Writing Custom Datasets

By Gerald Nunn
GExperts Inc, Copyright 2000

Introduction

The release of Delphi 3 introduced one of the most powerful features available in Delphi, the ability to create custom datasets. Prior to Delphi 3, developers had to jump through hoops to get the data aware controls to work with anything other then the BDE. In Delphi 3, Borland wisely separated the TDataset component from the BDE thus enabling anyone to write custom data access components.

Unfortunately, writing TDataset descendants has a reputation of being difficult, largely due to three facts: it's undocumented, it often involves pointers and memory manipulation, and finally there are a lot of methods in the TDataset class to override.

Thus the purpose of this lecture is two fold. The first purpose, and the obvious one, is to learn how to write a custom dataset. The second purpose is to write a TDataset descendant that can be reused as an ancestor for other datasets and handles all of the day to day drudgery involved with writing datasets.

I'm going to be honest here and admit that I'm foisting a bit of an experiment on my audience. Typically at these sort of presentations you usually get a simple example that seems easy to understand but doesn't go nearly far enough in explaining how to implement it beyond the simple example. I'm going to do the opposite.

This is the experiment. The base dataset, called TGXBaseDataset, that we are going to create is actually pretty complicated since it exposes a lot of features of the TDataset class. However, having a complicated intermediary base class in TGXBaseDataset greatly reduces the complexity of the Outlook dataset that is based on TGXBaseDataset. I'm hoping this will bridge both extremes by providing a simple entry point to exploring the world of custom datasets while giving a sufficiently complicated example to push the envelope when you are ready to do so.

I will only know if this experiment has been successful through you, the audience. Feel free to send me comments and suggestions to gnunn@gexperts.com.

Overview of Writing a Custom TDataset

Normally when you talk to people about writing custom datasets the first thing that pops into their mind is databases. However, the fact is that writing a custom dataset can be a powerful tool for displaying data from all sorts of sources. For example, how many times have you had to fill a listview with information from a collection? A reusable alternative would be to write a custom dataset that accesses information in a collection and enables you to display the information in a grid.

Writing custom datasets is about leveraging RAD to the maximum extent possible. It is about writing code once instead of many times. It is about easily moving information between visual controls and containers of information in an easy and efficient manner.

So now that we want to write one, how do we do it? Writing a custom dataset involves overriding a variety of abstract methods of the TDataset class. Typically, the more functionality that the custom dataset exposes the more methods you need to override. At a minimum, you will usually need to override the following methods:

AllocRecordBuffer	InternalAddRecord	InternalInitFieldDefs	IsCursorOpen
FreeRecordBuffer	InternalClose	InternalInitRecord	SetBookmarkFlag
GetBookmarkData	InternalDelete	InternalLast	SetBookmarkData
GetBookmarkFlag	InternalFirst	InternalOpen	SetFieldData
GetRecord	InternalGotoBookmark	InternalPost	SetFieldData
GetRecordSize	InternalHandleException	InternalSetToRecord	

The majority of these methods fall into one of two categories, navigation management and buffer management. Navigation management is fairly self explanatory, these are the methods that are called in response to the dataset being navigated through first, prior, next, etc. Methods that fall into this category include InternalFirst, InternalLast and GetRecord.

Buffer management is a little more complicated to explain. In order to accomplish it's work, the TDataset manages a set of buffers. These buffers are used by the TDataset as both a cache and a means of temporarily storing information during the editing process. As a user modifies fields during the edit process, the changes the user makes are stored into one of these record buffers, not the underlying data source. If the user elects to cancel the changes, the TDataset retrieves a fresh copy of the buffer to revert the buffer back to it's original state. If the user posts the changes, then the changes are copied from the buffer into the underlying data source.

Methods that involve buffer management include AllocRecordBuffer, FreeRecordBuffer, GetRecord, GetRecordSize, GetFieldData, SetFieldData and InternalPost. Note that GetRecord is a bit unusual in that it is both a navigation and buffer management method.

It's important to realize that the TDataset class itself does not care how the buffer is structured. The buffer could be a pointer to an object, a mapped memory block, a variant list, etc, it doesn't matter to the TDataset. This gives us a lot of freedom in how we design our TDataset descendants.

A record buffer will typically be composed of three pieces of information:

a. Navigational information. Each buffer must be able to hold a bookmark flag and a unique identifier that links the buffer to the corresponding row in the dataset. Additionally, it can also contain bookmark information. If you don't provide a mechanism to identify a buffer from a row, your dataset will not work with grids.

b. Field values. The buffer must contain a copy of the field values for the row it represents.
c. Calculated field values. The buffer must contain an area that is used to store values for calculated fields.

Generally speaking, when it comes to buffer management and creating a custom TDataset, one of two scenarios will prevail. In the first case, you are writing a custom dataset for something that already supports the concept of record buffers. In this case, managing record buffers within the custom dataset is merely a case of placing a thin veneer on the facility already available. Examples of this case include writing a dataset to wrap the BDE or writing a dataset that provides native access to Interbase. Both the BDE and the Interbase API provide an easy mechanism for creating and maintaining a buffer based on a given cursor.

The other scenario is when we are creating a custom dataset to wrap something that doesn't provide any help in creating and managing record buffers. In this case, the custom dataset will have to be responsible for devising an appropriate buffer scheme and managing them within the context of the TDataset environment.

It is this second scenario that we will be examining in this presentation. In this second scenario, an easy way to manage the record buffer is as a single chunk of memory organized as follows:

Field Values	Navigational Information	Calculated Fields

Most dataset examples you will see layout their buffers in this way, including the TextData demo shipped with Delphi.

Having to work with these record buffers is probably one of the largest reasons why people have difficulty in creating custom datasets. It can get quite complicated managing these buffers and fetching and storing information in each buffer as needed. Many programmers are turned off my having to manipulate pointers and move memory.

In the next part of this presentation, we will examine the various dataset methods that need to be overridden in descendant classes and what the TDataset class expects these methods to do.

TDataset Methods

Allocating and freeing record buffers

TDataset handles the allocation and disposal of record buffers through two methods, *AllocRecordBuffer* and *FreeRecordBuffer*. Descendant classes override these methods and create an appropriate buffer. Remember that the TDataset class does not care about how the buffer itself is structured. The buffer could be something as simple as a pointer to an object or as complex as a block of memory containing a mapped layout of all fields.

In terms of memory management, you are guaranteed that buffers will only be created and freed through the aforementioned methods and that the TDataset class will never alter the internal contents of the buffer on it's own. Thus it is safe to stuff pointers and other things into the buffer without worrying about generating memory leaks.

GetRecord

This is probably the single most important method that descendant TDataset classes must override. GetRecord is a combination navigation and data retrieval method that is prototyped as follows:

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

This function is called with three parameters. The GetMode parameter tells the dataset to perform a navigation operation and can be one of three values as follows:

a. *gmCurrent*: This tells the descendant class to return the current record in the buffer.
b. *gmNext*: This tells the descendant class to return the next record in the buffer. If there is no next record, *grEof* must be returned as the Result.
c. *gmPrior:* This tells the descendant class to return the previous record in the buffer. If there is no prior record, *grBOF* must be returned.

As is obvious from the preceding paragraph, since GetRecord also performs a navigational function, descendant classes must be able to pass back whether or not the navigational operation succeeded. For example, if gmNext is passed as the GetMode but the dataset is positioned on the last record, we must be able to pass back the fact that we can't go to the next record. This is done through the result of the function which is of the type TGetResult and can be one of the following values:

a. *grEof*: This is returned when the GetMode is gmNext but we are already on the last record.
b. *grBof*. This is returned when the GetMode is gmPrior but we are already on the first record.
c. *grOK*. The operation specified by GetMode was performed successfully and the buffer parameter is valid.
d. *grError*. An unexpected error occurred.

This function passes in a buffer variable which the descendant class must fill in with the current field values. How these field values are mapped into the buffer doesn't matter, but the field values must be there since the values in this buffer are what is used during edit operations. Note that this button only needs to be filled in when the result of the function is *grOK*.

The DoCheck parameter, if true, tells the database to raise an exception if an error has occurred. This usually only occurs during filtering operations.

<u>GetFieldData and SetFieldData</u>

As we have discussed, the TDataset class doesn't know anything about how our record buffer is organized. The next question then becomes that if the TDataset class doesn't know about our record structure, how it does it pull out field values from this structure? How it does it put them back in when the user is editing a record?

The answer is that it uses the GetFieldData and SetFieldData methods which a custom dataset class must override. These methods are called by the dataset when it needs to retrieve or set a specific field value from the current record buffer.
GetFieldData is prototyped as follows:

```
function TDataset.GetFieldData(Field: TField;
  Buffer: Pointer): Boolean;
```

The Field parameter is the field for which the value needs to be retrieved. The Field parameter is only passed for reference and should never be altered by this routine.

The Buffer parameter is where the field value needs to be copied to. Looking at the buffer parameter results in a question that doesn't have an obvious answer at first glance. That question is "What size is that buffer and what needs to be copied into it?". The only way of determining this is by looking at the various TField types in DB.pas and examining their GetValue and GetDataSize methods.

Here is a partial table with some values used in the base dataset we will create later on:

Field Type	Buffer Result
ftInteger,ftDate,ftTime	Integer
ftBoolean	Boolean
ftDateTime	TDateTimeRec
ftFloat,ftCurrency	Double
ftString	PChar

As we can see, most types map pretty cleanly with the noteable exception of TDateTime which requires some translation into a TDateTimeRec.

GetFieldData function returns True if a value was copied into the buffer by the method and False if no value was copied.

That covers the GetFieldData method. SetFieldData is the exact reverse operation of GetFieldData. It is passed a buffer with some field value in the buffer that must then be copied back into your record buffer. Note that if your dataset is a readonly dataset, then there is no need to implement SetFieldData. SetFieldData is prototyped as follows:

```
procedure TDataset.SetFieldData(Field: TField; Buffer: Pointer);
```

In the case of SetFieldData, the Field parameter is once again the field whose value needs to be copied into the record buffer. The buffer parameter contains the actual value to be copied. Once again, the size and nature of the Buffer will vary depending on the field type but is identical to the buffer type used for each field type in GetFieldData.

GetRecordSize

GetRecordSize returns the size of the record buffer that the TDataset is managing. Descendants override this function and return the size of the buffer they will be allocating in *AllocRecordBuffer*. Note that all buffers must be sized identically, you cannot allocate one buffer of twenty bytes and then allocate the next buffer at forty bytes. GetRecordSize is prototyped as follows:

```
function TDataset.GetRecordSize: Word;
```

GetCanModify

This function determines whether or not the dataset is read only and is prototyped as follows:

```
function TDataset.GetCanModify: Boolean;
```

Return *True* if the dataset can be modified and *False* if it is readonly.

InternalOpen

Descendant datasets override InternalOpen and perform the necessary logic to open the underlying datasource. Additionally several standard steps are also performed in this method such as initializing the field definitions, creating the default fields if required and binding the fields. Here is a sample InternalOpen method from TGXBaseDataset.

```
procedure TGXBaseDataset.InternalOpen;
begin
  if DoOpen then
    Begin
    BookmarkSize:=GetBookMarkSize;  //Bookmarks not supported
    InternalInitFieldDefs;
    if DefaultFields then CreateFields;
    BindFields(True);
    FisOpen:=True;
    FillBufferMap;
    End;
end;
```

Obviously, InternalOpen must be overrided in all descendant datasets.

InternalClose

InternalClose is called by the TDataset whenever the dataset is closed. Descendants should perform any necessary cleanup in this method. Note that no cleanup of buffers is required as the TDataset manages this automatically by calling *FreeRecordBuffer* as needed. Additionally, and default fields that were created should be destroyed and fields should be unbound by calling `BindFields(False)`.

InternalHandleException

This procedure is called by the TDataset when an exception occurs. Most of the time, you will simply call *Application.HandleException(Self)* in this procedure.

InternalInitRecord

This procedure is called whenever the dataset needs to reinitialize a buffer that has been previously allocated. It is prototyped as follows:

```
procedure TDataset.InternalInitRecord(Buffer: PChar);
```

This procedure passes one parameter, Buffer, which is the buffer that needs to be reinitialized. If your using a simple memory block for the buffer, it can be initialized using *FillChar(Buffer^,GetRecordSize,0)* which zeroes out all information in the buffer.

Bookmark Flags

Bookmark flags are used by the dataset to store row positioning information within the dataset. Bookmark flags can be on of the following values: *bfCurrent, bfBOF, bfEOF, bfInserted*. In order to handle bookmark flags correctly, a custom dataset must support reading and writing this flag to a specified record buffer. This is done by overriding the methods GetBookMarkFlag and SetBookMarkFlag which are prototyped as follows:

```
function TDataset.GetBookmarkFlag(Buffer: PChar): TBookmarkFlag;
procedure TDataset.SetBookmarkFlag(Buffer: PChar;
  Value: TBookmarkFlag);
```

BookMark Data

The bookmark data methods are used by the dataset to support, surprise, bookmarks. The way bookmarks work within the dataset is that a custom dataset specifies the size of the bookmark. The dataset then takes care of allocating this space and passes a pointer to the GetBookMarkData whenever it needs to fetch a copy of the bookmark data for a given record buffer. GetBookMarkData is prototyped as follows:

```
procedure TDataset.GetBookmarkData(Buffer: PChar; Data: Pointer);
```

The Data parameter is where you copy the bookmark data into. Remember that the dataset has already allocated the space required to stuff the bookmark data into the Data parameter.

The method SetBookMarkData is called when the bookmark needs to be stored into a specified record buffer by the dataset. You should copy the bookmark from the Data parameter into the appropriate location in the record buffer.

InternalSetToRecord

As mentioned in the overview of record buffers, each buffer must have an identifier which enables the dataset to be positioned on that buffer based on that identifier. The dataset performs this positioning operation by calling InternalSetToRecord which is prototyped as follows:

```
procedure TDataset.InternalSetToRecord(Buffer: PChar);
```

This function is passed a single parameter, Buffer. Based on the buffer passed to InternalSetToRecord, the internal cursor of the dataset must be set to whatever record this buffer represents.

IsCursorOpen

This function enables the dataset to determine whether or not data is available while the dataset is being opened, even if the state is dsInactive. It takes no parameters and returns a Boolean. Return *True* if data is available or *False* if it isn't.

InternalFirst

The InternalFirst procedure positions the dataset on the first row, descendants should override this method and take appropriate action.

InternalLast

The InternalLast procedure positions the dataset on the first row, descendants should override this method and take appropriate action.

InternalPost

This method is called whenever the user posts the current record. It is called for both inserts and edits. A TDataset descendant must copy the data from the current record buffer into the underlying datasource in this method. If this method is being called with an insert operation pending, the custom dataset is responsible for inserting a new record prior to saving the buffer contents.

Writing a Generic TDataset Descendant

Introduction

In the previous section we covered what is required to create a custom dataset. In this section we are going to put this information to good use and create a generic TDataset descendant that will greatly simplify writing custom datasets.

The intent of creating this custom dataset is to automate the whole process of having to manually manage record buffers in those scenarios where no buffer management facilities are available. The key to doing this is to realize that there is a finite number of field types defined in DB.pas and thus it is quite possible to create a generic record buffer system that saves and loads field values to the buffer based on the field type of each individual field.

After having written numerous TDataset descendants, I’ve come to the conclusion that buffer management is where you spend most of your time debugging the descendant and working to get it right. By creating a generic TDataset descendant that already includes a buffer management facility we will be able to greatly reduce the amount of work required to create custom datasets.

Our generic TDataset descendant, called TGXBaseDataset will therefore export a smaller and simpler set of abstract and virtual methods that must be overridden to create a dataset. In the first section of this part of the presentation we will cover the various TDataset methods that we override in TGXBaseDataset. In the second part of this presentation we will examine the TGXBaseDataset methods and how we use them to create a custom dataset by creating a dataset that accesses Outlook mail folders.

Creating TGXBaseDataset

Buffer Management

As mentioned in the introduction of this section, the point of the TGXBaseDataset is to automate the management of record buffers required by TDataset. Our generic buffer in the TGXBaseDataset will be a block of memory and will be laid out as below:

Field Values	Navigational Information	Calculated Fields

The navigational information is defined in a record structure as follows:

```
Type
  PRecordInfo = ^TRecordInfo;
  TRecordInfo=record
    RecordID: Pointer;
    BookMark: Pointer;
    BookMarkFlag: TBookmarkFlag;
End;
```

The Bookmark is what is used to uniquely identify the buffer and will enable the dataset to match a row to the buffer in the InternalSetToRecord method. The BookMarkFlag is defined in DB.pas and is used by the dataset to hold navigational information.

The size of the buffer required to hold the field values is determined in the method GetDataSize that is introduced in TGXBaseDataset.

```
function TGXBaseDataset.GetDataSize: Integer;
var Index: Integer;
Begin
  Result:=0;
  for Index:=0 to FieldCount-1 do
    case Fields[Index].DataType of
      ftString: Result:=Result+Fields[Index].Size+1;
                      //Leave space for terminating null
      ftInteger,ftSmallInt,ftDate,ftTime:
        Result:=Result+sizeof(Integer);
      ftFloat,ftCurrency,ftBCD,ftDateTime:
        Result:=Result+sizeof(Double);
      ftBoolean: Result:=Result+sizeof(Boolean);
      ftMemo,ftGraphic: Result:=Result+sizeof(Pointer);
    End;
End;
```

Examining the code above, we can see that we are simply cycling through the field definitions. We determine the size required to hold each field and then append that to the result. Once we have finished going through each field we will have the size required to hold the field values in the record buffer. One interesting thing to note in the code is that we are storing BLOB fields (ftMemo and ftGraphic) as pointers. Managing BLOB fields in custom datasets is covered in detail in Appendix 1 of this presentation.

This GetDataSize method is then used within the TDataset method GetRecordSize as follows:

```
function TGXBaseDataset.GetRecordSize: Word;
begin
  Result:=GetDataSize+sizeof(TRecordInfo)+CalcFieldsSize;
  FStartCalculated:=GetDataSize+sizeof(TRecordInfo);
end;
```

Thus the size of the record buffer is the size of the buffer needed to hold the field values (GetDataSize) plus the navigation record (TRecordInfo) plus the size needed to hold calculated field values (CalcFieldsSize). CalcFieldsSize is a method of TDataset. We also set a private variable called FStartCalculated that indicates where the start of the calculated fields buffer starts in the record buffer.

Buffers are allocated in the TDataset method AllocRecordBuffer which is implemented as follows in TGXBaseDataset:

```
function TGXBaseDataset.AllocRecordBuffer: PChar;
begin
  GetMem(Result,GetRecordSize);
  FillChar(Result^,GetRecordSize,0);
  AllocateBlobPointers(Result);
end;
```

This is quite straight forward as we are basically allocating an amount of memory determined by GetRecordSize using the Delphi GetMem procedure. See Appendix 1 for information on *AllocateBlobPointers*.

Now that we have allocated our buffer, we need to be able to get information into this buffer. We do this by overriding the TDataset method GetRecord as follows:

```
function TGXBaseDataset.GetRecord(Buffer: PChar;
  GetMode: TGetMode; DoCheck: Boolean): TGetResult;
begin
  Result:=Navigate(GetMode);
  if (Result=grOk) then
    Begin
    RecordToBuffer(Buffer);
    ClearCalcFields(Buffer);
    GetCalcFields(Buffer);
    End
  else if (Result=grError) and DoCheck then
    DatabaseError('No Records');
end;
```

The first thing that this routine does is call the Navigate method. This is a new method introduced in TGXBaseDataset that is overridden by descendants. Descendants use this method to tell the TGXBaseDataset whether or not the navigation function specified in the GetMode parameter was successful. Navigate is prototyped as follows:

```
function Navigate(GetMode: TGetMode): TGetResult;
```

If the Navigate method returns grOK, the TGXBaseDescendant starts the process of filling the record buffer by calling it's RecordToBuffer method. RecordToBuffer performs two tasks as follows:

a. It fills in the navigational record, PRecordInfo, in the record buffer. It does this by calling the method AllocateRecordID which a descendant dataset overrides to return an ID.
b. It fills in the record buffer with the field values by calling GetFieldValue for each field in the dataset. GetFieldValue is prototyped as follows:

```
function GetFieldValue(Field: TField): Variant;
```

The whole point of this exercise is that a dataset descending from TGXBaseDataset need only override GetFieldValue and AllocateRecordID instead of having to worry about managing buffers. It's a lot easier to return a variant for the field data in GetFieldValue then it would be to manage the copying of the field value into the buffer manually. Also, there will be no need to override the TDataset GetFieldData and SetFieldData methods in TGXBaseDataset descendants since it already handles these methods for us automatically.

Let's take a moment to talk about AllocateRecordID. This method is introduced in TGXBaseDataset and is used to get a unique identifier for each record buffer. AllocateRecordID returns a pointer, however if the dataset you are creating uses record numbers then just cast the record number as a pointer and return that. If you need a more complicated identifier, you can return an object or something else as the pointer, however you may need to override DisposeRecordID to free this object or memory as required.

Getting back to record buffer operations, the TGXBaseDataset also has a method called BufferToRecord, which is the reverse operation of RecordToBuffer. It handles copying the field values from the record buffer back to the underlying data source. It does this by calling the method SetFieldValue for each field in the dataset. SetFieldValue is prototyped as follows:

```
procedure SetFieldValue(Field: TField; Value: Variant);
```

BufferToRecord passes the field for which the value needs to be saved and the actual value to be saved as a variant.

If your dataset allows inserts, it will need to override the DoBeforeSetFieldValue method. This method is called by the TGXBaseDataset just before the copying of fields from the buffer with SetFieldValue starts. It provides an opportunity for descendant datasets to make any special preparations before saving the field values. In the case of inserting, the special preparation would likely be inserting a new, blank row into the data source. DoBeforeSetFieldValue is prototyped as follows:

```
      procedure DoBeforeSetFieldValue(Inserting: Boolean);
```

If the dataset is in the process of inserting, then the Inserting parameter will be set to True.

Navigational methods

TGXBaseDataset introduces several methods to manage navigation through the dataset. These methods must be overridden by descendant datasets in order for navigation to be available.

DoFirst

This method moves to the first record in the dataset. It is functionaly equivalent to the InternalFirst method in the TDataset class.

DoLast

This method moves to the last record in the dataset. It is functionaly equivalent to the InternalLast method in the TDataset class.

Navigate

This method is called by the TGXBaseDataset in the GetRecord method and is used to provide the navigational capabilities of GetRecord that we discussed previously in the TDataset section. Descendants override this method and perform the navigation has specified by the GetMode parameter.

GotoRecordID

This method is passed the buffer identifier allocated in AllocateRecordID. Descendant datasets must position the dataset on whatever row that this buffer ID represents.

Creating the Outlook Dataset

Finally, after all that work and discussion we are at the stage of actually creating a practical dataset. For the purpose of this presentation, I have opted to create a dataset which displays information from outlook mail folders. Our TGXOutlookMail dataset will descend from the TGXBaseDataset class.

The first thing we need to do in creating our outlook dataset is to override the TGXBaseDataset method DoOpen as follows:

```
function TGXOutlookMail.DoOpen: Boolean;
begin
  FCurRec:=-1;
  FApp:=CoOutlookApplication.Create;
  if FApp<>nil then FNmSpace:=FApp.GetNameSpace('MAPI');
  if FNmSpace<>nil then
    FFolder:=FNmSpace.GetDefaultFolder(Folders[FOutlookFolder]);
  if FFolder<>nil then FItems:=FFolder.Items;
  Result:=(FItems<>nil);
end;
```

The outlook dataset will use record numbers, thus we represent the record number with the private variable FCurRec. FCurRec is initialized to –1. Next we get an instance of Outlook, the outlook namespace and the folder. Note that depending on how your copy of outlook is setup, you may need to change the code above to log into the outlook namespace.

The next step is to override the TGXBaseDataset method, DoClose. This appears as follows:

```
procedure TGXOutlookMail.DoClose;
begin
  FMailItem:=nil;
  FItems:=nil;
  FFolder:=nil;
  FNMSpace:=nil;
  FApp:=nil;
end;
```

All we are doing in this method is releasing the interfaces we hold to the various outlook objects.

The next thing that must be done is defining what fields will be available in our dataset. This is done by overriding the TGXBaseDataset method DoCreateFieldDefs. In our outlook dataset, we implement this as follows:

```
procedure TGXOutlookMail.DoCreateFieldDefs;
begin
  FieldDefs.Add('ENTRYID',ftString,40,False);
  FieldDefs.Add('SUBJECT',ftString,255,False);
  FieldDefs.Add('RECEIVEDTIME',ftDateTime,0,False);
  FieldDefs.Add('BODY',ftMemo,0,False);
end;
```

Once we have defined our fields, we must provide the means to get the field data into the buffer. This is accomplished by overriding GetFieldValue from the TGXBaseDataset.

```
function TGXOutlookMail.GetFieldValue(Field: TField): Variant;
begin
  if FMailItem=nil then exit;
  if Field.FieldName='RECEIVEDTIME' then
    Result:=FMailItem.ReceivedTime;
  if Field.FieldName='ENTRYID' then
    Result:=FMailItem.EntryID;
  if Field.FieldName='SUBJECT' then
    Result:=FMailItem.Subject;
end;
```

All this function is doing is checking what field is requested and copying the appropriate value from the mail item to the Result of the method. The variable FMailItem is actually set in DoBeforeGetFieldValue. While the FMailItem could be set in the GetFieldValue each time it is called, it is more efficient to do this once in DoBeforeGetFieldValue.

The next thing we need to do is handle the writing of the field values back to the mail item after the user has modified a record and posted the changes. To do this, we first need to override DoBeforeSetFieldValue to check to see if we are inserting a record, and if we are, create an appropriate mail item.

```
procedure TGXOutlookMail.DoBeforeSetFieldValue(
  Inserting: Boolean);
begin
  if Inserting then FMailItem:=FItems.Add(olMailItem) as MailItem
  else FMailItem:=FItems.Item(FCurRec+1) as MailItem;
end;
```

As we can see above, if the Inserting parameter is set to True, we create a new mail item otherwise we set the mail item to be whatever the current one is indicated by FCurRec. We can now move on to SetFieldValue which appears as follows:

```
procedure TGXOutlookMail.SetFieldValue(Field:TField;
  Value: Variant);
begin
  if FMailItem<>nil then
    Begin
    if Field.FieldName='SUBJECT' then
      FMailItem.Subject:=Value;
    End;
end;
```

Again this is a very simple method, as we are simply checking what the field name is and then assigning it to the appropriate property of the mail item. Note that the only editable non-blob property in our example is Subject as both EntryID and ReceivedTime are not editable in Outlook mail items.

We then override the DoAfterSetFieldValue method to save the mail item.

```
procedure TGXOutlookMail.DoAfterSetFieldValue(
  Inserting: Boolean);
var Index: Integer;
begin
  FMailItem.Save;
  if Inserting then
    Begin
    FMailItem:=FMailItem.Move(FFolder) as MailItem;
    Index:=EntryIDToIndex(FMailItem.EntryID);
    if Index>=1 then FCurRec:=Index-1;
    End;
  FMailItem:=nil;
end;
```

Finally we move on to the navigation methods which appear as follows:

```
procedure TGXOutlookMail.DoFirst;
Begin
  FCurRec:=-1;
End;

procedure TGXOutlookMail.DoLast;
Begin
  FCurRec:=RecordCount;
End;

function TGXOutlookMail.Navigate(GetMode: TGetMode): TGetResult;
begin
  if RecordCount<1 then
    Result := grEOF
  else
    begin
    Result:=grOK;
    case GetMode of
      gmNext:
        Begin
        if FCurRec>=RecordCount-1 then Result:=grEOF
        else Inc(FCurRec);
        End;
      gmPrior:
        Begin
        if FCurRec<=0 then
          Begin
          Result:=grBOF;
          FCurRec:=-1;
          End
        else Dec(FCurRec);
        End;
      gmCurrent:
        if (FCurRec < 0) or (FCurRec >= RecordCount) then
```

```
                Result := grError;
          End;
          End;
      end;
```

The methods DoFirst and DoLast are very simple and are self explanatory. The Navigate method is somewhat more complicated. When the navigate method is called, it is passed the GetMode parameter which indicates what navigate should do. GetMode will be gmNext, gmPrior or gmCurrent. Navigate then attempts to set the current row accordingly through FCurRec. If it can't, it returns a suitable response indicating what happened such as grEOF or grBOF. If everything is allright, it returns grOK.

Conclusion

In this presentation we have covered how to create a custom dataset. We have created a reusable base class which enables to perform this task very easily. We have then proved the concept by creating a simple dataset example to access Outlook mail items.

Interested parties can read Appendix 1 for information on how to handle BLOB fields.

Appendix 1
Handling BLOB fields in TDataset descendants

I have explicitly decided not cover BLOB fields in the main presentation. After starting to write this presentation it became obvious that it was going to be complicated and in an attempt to simplify things, I elected to move the discussion of handling BLOB fields to this appendix.

BLOB fields are binary large object fields which can be of varying length. They present a special problem when you are attempting to create a dataset which manages it's own record buffers. This problem arises because the space allocated to the record buffers is fixed in length whereas the data in a BLOB field is not.

The easy way to solve this problem is to allocate pointers in the record buffer to another chunk of memory that holds the BLOB field. Care must be taken when using this technique to not overwrite these pointers in the buffer otherwise memory leaks and access violations will ensue.

The TGXBaseDataset that we presented in this presentation supports BLOB fields of type ftMemo and ftGraphic. To support BLOB fields, we have added some additional code to various methods.

The first method we will look at is AllocRecordBuffer. As you may recall, it was implemented as follows in the TGXBaseDataset:

```
function TGXBaseDataset.AllocRecordBuffer: PChar;
begin
  GetMem(Result,GetRecordSize);
  AllocateBlobPointers(Result);
end;
```

As explained earlier, the first line allocates memory for the buffer. The second line, which calls AllocateBlobPointers, creates the various BLOB pointers and stuffs them into the record buffer.

```
procedure TGXBaseDataset.AllocateBLOBPointers(Buffer: PChar);
var Index: Integer;
Offset: Integer;
Stream: TMemoryStream;
Begin
  for Index:=0 to FieldDefs.Count-1 do
    if FieldDefs[Index].DataType in [ftMemo,ftGraphic] then
      Begin
      Offset:=GetFieldOffset(Index);
      Stream:=TMemoryStream.Create;
      Move(Pointer(Stream),(Buffer+Offset)^,sizeof(Pointer));
      End;
End;
```

As we can see above, we simply iterate through the fielddefs of the dataset and for any BLOB fields we come across we create a memory stream and stuff a pointer to the memory stream in the appropriate position in the record buffer.

A companion method, FreeBlobPointers, performs the reverse operation by freeing all Blob pointers in the record buffer.

Now that we are allocating a spot to hold the Blob fields, we need to move the Blob information in and out of our memory stream. The first step in doing this is in the TGXBaseDataset methods RecordToBuffer. When we load the record into the buffer in this method, if the field is a ftMemo or ftGraphic, we call the TGXBaseDataset method GetBlobField.

```
procedure GetBlobField(Field: TField; Stream: TStream);
```

Descendant datasets override the GetBlobField method and copy the Blob field data into the stream. This stream is the stream whose pointer is stored in the record buffer for that field.

The second step is done by overriding the method CreateBlobStream in TDataset. It has been implemented as follows:

```
function TGXBaseDataSet.CreateBlobStream(Field: TField;
  Mode: TBlobStreamMode): TStream;
begin
  Result:=TGXBlobStream.Create(Field as TBlobField, Mode);
end;
```

This function is called whenever the dataset needs to read or write the data in the blob field. The field parameter is the field which needs to be handled while Mode specifies whether or not the blob field is being read, written or is in read/write mode.

In the implementation above, we are creating a custom streaming object called TGXBlobStream. This object provides the interface between the blob data and the dataset. In this example TGXBlobStream is derived from TMemoryStream, though in your own implementations this stream can be based on any form of stream.

In order to provide communication between the stream and the dataset, we have added a custom constructor to the stream so that the field the stream works with is passed as a parameter. We have also passed the mode into the constructor as well.

```
Constructor TGXBlobStream.Create(Field: TBlobField;
  Mode: TBlobStreamMode);
Begin
  inherited Create;
  FField:=Field;
  FMode:=Mode;
  FDataSet:=FField.DataSet as TGXBaseDataset;
  if Mode<>bmWrite then LoadBlobData;
```

```
End;
```

In the constructor above, we save a reference to the field and dataset that the stream will be working with. We also save the blob stream mode that was passed. Finally, if the mode isn't bmWrite, we load the blob data using the method called LoadBlobData.

```
procedure TGXBlobStream.LoadBlobData;
var Stream: TMemoryStream;
Offset: Integer;
RecBuffer: PChar;
Begin
  Self.Size:=0;
  RecBuffer:=FDataset.GetActiveRecordBuffer;
  if RecBuffer<>nil then
    Begin
    Offset:=FDataset.GetFieldOffset(FField.Index);
    Move((RecBuffer+Offset)^,Pointer(Stream),sizeof(Pointer));
    Self.CopyFrom(Stream,0);
    End;
  Position:=0;
End;
```

This method simply locates the Stream pointer we stored in the record buffer earlier in the AllocateBlobPointers method and then reads in the BLOB data from that stream.

We then override the Write method that TGXBlobStream inherits from TStream and set the private variable FModified to True if any information is written. When the stream is destroyed, we check to see if this variable is true, and if so, it indicates that the blob data has changed and must be written back to the record buffer. The stream is written back in the SaveBlobData method of TGXBlobStream.

Finally, when the data is posted in InternalPost, the TGXBaseDataset calls the method SetBlobData. Descendant datasets override SetBlobData and write back the stream passed in this method to the specified Blob field.

Conclusion

While BLOB fields had a bit of extra complexity to implementing a dataset, it is not that difficult once you know what all of the steps are.