime));
      end else if Field.DataType = ftString then begin
        StrBuff := StrPas(Buffer);
        Move(StrBuff, ActiveBuffer[Offset], DataSize);
      end else
        Move(Buffer^, ActiveBuffer[Offset], DataSize);
      { Set flag to nonnull }
      NullFlags^ := NullFlags^ and not (1 shl (Field.FieldNo - 1));
    end;
  end else begin   { a calculated field }
    Offset := FCalcFieldsOffset + Field.Offset;
    Boolean(CalcBuffer[0]) := not Assigned(Buffer);
    if Assigned(Buffer) then begin
      Move(Buffer^, CalcBuffer[Offset + 1], Field.DataSize);
    end;
  end;
   if not (State in [dsCalcFields]) then
     DataEvent(deFieldChange, Longint(Field));
end;
```

which is setup for us automatically, to write calculated field values.

If the calculated field is being set to null, we store a boolean True in the null flag (again, we can store anything we want). Otherwise, we store a boolean False in the null flag and copy the data from the field buffer into the record buffer, remembering to add one to the offset to skip over our null flag.

Date/Time Fields

When a datetime field is accessed through a TDateTimeField component, Delphi does not expect the raw record buffer to contain data formatted for a Delphi TDateTime variable. Instead, it expects the raw field data to conform to the TDateTimeRec type defined in the DB unit and shown in Listing 8. Basically this format requires that the time value represent the number of milliseconds since midnight and the date value represent the number of days since December 31st, 0000 (that is, a date value of 1 means January 1st, 0001).

Since our raw datetime data is a true Delphi TDateTime variable, we must translate the TDateTimeRec data given to us by TDateTimeField. This seems silly since the data originated as a TDateTime in the first place and that's what we want to store in our file, but we can't override the behavior of TDateTimeField, so we must make this translation. Fortunately, Delphi provides a number of routines in the SysUtils unit to help us convert the values.

Listing 7 shows the details, which are based on the actual translation code in TDateTimeField.GetValue.

Reading Field Data

When field data is requested through a TField component, the call ends up in TDataSet.GetFieldData, an abstract method which we must override. GetFieldData works similarly to SetFieldData.

The TField component for the field of interest is passed in along with a pointer to a buffer where the field data should be written. The GetFieldData function returns True if field data was returned or False if the field value is null. Sometimes GetFieldData is called with a nil value for the buffer pointer. This happens when TField is interrogating the null status of the field (for example in response to the IsNull property) and is not interested in the actual field data.

Again, when OnCalcFields is active, the dataset state is dsCalcFields and we must read all data from the CalcBuffer property rather than the ActiveBuffer property. This includes reading values from physical fields since they will generally be used to compute new values and you must ensure that the physical field data and the calculated field data all come from the same buffer.

With this in mind, and having already gone through the details of SetFieldData, the implementation

➤ *Listing 8*

```
type
  TDateTimeRec = record
    case TFieldType of
      ftDate: (Date: Longint);
      ftTime: (Time: Longint);
      ftDateTime: (DateTime: TDateTime);
  end;
```

➤ *Listing 9*

```
function TMyDataSet.GetFieldData(Field: TField; Buffer: Pointer): Boolean;
{ Get the data for the given field from the active buffer and stick it in given
  buffer.  Return False if the field value is null; otherwise return True. Buffer
  may be nil if TDataSet is checking for null only. }
var
  Offset,
  DataSize: Integer;
  NullFlags: ^LongInt;
  TimeStamp: TTimeStamp;  { TTimeStamp declared in SysUtils }
  DateTime: TDateTime;
  RecBuf: PChar;
begin
  RecBuf := ActiveBuffer;
  if State = dsCalcFields then
    RecBuf := CalcBuffer;
  if Field.FieldNo <> -1 then begin   { a physical field }
    { Check for a null value }
    NullFlags := @RecBuf[FNullFlagsOffset];
    Result := ((NullFlags^ and (1 shl (Field.FieldNo - 1))) = 0);
    { If value is not null }
    if Result and Assigned(Buffer) then begin
      FillChar(Buffer^, Field.DataSize, 0);
      Offset := LongInt(FieldOffsets[Field.FieldNo - 1]);
      DataSize := Field.DataSize;
      { Special handing for date/time fields }
      if Field.DataType in [ftDateTime, ftDate, ftTime] then begin
        Move(RecBuf[Offset], DateTime, DataSize);
        TimeStamp := DateTimeToTimeStamp(DateTime);
        case Field.DataType of
          ftDate: TDateTimeRec(Buffer^).Date := TimeStamp.Date;
          ftTime: TDateTimeRec(Buffer^).Time := TimeStamp.Time;
        else
          TDateTimeRec(Buffer^).DateTime := TimeStampToMSecs(TimeStamp);
        end;
      end else begin
        if Field.DataType = ftString then begin
          DataSize := Byte(RecBuf[Offset]);
          Inc(Offset);
        end;
        Move(RecBuf[Offset], Buffer^, DataSize);
      end;
    end;
  end else begin   { a calculated field }
    Offset := FCalcFieldsOffset + Field.Offset;
    Result := not Boolean(RecBuf[Offset]);
    if Result and Assigned(Buffer) then begin
      Move(RecBuf[Offset + 1], Buffer^, Field.DataSize);
    end;
  end;
end;
```

➤ *Listing 10*

```
procedure TMyDataSet.InternalAddRecord(Buffer: Pointer; Append: Boolean);
begin
  Byte(Buffer^) := 0;  { reset deleted flag as a precaution }
  Seek(FInternalFile, FileSize(FInternalFile));
  BlockWrite(FInternalFile, Buffer^, FRecSize);
end;
```

of GetFieldData shown in Listing 9 should be intuitive.

InsertRecord/AppendRecord

TDataSet supports two public methods, InsertRecord and AppendRecord, which fill all the fields for a new record and add that record to the table.

To support these methods, we must override InternalAddRecord. This method passes in a pointer to the record buffer containing the field data and a boolean flag indicating whether it should be inserted or appended. In our case, there is no difference between inserting or appending a record; either way the record is actually appended to the end of the table.

Listing 10 shows the details of our implementation.

Refresh

When the public method Refresh is called, TDataSet calls InternalRefresh to give us the opportunity to refresh our database driver buffers (ie, the Borland Database Engine's DbiForceReread call) before it refreshes it's internal record buffers. Since we are not buffering our data in any way outside of TDataSet's internal record buffers, we don't need to do anything in InternalRefresh.

RecordCount

You may have noticed a bug in the RecordCount property as implemented last month. We simply retrieved the total number of bytes in the file and divided by the number of bytes in one record to yield the number of records in the file.

Well, remember that when we delete records, we only set a flag in the record and leave the physical record in the file. Our RecordCount technique misreports total records because it includes all the deleted records as well. Ideally we would create a header area at the top of the file and store a field that keeps track of the total number of nondeleted records in the table. I won't show the implementation for that because it is specific to our sample datafile and not worth taking up the space with here.

Conclusion

That sums up our custom TDataSet decendant. We now have all the significant functionality needed to manipulate data through our own TMyTable component, including binding data-aware controls to our data.

There are a few more odds and ends, such as handling BLOB fields, indexes and so forth. Some of these things are not inherently a part of TDataSet but are in fact TTable functions. Note that although TDataSet does contain methods and properties to support filters, they actually do nothing. Filters are fundamentally bound to the BDE and if you want them in your custom datasets, you'll have to parse the filter text and implement the record scanning from scratch.

In any case the foundation laid here, plus a good look at the source code for Borland's TBDEDataSet, TDBDataSet, and TTable classes will get you well on your way to making your own full-fledged custom datasets.

Next month we'll take a look at the new TDecisionCube component and all it's sister components. What can they do for us? How to we get them to do it? And what will they do to my performance? See you next month.

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.