File Handling 5: Streaming Components

by Brian Long

In the last issue we introduced the way Delphi puts components in form files: it uses the run time type information (RTTI) generated for a component's `published` section to find all the published properties, which it then proceeds to write out to the DFM file.

The DFM file is a Windows resource file containing a custom resource. It is linked into your program by the `$R` compiler directive at the top of your form unit's `implementation` section. The directive `{$R *.DFM}` *doesn't* mean link in all DFM files, as you might expect, but refers to the DFM file with the same name as this unit. The custom resource in the form file is an object stream: the form object's non-default property values and all the non-default properties of all the objects on the form.

You can manufacture a file exactly the same as a DFM file at run-time using `WriteComponentResFile` from the `Classes` unit, as shown in Listing 1. I placed this in the form's `OnClick` handler and when I clicked on the form at run-time I was given an FRM file that exactly matched my DFM file. Running the utility CONVERT.EXE over both of them gave the same text file version of the form.

All `WriteComponentResFile` does is simply construct a temporary `TFileStream` and then calls its `WriteComponentRes` method, using the component's class name (as returned by the `ClassName` method) as the resource name, and the component as the resource. `WriteComponentRes` in turn writes a Windows resource header and then calls the stream's `WriteComponent` method which again delegates the hard work to someone else. It constructs a `TWriter` object and calls its `WriteRootComponent` method. When writing a form file, the form is the root component as it owns all components which also need to be written to the stream, ie all the components on the form.

Custom Component Streaming (Version 2)

We will now return to the STRM1.DPR program from the last issue (also included on this issue's disk for easy comparison), which defined basic objects and streamed them by using custom `SaveToStream` and `LoadFromStream` methods. The idea of using components is that we can somehow improve the situation. The problem before was that we needed to know what objects were in the stream, and in what order, to get them back out again. Also, we were responsible for creating all the objects whose data was read back in. A form in a DFM file does not have this requirement: all components on the form are automatically manufactured when the form is read in.

As the first stage along the road of improvement, we will simply redeclare the `TPointData` structure as a component, as shown in Listing 2. Since components have the capability of being owned, we ensure that the constructor takes an owner component as a parameter. This means we don't need to delete any outstanding `PointData` objects

➤ *Listing 1*

```
{ At run-time, a form's name property is blank. Remember there could be
  more than one instance of a form - consider MDI children. A blank name
  means no component name conflicts are likely. This sets the name back to
  what I had it as at design time }
Name := 'Form1';
{ Visible is False at design time - this sees to that }
Hide;
{ ActiveControl was blank at design time }
ActiveControl := nil;
{ Write out a form file - UNIT1.FRM should end up the same as UNIT1.DFM }
WriteComponentResFile('UNIT1.FRM', Form1);
{ Unhide the form }
Show;
```

➤ *Listing 2*

```
TPointData = class(TComponent)
private
  FX, FY: Word;
public
  constructor CreateXY(AOwner: TComponent; AX, AY: Word);
  procedure SwapXY;
published
  property X: Word read FX write FX default 0;
  property Y: Word read FY write FY default 0;
end;
...
constructor TPointData.CreateXY(AOwner: TComponent; AX, AY: Word);
begin
  inherited Create(AOwner);
  FX := AX;
  FY := AY;
end;
procedure TPointData.SwapXY;
begin
  Tag := FX;
  FX := FY;
  FY := Tag;
end;
```

at the end of the program run, someone else (the owner) will take care of that.

Notice the `default` specifier in the property declaration. This specifies the value in the RTTI that the VCL streaming mechanism will compare the property value against before deciding whether to store it or not. It does not actually give the property a default value (a common misunderstanding) – that job is left to the programmer. In our case it's easy: if the normal `Create` constructor is called, `FX` and `FY` will be zero anyway, as all object data fields are zeroed when the object is constructed.

We could have achieved the same net result as the `default` specifier by using the `stored` directive instead. Listing 3 is an alternative component definition for `TPointData` that uses a `Boolean` function specified with the `stored` directive, to decide whether to store the property in the stream or not. Note you can also use a `Boolean` data field or a `Boolean` constant.

As was discussed briefly last time, a stream has a method for writing a component's properties, so we could change `LoadBtnClick` and `SaveBtnClick` as in Listing 4.

But these versions have the same disadvantage as the old versions. We have to know which components are in the stream and create them ourselves. The idea was to make the streaming mechanism construct the components as they are read from the stream. Fortunately, a simple change to the `LoadBtnClick` handler achieves this. The `while` loop changes to

```
while Stream.Position <>
  Stream.Size do
  PointList.Add(
    Stream.ReadComponent(nil));
```

and now the stream (or more correctly the `TReader` it uses, where a `TReader`, like its compatriot the `TWriter`, are both descendants of `TFiler`) finds the name of the class in the stream and attempts to make an instance of the class. However, all that's in the stream is a string, and that is not sufficient to make a new class with. It searches a list of classes it knows about to see if it can find a class reference whose name matches the string, but in this case it will be unsuccessful.

For custom objects that are not already dealt with by a form, we need to register the classes. In the `initialization` section of the form unit I have added

```
RegisterClass(TPointData);
```

If I needed to register several classes, I would have used

```
RegisterClasses([TPointData,
  SecondClass, ThirdClass]);
```

The finished project is STRM2.DPR, but it is not yet done with. If you compare a stream written by STRM1.EXE with one written by STRM2.EXE you will see that the latter's is considerably larger. All that is stored in the first version's file are numbers. In the latest version we have information to allow the streaming mechanism to construct new objects. Close

```
TPointData = class(TComponent)
private
  FX, FY: Word;
  function IsX: Boolean;
  function IsY: Boolean;
public
  constructor CreateXY(AOwner: TComponent; AX, AY: Word);
  procedure SwapXY;
published
  property X: Word read FX write FX stored IsX;
  property Y: Word read FY write FY stored IsY;
end;
...
function TPointData.IsX;
begin
  Result := X <> 0;
end;
function TPointData.IsY;
begin
  Result := Y <> 0;
end;
```

➤ *Listing 3*

```
procedure TForm1.SaveBtnClick(Sender: TObject);
var
  Stream: TFileStream;
begin
  Stream := TFileStream.Create(DataFile, fmCreate);
  try
    for Loop := 0 to PointList.Count - 1 do begin
      Pt := TPointData(PointList.Items[Loop]);
      Stream.WriteComponent(Pt);
    end;
  finally
    Stream.Free;
  end;
  ClearPoints;
  PaintBox1.Invalidate;
end;
procedure TForm1.LoadBtnClick(Sender: TObject);
var
  Stream: TFileStream;
begin
  ClearPoints;
  Stream := TFileStream.Create(DataFile, fmOpenRead or fmShareDenyWrite);
  try
    while Stream.Position <> Stream.Size do begin
      Pt := TPointData.Create;
      Stream.ReadComponent(Pt);
      PointList.Add(Pt);
    end;
  finally
    Stream.Free;
  end;
  Invalidate;
end;
```

➤ *Listing 4*

examination of this information shows that for each object in the stream there is a signature string, TPF0 (Turbo Pascal Filer version 0), a class name and some binary data representing the property values. There are also symbols indicating the size of each particular value (these are the ordinal values of some members of the Classes unit enumerated type TValueType, vaInt8 and vaInt16).

Version 3

Every component that gets explicitly written out is called a root component. Each root component is preceded by the signature TPF0, so that some simple validation can be performed on streams. The only way to avoid having this written for each point object would be to write all the points out at the same time, instead of iterating over all of them. We are unable to write the list out as it is based on type TObject, not TPersistent (or TComponent) and anyway it knows nothing about the items it maintains references to, other than their addresses.

Instead, for the next version we'll replace the TList version of PointList with a component to hold the points. There is no need to worry about adding list functionality, a component already has as much as we need, so TPointList can be based on type TComponent. When a component becomes the owner of another component, the owned component goes into an array property called Components and another property which is called ComponentCount gets incremented.

There are several benefits to doing this. Since the list is a component, we can get it owned by the form, thus relinquishing our responsibility for destroying it. When we wish to write all our points out, providing we make the list own the points, we can simply write the list component out, and similarly for reading. This means only one filer signature will be written and read. Also, the ClearPoints method, which empties the list and destroys all the points, can now become a simple call to the PointList method DestroyComponents, which will destroy all that it owns, ie all the points. The TPointList class in all its glory, along with some of the methods from STRM3.DPR, are given in Listing 5. Indeed, there was no real need to make a new class, TComponent would have been fine on its own.

```
TPointList = class(TComponent)
end;
...
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
  for Loop := 0 to PointList.ComponentCount - 1 do begin
    Pt := PointList.Components[Loop] as TPointData;
    if Loop = 0 then
      PaintBox1.Canvas.MoveTo(Pt.X, Pt.Y)
    else
      PaintBox1.Canvas.LineTo(Pt.X, Pt.Y)
  end;
end;
...
procedure TForm1.SaveBtnClick(Sender: TObject);
var
  Stream: TFileStream;
begin
  Stream := TFileStream.Create(DataFile, fmCreate);
  try
    Stream.WriteComponent(PointList);
  finally
    Stream.Free;
  end;
  ClearPoints;
  PaintBox1.Invalidate;
end;
...
initialization
  Randomize;
  RegisterClass(TPointData);
end.
```

➤ *Listing 5*

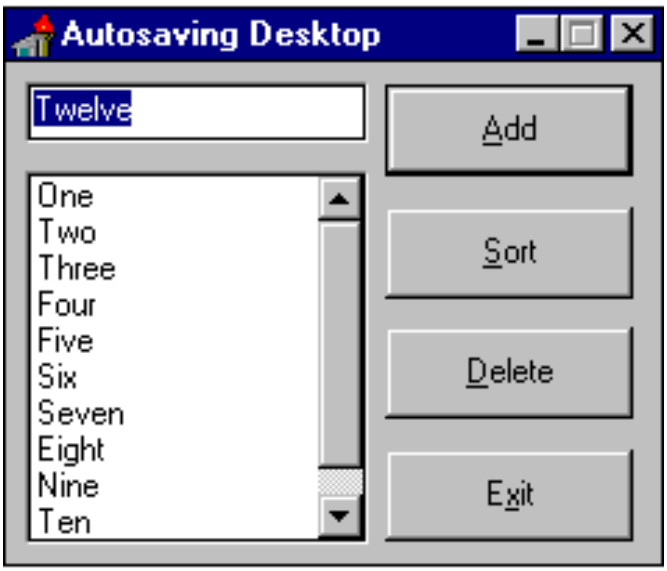
```
TPointData = class(TComponent)
public
  X, Y: Word;
  constructor CreateXY(AOwner: TComponent; AX, AY: Word);
  procedure SwapXY;
  procedure DefineProperties(Filer: TFiler); override;
  procedure ReadData(Reader: TReader);
  procedure WriteData(Writer: TWriter);
end;
...
procedure TPointData.DefineProperties(Filer: TFiler);
begin
  { Not calling inherited version cos I don't want any properties bar X and
Y stored }
  Filer.DefineProperty('XY', ReadData, WriteData, X or Y <> 0);
end;
procedure TPointData.ReadData(Reader: TReader);
begin
  X := Reader.ReadInteger;
  Y := Reader.ReadInteger;
end;
procedure TPointData.WriteData(Writer: TWriter);
begin
  Writer.WriteInteger(X);
  Writer.WriteInteger(Y);
end;
```

➤ *Listing 6*

Version 4

There's one more thing to mention about standard component streaming and it again relates to type TPersistent. A TWriter object will write the class name, instance name and properties of a component out to a stream. Sometimes it isn't desirable to turn everything that should be streamed into a property. If this is the case, there is another option available to allow storage of your data. TPersistent has a virtual method called

➤ *Figure 1*




```
procedure TForm1.FormClose(Sender: TObject; var Action: TCloseAction);
begin
  with TFileStream.Create(DataFile, fmCreate) do
    try
      WriteComponent(ItemEdt);
      WriteComponent(ItemsLst);
    finally
      Free;
    end;
end;
procedure TForm1.Loaded;
begin
  inherited Loaded;
  try
    with TFileStream.Create(DataFile, fmOpenRead or fmShareDenyWrite) do
      try
        ReadComponent(ItemEdt);
        ReadComponent(ItemsLst);
      finally
        Free;
      end;
  except
    { Smother desktop not found exception as it won't be found on the first run }
    on EFOpenError do {nothing};
  end;
end;
procedure TForm1.DelBtnClick(Sender: TObject);
var Loop: Byte;
begin
  with ItemsLst, Items do begin
    BeginUpdate;
    if not MultiSelect then
      Delete(ItemIndex)
    else
      for Loop := Pred(Count) downto 0 do
        if Selected[Loop] then
          Delete(Loop);
    EndUpdate;
  end;
end;
```

➤ *Listing 7*

DefineProperties. After all the real properties have been written, a TWriter calls the DefineProperties method of a TPersistent derivative, which allows it to define fake properties and dictate how they're written out.

To demonstrate, STRM4.DPR has had the TPointData properties removed, and X and Y are once again public data fields. The DefineProperties method has been overridden (see Listing 6) to define one fake property called XY. Inside a component's DefineProperties method, you normally call the inherited method, but this one elects not to. In this case, the only property I want on the stream is my new XY property and nothing else.

To define a new property you call either DefineProperty or DefineBinaryProperty, each of which is a method of the passed-in TFiler object. Normally the former is used, although graphic objects use the latter. DefineProperty takes a property name, a method that can read the property, a method to write the property and also a Boolean expression that dictates whether there is any data to store. The only combination of two values OR-ed together that yield a zero are zero and zero, hence my expression is X OR Y <> 0. As the listing shows, ReadInteger can be used to read a value of any integer type, and WriteInteger will write any integer type.

Saving A Desktop File

We can take advantage of this component streaming ability to add a desktop-saving feature to our programs. The project DESKTOP.DPR does this (see Figure 1). When the program exits, it saves the edit box and list box to a stream, which we can call our desktop file. When the program starts up, after all components have been read in and set up from the form, the overridden Loaded method reads the properties of the edit and list box back from the file. The net effect is that each time you run the program, it looks just like it did when you last closed it, data and all. The Loaded method and the form's OnClose event are shown in Listing 7. Also included in that listing is the code for the Delete button. Delete removes the currently selected items from the list box, and caters for both single-selection and multiple-selection list boxes.

Making Your Own Run-Time Resource Stream (Version 5)

Earlier, we saw how to make a DFM file, just like the ones Delphi generates. Indeed we can make a Windows resource file filled with components any time we like, but what's the point? Well, going back to the discussion in the last issue, about object streams allowing you to remove initialisation from your

program, it can be left to another program to set a stream up and your program can then read it. We will test out the theory.

Delphi generates an object resource file and your program reads in the resource at run-time. We will again amend the stream program, developed through this article, to do the same thing with some custom objects. First things first, we need a set-up program. SETUP.DPR on the disk will generate a resource file with a list of five point objects in it. Because the file format of Windows resources changes between 16- and 32-bit, some conditional compilation is used to generate a uniquely named file. All the hard work is done in the `Loaded` method (see Listing 8), called when the form has read itself in.

The five point objects draw out a square. The plan is that when the stream program starts up, it will read in its point list from a resource, rather than constructing one and waiting for the user to generate some random points. This means that as soon as the form shows on the screen, a square will be drawn on it.

The stream program will be an extension of STRM3.DPR – the one with the real properties, rather than faked ones. There are just a few changes to be made (see Listing 9). Firstly, we need to link the resource into the program. A `$R` directive does that. Then we need to amend the `initialization` section of the unit, so that both `TPointData` and `TPointList` are registered – we're going to get the streaming mechanism to construct the list and the points. Lastly, we change the code in the form's `OnCreate` handler. It currently creates a `TPointList` object, instead we will call `ReadComponentRes` to read it in.

A rather pleasing side-effect of writing Delphi objects into resources is that the CONVERT.EXE program will now translate the binary file into a text file. A command line of

```
CONVERT POINTS.R16
```

yields the text shown in Listing 10. Of course, like a form, this can be

```
const
{$ifdef Windows}
  ResFile = 'Points.R16';
{$else}
  ResFile = 'Points.R32';
{$endif}
procedure TForm1.Loaded;
begin
  inherited Loaded;
  PointList := TPointList.Create(Self);
  Pt := TPointData.CreateXY(PointList, 10, 10);
  Pt := TPointData.CreateXY(PointList, 366, 10);
  Pt := TPointData.CreateXY(PointList, 366, 191);
  Pt := TPointData.CreateXY(PointList, 10, 191);
  Pt := TPointData.CreateXY(PointList, 10, 10);
  WriteComponentResFile(ResFile, PointList);
  MessageDlg('Job done!', mtInformation, [mbOk], 0);
  Application.Terminate;
end;
```

➤ *Listing 8*

```
...
{$ifdef Windows}
  {$R Points.R16}
{$else}
  {$R Points.R32}
{$endif}
...
procedure TForm1.FormCreate(Sender: TObject);
begin
  PointList := ReadComponentRes(TPointList.ClassName, nil) as TPointList;
end;
...
initialization
  Randomize;
  RegisterClasses([TPointList, TPointData]);
end.
```

➤ *Listing 9*

manually edited and changed back to a binary file with:

```
CONVERT POINTS.TXT
REN POINTS.DFM POINTS.R16
```

Delphi 2

And that should have been that... but as I found to my horror when I checked all these examples in Delphi 2, several of them didn't work. STRM3, STRM4, STRM5 and SETUP failed to store any points in their streams. The common factor between these applications is that they all store the `TPointList` on the stream and expect all its owned components to be saved. In Delphi 1 this was the case because the `TWriter` object stores the root component's owned components, providing they have no parent to do it for them. It finds that out by calling the `HasParent` function, which by default returns `False`.

In Delphi 2 the `TWriter` object does not do this. Drat. However, in both versions the `TWriter` does give

```
object TPointList
  object TPointData
    X = 10
    Y = 10
  end
  object TPointData
    X = 366
    Y = 10
  end
  object TPointData
    X = 366
    Y = 191
  end
  object TPointData
    X = 10
    Y = 191
  end
  object TPointData
    X = 10
    Y = 10
  end
end
```

➤ *Listing 10*

a component an opportunity to write all (or some, or none of) the components it owns by calling one of the component's methods. Unfortunately for us, the method, and approach to its calling differs between versions. Apparently, whilst adding the support for

```
TPointList = class(TComponent)
  protected
{$ifdef Windows}
    procedure WriteComponents(Writer: TWriter); override;
{$else}
    procedure GetChildren(Proc: TGetChildProc); override;
{$endif}
  end;
TPointData = class(TComponent)
  ...
{$ifdef Windows}
  protected
    function HasParent: Boolean; override;
{$endif}
  ...
end;
...
{$ifdef Windows}
procedure TPointList.WriteComponents(Writer: TWriter);
var Loop: Integer;
begin
  { inherited version does nothing - no need to call it }
  for Loop := 0 to ComponentCount - 1 do
    Writer.WriteComponent(Components[Loop]);
end;
{$else}
procedure TPointList.GetChildren(Proc: TGetChildProc);
var Loop: Integer;
begin
  { inherited version does nothing - no need to call it }
  for Loop := 0 to ComponentCount - 1 do
    Proc(Components[Loop]);
end;
{$endif}
...
{$ifdef Windows}
function TPointData.HasParent: Boolean;
begin
  Result := True;
end;
{$endif}
...
```

➤ *Listing 11*

```
procedure TForm1.AddToList(Child: TComponent);
begin
  with Child do
    Listbox1.Items.Add(Format('%s: %s (%s)',
      [Name, ClassName, ClassParent.ClassName]));
end;
procedure TForm1.Button1Click(Sender: TObject);
var Loop: Integer;
begin
  Listbox1.Items.Clear;
{$ifdef Win32}
  GetChildren(AddToList);
{$else}
  for Loop := 0 to ComponentCount - 1 do
    AddToList(Components[Loop]);
{$endif}
end;
```

➤ *Above: Listing 12*

➤ *Below: Listing 13*

```
procedure FileCopy(const InFile, OutFile: String);
var InStream, OutStream: TFileStream;
begin
  InStream := TFileStream.Create(InFile, fmOpenRead + fmShareDenyWrite);
  try
    OutStream := TFileStream.Create(OutFile, fmCreate);
    try
      OutStream.CopyFrom(InStream, 0);
      FileSetDate(OutStream.Handle, FileGetDate(InStream.Handle));
    finally
      OutStream.Free;
    end;
  finally
    InStream.Free;
  end;
end;
```

inherited forms, Borland made a few architectural changes to the streaming. Sounds like a job for conditional compilation to me. The version-proof way of writing a component, and getting all the owned components to be written to a stream, is exemplified by the code in Listing 11. All the failing examples have been modified to include the various changes for the files on this month's disk.

Firstly, in 16-bit, the `HasResult` virtual method (of the owned component) needs overriding to return `True`, to suggest that the Delphi 1 `TWriter` does not write them out (we'd wind up with two copies at the end otherwise).

Secondly, in 16-bit, the owner needs a `WriteComponents` method. This is passed a `TWriter`, and the component can call the writer's `WriteComponent` method for any components that need streaming.

Lastly, in 32-bit, the owner needs a `GetChildren` method. This is passed a parameter of a procedural type, ie a reference to a `TWriter` method. For each component that needs streaming, you call that method and pass the component as a parameter.

This change from using `WriteComponents` to `GetChildren` is a good idea. It provides a general iterator. You can iterate through all the components that the owner is interested in and have any old routine of yours called for each one, with the component passed as a parameter. The only restriction is that your routine must be some procedure method that takes only one parameter: a component.

An example project that shows this is ITERATE.DPR. When `Button1` is pushed, `Listbox1` is filled with information about each component on the form, including its name, class and ancestor. Listing 12 shows the 32-bit and 16-bit ways of achieving this.

File Copying

Before finishing off, there is an aspect of stream usage I couldn't squeeze into last issue – that of using streams to copy files.

Two file streams in combination can be used to copy a file, as shown

in Listing 13. Note that the code takes advantage of the fact that `CopyFrom` takes a zero as the byte count parameter to mean "copy the whole stream."

This compares with the often used way of doing it with un-typed file variables and `BlockRead`, as in Listing 14.

Next Time

In the next issue we will turn our attention to writing text file device drivers. Never heard of them before? Well you can find out all about them next month.

Brian Long is a freelance Delphi consultant and trainer based in the UK. He is available for bookings and can be contacted by email on 76004.3437@compuserve.com

Copyright ©1996 Brian Long
All rights reserved.

```
procedure FileCopy(const InFileName, OutFileName: String);
const
  BufSize = 8 * 4096; { 32kb }
type
  PBuffer = ^TBuffer;
  TBuffer = array[1..BufSize] of Byte;
var
  Size: Cardinal;
  Buffer: PBuffer;
  InFile, OutFile: File;
begin
  if InFileName = OutFileName then
    raise EInOutError.Create('File cannot be copied onto itself')
  else begin
    Buffer := nil;
    Assign(InFile, InFileName);
    Reset(InFile, 1);
    try
      Assign(OutFile, OutFileName);
      Rewrite(OutFile, 1);
      try
        New(Buffer);
        repeat
          BlockRead(InFile, Buffer^, BufSize, Size);
          BlockWrite(OutFile, Buffer^, Size)
        until Size < BufSize;
        FileSetDate(TFileRec(OutFile).Handle,
          FileGetDate(TFileRec(InFile).Handle));
      finally
        if Buffer <> nil then
          Dispose(Buffer);
        CloseFile(OutFile)
      end;
    finally
      CloseFile(InFile);
    end
  end
end;
```

➤ *Listing 14*