The Ultimate TDataLink?

by Joanna Carter

During the course of writing GridProQuo (*The Ultimate Grid Component?*) I discovered I needed to use a TDataLink derivative to communicate between the data and the component. A TFieldDatalink is all very well and good for those components that only need to access one field at a time, but GridProQuo needed to access all the fields, all the time.

I took a long hard look at the base TDataLink class and then at the source code for TDB..Grid. This is where the problems started. Firstly, I was creating my TFlexiGrid component from scratch. My starting place was not to be TCustomGrid or anything as sophisticated as that, because one of the major requirements of TFlexiGrid was that it had to be capable of displaying every row at a different height, according to how much data was in memo or graphic fields. So I determined that I would have to derive from TCustomControl. This meant that much of the functionality in the TDB..Grid source was of little help.

Now the TDB..Grid family of components uses a TGridDataLink class that has been written with those classes in mind and is fairly inextricably bound in to the source for those components. What was required was a generic TDataLink derivative that would serve the needs of any component that used more than one field of a dataset.

There is a common methodology for including data handling in a component that suggests including a TDataLink as a private field in the component. This is to be recommended, but one problem occurs when you delete either the DataSet or DataSource components from the form. The component then has a hanging pointer that needs to be set to nil so that you can test its validity before doing any operation in the component. The most common advice given in this situation is to override the Notification method in the component to trap when the DataSource or DataSet is removed from the form and set the internal reference to that data component to nil.

```
procedure TMyComponent.Notification(AComponent: TComponent;
  Operation: TOperation);
begin
  inherited Notification(AComponent, Operation);
  if (AComponent = fDataLink.DataSource) and
    (Operation = opRemove) then
    fDataLink.DataSource := nil;
end;
```

➤ *Listing 1*

```
constructor Create(AComponent: TComponent); virtual;
destructor Destroy; override;
property DataSetName: string read GetDataSetName write SetDataSetName;
property OnNewDataSet: TDataSetNotifyEvent
property OnDataSetOpen: TDataSetNotifyEvent
property OnDataSetChange: TDataSetNotifyEvent
property OnIndexChange: TDataSetNotifyEvent
property OnLayoutChange: TDataSetNotifyEvent
property OnPostData: TDataSetNotifyEvent
property OnDataSetClose: TDataSetNotifyEvent
property OnInvalidDataSet: TDataSetNotifyEvent
property OnInvalidDataSource: TDataSetNotifyEvent
property OnDestroyDataLink: TDataSetNotifyEvent
```

➤ *Listing 2*

```
function TComponentDataLink.GetDataSetName: string;
begin
  Result := fDataSetName;
  if DataSet <> nil then
    Result := DataSet.Name;
  if DataSet is TTable then
    Result := TTable(DataSet).TableName;
  if DataSet is TQuery then
    if DataSet.DataSource <> nil then
      Result := TTable(TQuery(DataSet).DataSource.DataSet).TableName;
end;
```

➤ *Listing 3*

If you have an internal reference to both the DataSet and the DataSource stored in, say, fDataSource and fDataSet, within the component, then this will work. In fact, the idea of using a TDataLink then becomes somewhat redundant. However, using a TDataLink is definitely to be preferred. As you will see, it provides you with a great deal more functionality that would be difficult to replicate repeatedly, every time you write a new component. However, if you use the code in Listing 1 to try and determine when and whether fDataLink.DataSource is invalid then you will come to a sticky end. When TDataSource is being destroyed, it makes a NotifyDataLinks call that sets the DataSource property of the TDataLink to Nil, long before Notification ever gets called. What is needed is to get into the workings of the TDataLink itself and use what is in there to catch whatever happens to the data side of the link.

So, I came up with a class that I have called TComponentDataLink which is derived from TDataLink and uses and enhances that functionality. Listing 2 shows the public interface (the full code is on the disk of course).

First we create a new constructor that takes a TComponent as a parameter, this is to ensure that we have an internal pointer to the component for the life of the datalink. Then we need a reference to the dataset that is being accessed, so that, when the form is streamed to and from the DFM file, we can keep track of the dataset to which we are connected. This is provided by the DataSetName property. Although the Set method simply assigns a string to an internal variable, fDataSetName, the Get method is a little more sophisticated (Listing3).

Firstly the stored value of the internal variable that may or may not be valid is assigned to Result. Then the actual dataset name is assigned over the top of it, in case the dataset has changed. This allows a check to be made for this change by comparing the private fDataSetName and the more sophisticated DataSetName property. Finally two checks are made to see whether the dataset is a TTable or TQuery and in either case, if a TableName has been assigned. If you think that sounds a bit odd, take it from me, it was necessary to make the whole thing work!

Now we come to the pivotal part of using a TComponentDataLink. To find out what is happening to the data components. TDataLink declares several empty virtual procedures that allow you to intercept events that are generated from the dataset and datasource, these must be declared and overridden if we want to make use of them (Listing 4). I did not need to use every one, but here are the three that I found to be most useful.

The first one we will look at is the ActiveChanged procedure. The code is quite complex, so rather than just comment it, I will say something about each section as we go through it. You will see reference to events being called, that can be assigned to in the component and the appropriate action taken.

This procedure is called whenever the Active status of the TComponentDataLink changes. The first case to check for is whether the link has just become active or inactive (Listing 5).

Now we know that the link has just become active, we now need to check whether the dataset has changed or just been re-opened. We also need to grab a pointer to

```
procedure ActiveChanged; override;
procedure CheckBrowseMode; override;
procedure DataSetChanged; override;
procedure DataSetScrolled(Distance: Integer); override;
procedure FocusControl(Field: TFieldRef); override;
procedure EditingChanged; override;
procedure LayoutChanged; override;
procedure RecordChanged(Field: TField); override;
procedure UpdateData; override;
```

➤ *Listing 4*

```
procedure TComponentDataLink.Activechanged;
begin
  if Active then begin
    fDataSet := DataSet;
    if DataSetName <> fDataSetName then begin
      fDataSetName := DataSetName;
      fIndexNames := TTable(fDataSet).IndexFieldNames;
      if Assigned(fOnNewDataSet) then fOnNewDataSet(DataSet);
    end else begin
     fIndexNames := TTable(DataSet).IndexFieldNames;
     if Assigned(fOnDataSetOpen) then fOnDataSetOpen(DataSet);
    end;
  end
end;
```

➤ *Listing 5*

```
...
else begin
  // Active = False
  if DataSet = nil then begin
    if Assigned(fOnInvalidDataSource) then
      fOnInvalidDataSource(fDataSet);
    fDataSet := nil;
    fDataSetName := '<INVALID>';
    fIndexNames := '';
  end
...
```

➤ *Listing 6*

```
...
else begin
  if (csDestroying in DataSet.ComponentState) then begin
    if Assigned(fOnInvalidDataSet) then
      fOnInvalidDataSet(fDataSet);
    fDataSet := nil;
    fDataSetName := '<INVALID>';
    fIndexNames := '';
  end
...
```

➤ *Listing 7*

the DataSet, for reasons that will become apparent later. A copy of the DataSetName and IndexFieldNames is also kept for later use. As you can see, coping with the TComponentDataLink becoming active is quite straightforward. Now for the fun bit, coping with things being closed, destroyed, changed and so on (Listing 6).

Finding out how to handle data components being deleted from the form involved drawing up a truth table, trapping every change and checking all possible states. If the link has just gone inactive then, if we find the DataSet is Nil, this indicates that the DataSource has been removed or invalidated in some way. Once again, it might seem odd, but that is how it works.

In order to detect when the DataSet is being destroyed, all we have to do is check it's ComponentState (Listing 7).

Finally, if the code executes successfully as far as Listing 8, then we know that all that has happened is that the DatSet has closed. We need to keep a note of things like the DataSetName and IndexFieldNames, so that we can detect any changes the next time the link becomes active.

```
...
else begin
  if Assigned(fOnDataSetClose) then
    fOnDataSetClose(DataSet);
  if DataSet <> nil then begin
    fDataSetName := DataSetName;
    fIndexNames := TTable(DataSet).IndexFieldNames;
  end;
...
```

➤ *Listing 8*

```
procedure TComponentDataLink.DataSetChanged;
begin
  if TTable(DataSet).IndexFieldNames <> fIndexNames then begin
    fIndexNames := TTable(DataSet).IndexFieldNames;
    if Assigned(fOnIndexChange) then
      fOnIndexChange(DataSet);
  end else
    if Assigned(fOnDataSetChange) then
      fOnDataSetChange(DataSet);
end;
```

➤ *Listing 9*

```
procedure TComponentDataLink.LayoutChanged;
begin
  if Assigned(fOnLayoutChange) then
    fOnLayoutChange(DataSet);
end;
```

➤ *Listing 10*

```
procedure TMyComponent.DefineProperties(Filer: TFiler);
begin
  inherited;
  Filer.DefineProperty('DataSetName', ReadDataLink, WriteDataLink,
    fDataLink.DataSource <> nil);
end ;
procedure TMyComponent.ReadDataLink(Reader: TReader);
begin
  fDataLink.DataSetName := Reader.ReadString;
end;
procedure TMyComponent.WriteDataLink(Writer: TWriter);
begin
  Writer.WriteString(fDataLink.DataSetName);
end;
```

➤ *Listing 11*

Now we need to look at changes that can occur in the `DataSet` without the necessity for opening and closing the `DataSet` (Listing 9).

`DataSetChanged` is called on many different occasions, but the main one I was interested in was when the `IndexFieldNames` property got changed. Looking at the above code you will now realise why we kept an internal reference to the index fields when we both opened and closed the dataset. Of course, as a courtesy, if we aren't handling anything else here, then we call an assignable event so as not to break the event chain. We could also separate out other events in this procedure.

Moving on to `LayoutChanged` (see Listing 10), this is called when you change the order of fields in the dataset, maybe by using the Fields Editor in the IDE. All that is needed here is to call the assignable event.

That handles most of the code that can be handled in the `TComponentDataLink`, but there is one thing that I found very important, that can only be handled in the component that you are writing.

It is important in maintaining the rest of the `TComponentDataLink` to ensure that the name of the current `DataSet` is written out to the DFM file as and when the component is streamed out. To accomplish this we need to use the `DefineProperties` method (Listing 11). Although it would be possible to tamper with the DFM file, using this method will mean that the `DataSetName` property will not appear in the Object Inspector, where it could more easily be corrupted.

I would be the first to agree that some of this logic could be handled by the dataset events that are provided and maybe `TComponentDataLink` could be further enhanced, but, on the other hand, this seems to be the only way to reliably detect removal of `TDataSource` and `TTable` components. Unless, of course, you know otherwise?

Joanna Carter is a freelance consultant, developer and trainer based in Liverpool. You can email her at joannac@btinternet.com