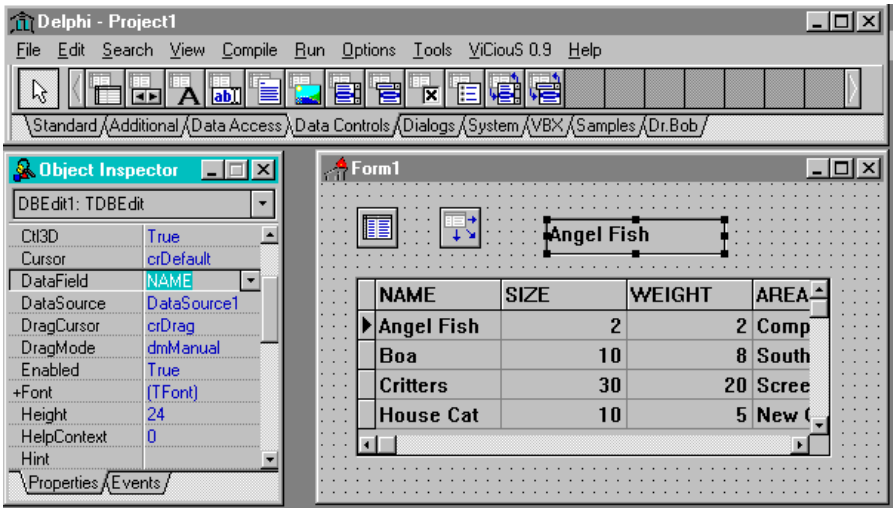
Under Construction: Data-Aware Components

by Bob Swart

This month we're going to look at a topic we have not covered at all so far: visual data-aware components!

Components can be classified into two groups: visual and non-visual, as we've seen in previous columns. However, we can also use other criteria to group components together, such as VBX-based, DLL-based or wholly native components, or data-aware and non-data-aware components. For data-aware components, Delphi has reserved two pages on the component palette: Data Access and Data Controls.



➤ *Figure 1*

Data Access

The components on the Data Access page can be seen as the non-visual data-aware components. Among them are `TTable`, `TQuery` and a special one named `TDataSource`. This last component can be seen as a connector between a non-visual data-aware component (the ones that hold the actual data) and a visual data-aware component (the ones that display that data). Actually, `TDataSource` is connected to `TDataSet` (from which both `TTable` and `TQuery` are derived) and stores the connected `TDataSet` in the `DataSet` property. It's up to the visual data-aware controls to connect to the `TDataSource` from the other side and hence visualise the data on the form.

Data Controls

The components on the Data Controls page are the visual data-aware components. In order for them to work they have to be connected to a `TDataSource` and for this purpose they have a property called `DataSource`. The main reason why visual data-aware controls are not directly connected to a dataset but to a connecting datasource instead is to give the user the ability to switch an entire set of visual data-aware controls from one table to another by giving the `DataSet` property of the connecting datasource another value. If there was no datasource in between them, the user would have to change all the `DataSet` properties of all the visual data-aware components. Furthermore, the datasource abstracts the visual data-aware controls from the actual source of the data. The data could be from a table or query and the 'end-user' controls don't really have to know!

Note that the data can flow from the table to the visual component (if you display something) or the other way around (if a new value has been entered). The data-flow itself is therefore bi-directional. As a practical example, we can drop a `TTable` (`Table1`) onto a form, set its `DatabaseName` property (to `DBDEMOS`) and `TableName` property (to table ANIMALS.DBF), and set `Active` to `True`. Now, drop a `TDataSource` (`DataSource1`) next to it, set its `DataSet` property to `Table1` and now you're ready to drop visual data-aware controls from the Data Controls page onto the form and connect them to `DataSource1`.

If we do this, we can see a distinction between some of the visual data-aware controls: some of them start to work immediately when we set the DataSource property (like `TDBGrid`) and some need some additional work (like `TDBEdit`). The latter one does not work on entire records, but on single fields. Actually, this is the case for most of the visual data-aware components: we have to specify a field as well and therefore need to give the property `DataField` a valid field value (see Figure 1).

Writing Data-Aware Controls

So, enough background about data-aware controls for now. Let's move on and actually create a visual data-aware control of our own. The main problem is to think of an original control: one that isn't already present on the Data Control page of the component palette and that hasn't been done before (I've seen lots of data-aware outliners, for example). Personally, I have always wondered about BLOB (Binary Large OBject) fields. They are most often used to hold

image data, but can they be used for anything else? Yes, of course they can. My version control system (*ViCiouS*, more about this in a future article) puts entire source or form files in BLOB fields.

In fact, you could store just about anything in BLOB fields, including multimedia stuff such as WAV files. Hey, now there's a good idea, why not make a data-aware WAV playing button based on a BLOB field: a `TDBWavButton`!

TDBWavButton

From what we've learnt in the first part of this article, it seems that a `TDBWavButton` would need a `DataSource` property as well as a `DataField` property (after all, it's only interested in the data of one BLOB field). Using the component expert to derive a new component from `TBitBtn` (so we can add a nice glyph as well, using the property editor we developed in the February issue) and adding these two properties, we get the skeleton source code for our `TDBWavButton` shown in Listing 1.

The question is, where or how do we store the values of the `DataSource` and `DataField` properties? In this situation, it always helps to have the Visual Class Library source code available on disk. Remember, it's included in the RAD Pack and the Client/Server version of Delphi 1.0 and 2.0, plus Delphi Developer 2.0, or you can buy it separately from Borland – it's a serious *must have* for all component builders!

Consulting the VCL source code, it seems that the connection between a visual data-aware control and a `TDataSource` component does not actually happen using a property of type `TDataSource`, but with a so-called data link property (`FDataLink` of type `TDataLink`).

There are two kinds of data links with Delphi: the ones that connect an entire table or record (for example `TDBGrid`) and the ones that connect to a specific field. The latter is the one we are interested in and is of type `TFieldDataLink`. It's the data link that has internal properties of type `TDataSource` (for the `DataSource` property) and `String` for the `DataField` property. Adding the access function for the `DataSource` and `DataField` properties yields the new source for `TDBWavButton` which is shown in Listing 2.

```
Type
  TDBWavButton = class(TBitBtn)
  published
    property DataSource: TDataSource;
    property DataField: String;
  end;
```

➤ *Listing 1*

```
unit DbWavBtn;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, DB, DBTables;
Type
  TDBWavButton = class(TBitBtn)
  private
    { Private declarations }
    FDataLink: TFieldDataLink;
  protected
    { Protected declarations }
    function GetDataSource: TDataSource;
    procedure SetDataSource(Value: TDataSource);
    function GetDataField: string;
    procedure SetDataField(const Value: string);
  published
    { Published declarations }
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    property DataField: String read GetDataField write SetDataField;
  end;
procedure Register;
implementation
function TDBWavButton.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;
procedure TDBWavButton.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
function TDBWavButton.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;
procedure TDBWavButton.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TDBWavButton]);
end;
end.
```

➤ *Listing 2*

But we're not done yet: we must of course make sure that the data field is created and destroyed together with the object itself. To do this, we must override the `public` constructor `Create` and the destructor `Destroy` as shown in Listing 3.

Now we're ready to install the component on the component palette. It will be able to connect to a datasource, and give us the ability to pick a field from that datasource. Next we'll look at how we can include functionality to let the data link and the bit-button interact with each other (it's almost like multiple inheritance...).

OnDataChange

The `TDBWavButton` component that we've designed so far is still an

empty shell. It does nothing to even remotely connect the field data and the button click action.

But before we even talk about this connection, first of all the button needs to be made truly aware of its data. For this, it needs to respond to the `OnDataChange` event of the data link: an event that gets fired whenever the data in the field changes (for example, when the user scrolls from one record to another, or when the datasource gets disconnected). An initial way to respond to this event could be as follows:

```
Enabled := Assigned(FDataLink)
```

But we need to do more, of course. This `DataChange` event is the perfect place to get the WAV data out of the field and into a local memory stream that can be used by the click method of the button to actually play the WAV file.

For this, we need to add a hidden field named `MemoryStream` to the component, of type `TMemoryStream`. This field is to be created inside the constructor and freed inside the destructor (just like the `DataField` itself). Also, in the `DataChange` method, we can fill it with the contents of the blob field as shown in Listing 4.

Note that we need to check if the `FDataLink.Field` is actually a `TBlobField`. And, even if it *is* a `TBlobField`, we cannot be certain that it will necessarily contain a WAV file data. For all we know, it could contain image data (which is actually more usual, see the *Database Expert* article elsewhere in this issue).

But, for now, we'll assume that the user of the component knows what they are doing and has been careful to assign the `TDBWavButton` to a valid `TBlobField` which holds WAV data.

Click

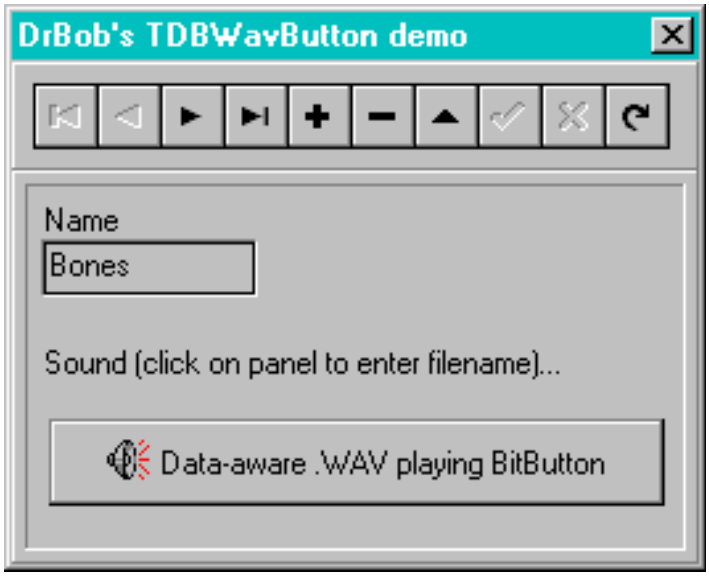
All that remains now for our new `TDBWavButton` is to override the click method of the parent class `TBitBtn`, in order to play the WAV file inside the `MemoryStream` and execute inherited click as well. Note that we can use the `Memory` property of the `MemoryStream` to supply a pointer to `sndPlaySound`:

```
procedure TDBWavButton.Click;
begin
  sndPlaySound(
    MemoryStream.Memory,
    SND_ASYNC OR SND_MEMORY);
  inherited Click;
end;
```

This code is actually very dangerous. It plays the sound asynchronously, in the background, and it won't stop until the sound playing has finished. But what if we change the contents of the `MemoryStream` during playing, ie what if we go to another record while the sound is still playing? Well, all I can say is don't do it! I've been able to bring Windows 95 down with this test and that's not always a simple thing to do!

We need to stop playing the sound just before we do anything else in the `OnDataChange` event method and while we're at it, we need to make sure to stop playing the sound when we destroy the component as well (for the same obvious reason). To stop `sndPlaySound` from playing the current sound, we have to call it again with a `nil` argument (I'm not sure if this is officially documented somewhere, but at least it's good to know):

➤ *Figure 2: Our new component in action*




```
constructor TDBWavButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FDataLink := TFieldDataLink.Create;
end;
destructor TDBWavButton.Destroy;
begin
  FDataLink.Free;
  inherited Destroy;
end;
```

➤ *Listing 3*

```
procedure TDBWavButton.DataChange(Sender: TObject);
begin
  Enabled := Assigned(FDataLink);
  if Enabled then begin
    MemoryStream.Clear;
    if (FDataLink.Field IS TBlobField) then
      with (FDataLink.Field AS TBlobField) do SaveToStream(MemoryStream)
  end
end;
```

➤ *Listing 4*

```
sndPlaySound(nil, 0);
```

All this leads us to the final version of the source code of TDBWavButton, shown in Listing 5.

Demo

The TDBWavButton is a read-only visual data-aware component. It will not be able to enter a WAV file into a field of a database. So how do we test this component? Well, to that end I've written a little demo program (see Figure 2). It's based on a table with two records. Each record has two fields, an alpha field called Name and a BLOB field called Sound. A TOpenDialog is used to get a filename from the user in order to fill the BLOB field, as follows:

```
if OpenDialog1.Execute then
  Table1Sound.LoadFromFile(
    OpenDialog1.FileName);
```

If there is a WAV file loaded in the Sound field we can click on the data-aware WAV playing button to play this file. I've put two demo sounds in the database: one of Spock and his auto-answering machine and another of Bones (his reaction when he read this article and saw the component in action...).

Next Time

Next time in Under Construction we'll return to the subject of Tools APIs and Delphi IDE Experts when we start exploring the so-called Component Editors.

Stay tuned, and make sure you've always got a backup of your COMPLIB.DCL in a save place!

Bob Swart (you can email him at 100434.2072@compuserve.com) is a professional 16- and 32-bit software developer using Borland Delphi and sometimes a bit of Pascal or C++. In his spare time, he likes to watch video tapes of Star Trek Voyager with his almost two year old son Erik Mark Pascal.

➤ *Listing 5*

```
unit Dbwavbtn;
interface
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, Buttons, DB, DBTables;
Type
  TDBWavButton = class(TBitBtn)
  private   { Private declarations }
    FDataLink: TFieldDataLink;
    MemoryStream: TMemoryStream;
    procedure DataChange(Sender: TObject);
  protected { Protected declarations }
    function GetDataSource: TDataSource;
    procedure SetDataSource(Value: TDataSource);
    function GetDataField: string;
    procedure SetDataField(const Value: string);
  public    { Public declarations }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure Click; override;
  published { Published declarations }
    property DataSource: TDataSource read GetDataSource write SetDataSource;
    property DataField: String read GetDataField write SetDataField;
  end;
procedure Register;
implementation
uses MMSystem;
constructor TDBWavButton.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
  MemoryStream := TMemoryStream.Create;
end;
destructor TDBWavButton.Destroy;
begin
  sndPlaySound(nil, 0);
  FDataLink.Free;
  MemoryStream.Free;
  inherited Destroy;
end;
procedure TDBWavButton.DataChange(Sender: TObject);
begin
  Enabled := Assigned(FDataLink);
  if Enabled then
  begin
    sndPlaySound(nil, 0);
    MemoryStream.Clear;
    if (FDataLink.Field IS TBlobField) then
      with (FDataLink.Field AS TBlobField) do SaveToStream(MemoryStream)
  end
end;
procedure TDBWavButton.Click;
begin
  sndPlaySound(MemoryStream.Memory, SND_ASYNC OR SND_MEMORY);
  inherited Click;
end;
function TDBWavButton.GetDataSource: TDataSource;
begin
  Result := FDataLink.DataSource;
end;
procedure TDBWavButton.SetDataSource(Value: TDataSource);
begin
  FDataLink.DataSource := Value;
end;
function TDBWavButton.GetDataField: string;
begin
  Result := FDataLink.FieldName;
end;
procedure TDBWavButton.SetDataField(const Value: string);
begin
  FDataLink.FieldName := Value;
end;
procedure Register;
begin
  RegisterComponents('Dr.Bob', [TDBWavButton]);
end;
end.
```