File Handling 4: Streaming Objects

by Brian Long

Having looked at file variables and file handles in previous articles, we will now turn our attention to streams, with a slight digression on our way through. These possibly unfamiliar items are used by Delphi developers day in and day out under the guise of form files, but more of that later...

Stream Definition

A stream is intended to be a generic interface to any storage medium. The storage medium could be, amongst other possibilities, a block of memory, a disk file, a database BLOB field or a Windows resource file. In Delphi there are several stream types, all unsurprisingly represented by objects. They are all derived from the abstract class (ie useless as a class for instantiating objects, but useful as a basis for deriving more specific classes) `TStream`.

Basic Stream Capabilities

Any Delphi stream object has at least two properties. The current position within the stream is given by the `Position` property and the size of the stream is given by `Size`. To move to a different position in the stream there is a method called `Seek` (rather similar to `FileSeek` as used with file handles). This takes an offset to move by and another parameter indicating where to move from. This can be one of the symbols shown in Table 1. If `Seek` is successful, it returns the new position in the stream.

To write to the stream, you call the `Write` method, which takes an untyped `const` parameter that acts as an information buffer, and a number of bytes to write. It will return the number of bytes written. The implication of the parameter being untyped is that any data item can be passed.

Symbol	Implication for the Offset Parameter
soFromBeginning	Relative to the beginning of the stream
soFromCurrent	Relative to the current stream position
soFromEnd	Relative to the end of the stream

➤ *Table 1: Symbols for the Seek method*

Stream class	Comments
THandleStream	Takes an existing open file handle and allows you to access the file as a stream.
TFileStream	Takes a filename and a file access mode and treats the file as a stream. Based on a `THandleStream`.
TBlobStream	Takes a BLOB field and a BLOB access mode (`bmRead`, `bmWrite` or `bmReadWrite`). Used to manipulate BLOB field data.
TMemoryStream	Treats a block of memory as a stream.
TResourceStream	New in Delphi 2. Takes an executable's instance handle, a resource name or number and a resource type and lets you access a Windows resource as a stream.

➤ *Table 2: Stream types*

The implementation of a `const` or `var` parameter where a variable is passed (a pass by reference parameter) is that the *address* of the actual argument is passed (transparently to the programmer). If there is no type supplied in the parameter list of the subroutine definition then any variable will be accepted and its address gets passed along to the routine. The `Read` method takes an untyped `var` parameter and a byte count, and returns the number of bytes read.

In addition to this basic set of functionality, a stream object also has a `CopyFrom` method which knows how to read data from another stream, and also methods to write and read components to and from itself. The `WriteComponent` and `ReadComponent` methods are complemented by `WriteComponentRes` and `ReadComponentRes`, which write and read a standard Windows resource file header before the component respectively. These can be used to manufacture custom resource files filled with components that can be linked into your executable.

Types Of Streams

Delphi 1.0x offers a number of specific stream types, and Delphi 2 adds another one to the list, as shown in Table 2.

A `THandleStream` (and consequently a `TFileStream`) have a read-only `Handle` property to surface the file handle. When creating a `TFileStream` object, you pass the file name to the constructor and also a file access mode. These modes were discussed in the last issue, but there is an additional one available. If you wish your file stream to represent a newly created file, then you can use the `fmCreate` mode and it will endeavour to honour that request.

A TMemoryStream has a Memory property which returns a pointer to the beginning of the memory block. To start with it has no memory allocated and so Memory will be nil. You can make a memory stream as large as you like using the SetSize method; however, be warned that this disposes of any currently allocated memory through a call to its Clear method. Normally when writing to a memory stream, if it finds it hasn't got enough memory, it will allocate as much as is needed, rounding up to the nearest 8Kb. It is good practice to call SetSize straight after constructing a memory stream, passing a value that is as large as you need in the short term future, thus preventing lots of re-allocations.

A TResourceStream also has a Memory property, although this isn't used much. This object is designed for reading components from custom resources in your executable. We'll have a look at this later. To show a resource stream in operation (though not reading components) and also a TMemoryStream, consider the Tips & Tricks entry in Issue 5 (January 1996) that gave a TBitmap derivative which could read a 256 colour bitmap from a resource. It used several APIs to locate the bitmap resource, and then used a memory stream to communicate the data to a TBitmap object. The code looked like that in Listing 1.

We can make use of a Delphi 2 TResourceStream by changing the code to be as shown in Listing 2. The complete details of the two listings aren't important in this context, but notice that I have elected not to call the memory stream's SetSize method and also I am making use of the fact that CopyFrom understands that a second parameter of value zero means 'copy all of the stream'.

Both versions rely on the fact that a TBitmap object can set itself up from data on a stream, by using its LoadFromStream method. Many objects have a LoadFromStream and a corresponding SaveToStream method to allow storage or retrieval of their data to or from a stream. Those that do are:

```
TResBitmap = class(TBitmap)
public
  constructor Create(ID: PChar);
end;
...
constructor TResBitmap.Create(ID: PChar);
var
  HResInfo: THandle;
  MemHandle: THandle;
  Stream: TMemoryStream;
  ResPtr: PByte;
  ResSize: Longint;
const
  BMF: TBitmapFileHeader = (bfType: $4D42);
begin
  inherited Create;
  HResInfo := FindResource(HInstance, ID, RT_Bitmap);
  ResSize := SizeofResource(HInstance, HResInfo);
  MemHandle := LoadResource(HInstance, HResInfo);
  ResPtr := LockResource(MemHandle);
  Stream := TMemoryStream.Create;
  try
    Stream.Write(BMF, SizeOf(BMF));
    Stream.Write(ResPtr^, ResSize);
    FreeResource(MemHandle);
    Stream.Seek(0, soFromBeginning);
    LoadFromStream(Stream);
  finally
    Stream.Free;
  end;
end;
```

➤ *Listing 1*

```
TResBitmap2 = class(TBitmap)
public
  constructor Create(ID: PChar);
end;
...
constructor TResBitmap2.Create(ID: PChar);
var
  ResStream: TResourceStream;
  Stream: TMemoryStream;
const
  BMF: TBitmapFileHeader = (bfType: $4D42);
begin
  inherited Create;
  ResStream := TResourceStream.Create(HInstance, ID, RT_Bitmap);
  try
    Stream := TMemoryStream.Create;
    try
      Stream.Write(BMF, SizeOf(BMF));
      Stream.CopyFrom(ResStream, 0);
      Stream.Seek(0, soFromBeginning);
      LoadFromStream(Stream);
    finally
      Stream.Free;
    end;
  finally
    ResStream.Free;
  end;
end;
```

➤ *Listing 2*

- ➢ TGraphic and its descendants TIcon, TMetafile and TBitmap;
- ➢ TStrings and its descendent TStringList;
- ➢ TMemoryStream;
- ➢ TBlobField and its descendants TGraphicField and TMemoField;
- ➢ TOleContainer;
- ➢ TOutline;
- ➢ TTreeView (Delphi 2 only).

Streaming Data

Reading from and writing to a stream is reasonably straightforward for normal data types and matches up quite well with file handle operations. For example, to write a variable called I to a stream you would use:

```
MyStream.Write(I, SizeOf(I));
```

and to read it back in would require:

```
MyStream.Read(I, SizeOf(I));
```

If you have delved into pointers and, for example, you have a pointer to a Longint called P, you could use:

```
MyStream.Write(
  P^, SizeOf(Longint));
...
MyStream.Read(
  P^, SizeOf(Longint));
```

NAMES.EXE Version 4

The NAMES.DPR "database" project which went through three versions in the previous issue gets another rewrite this time round. When we left it, version 3 was using a file handle. This changes in NAMES4.DPR to be a TFileStream. The TDataFile constructor has the job of either opening the data file, or creating and then opening the data file. Using a file stream, the code looks like that shown in Listing 3. The destructor simply calls FDataFile.Free. The size of the file in terms of data records, as returned by GetCount, can be implemented as:

```
Result := FDataFile.Size div
  SizeOf(TDataRec);
```

We can find the current position in GetCurrent by using Seek, as we did last time around:

```
Result := FDataFile.Seek(
  0, soFromCurrent);
if Result > -1 then
  Result := Result div
    SizeOf(TDataRec);
```

and go to a requested record similarly:

```
FDataFile.Seek(
  RecNo * SizeOf(TDataRec),
  soFromBeginning);
```

The record reading and writing operations are also pretty much the same as before. The full implementation is included on the disk with this issue in the unit NAMES4U2.PAS.

```
constructor TDataFile.Create;
begin
  { Make current directory where EXE file is, just in case }
  ChDir(ExtractFilePath(Application.ExeName));
  { Make file if it ain't there }
  if not FileExists(FileName) then begin
    { We don't need a try..finally block here cos if the file creation
      fails, the constructor raises an exception which causes the object
      to be freed }
    FDataFile := TFileStream.Create(FileName, fmCreate);
    FDataFile.Free;
  end;
  FDataFile := TFileStream.Create(
                FileName, fmOpenReadWrite or fmShareDenyNone);
end;
```

➤ *Listing 3*

```
TPointData = class
  public
    X, Y: Word;
    constructor CreateXY(AX, AY: Word);
    procedure SwapXY;
    procedure LoadFromStream(Stream: TStream); virtual;
    procedure SaveToStream(Stream: TStream); virtual;
  end;
...
procedure TPointData.LoadFromStream(Stream: TStream);
begin
  Stream.Read(X, SizeOf(X));
  Stream.Read(Y, SizeOf(Y));
end;
procedure TPointData.SaveToStream(Stream: TStream);
begin
  Stream.Write(X, SizeOf(X));
  Stream.Write(Y, SizeOf(Y));
end;
```

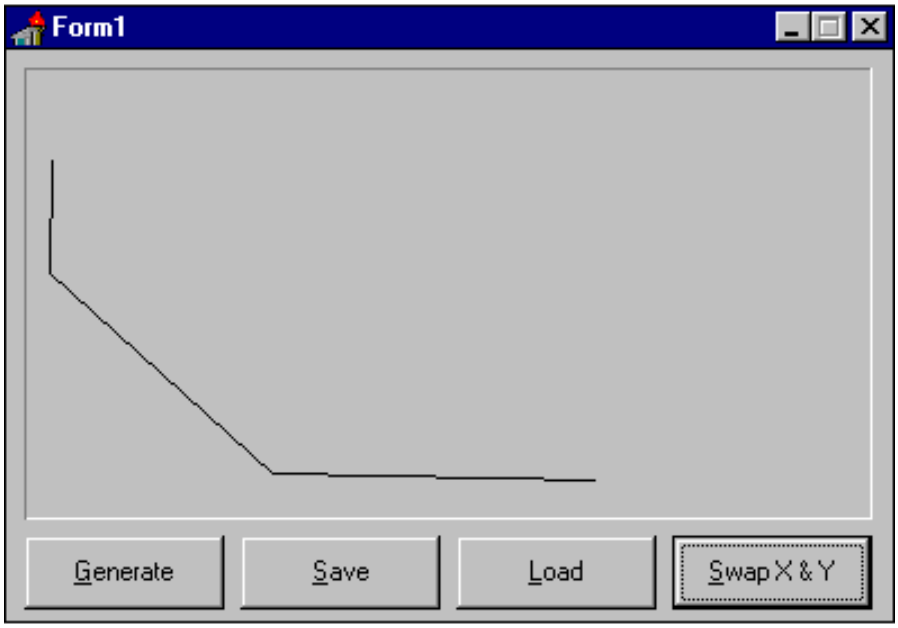
➤ *Listing 4*

Making Objects Streamable

If you are interested in streaming objects then things get a little more involved. Objects do not inherently know how to stream themselves. Objects that do know how are termed *persistent* objects – their data can potentially live on after a program has finished. There is a type called TPersistent in the VCL that is intended to act as a basis for all persistent objects – all objects that can potentially be streamed. Unfortunately, all the VCL-supplied mechanics for reading and writing persistent objects operate solely on objects based on a descendent of TPersistent called TComponent. So components can be streamed in a standard and well-supported way, but other objects can't. Let's look into how we can get any arbitrary object into and out of a stream.

What we need to do rather depends on what we want to achieve. If we want a stream filled with the data of many objects of the same class, then we can simply add our own LoadFromStream and SaveToStream methods directly to that class. Here's a simple example to demonstrate this, which makes use of a TList populated with TPointData objects, where TPointData merely holds X and Y co-ordinates. To make it have a point (pun not intended) it has a method to swap the X and Y co-ordinates and a choice of constructors. The normal constructor is inherited from type TObject and, if it is called, X and Y will be left with their default values of zero (all object data fields are initialised with zeros, a very handy feature). The alternative constructor CreateXY takes two parameters and sets X and Y with those values.

Our TPointData class will not be derived from type TPersistent since we won't be using the VCL-supplied streaming mechanism (we don't have TComponent as an ancestor, so we can't use it). This example is supplied in the project STRM1.DPR and the interesting



➤ *Figure 1*

code is in STRM1U.PAS. The object's definition and the stream access methods are shown in Listing 4.

The program that uses this class looks like Figure 1. The `Generate` button makes a random number of `TPointData` objects and stores them in a `TList` object called `PointList`. The `Save` button iterates through the list and calls the `SaveToStream` method for each `PointData` object, then empties the list. The `Load` button reads through the stream, creating an object and calling its `LoadFromStream` method until the end of the stream is reached. Lastly, the `Swap` button will iterate through the list and call the objects' `SwapXY` methods. Whenever points are made or altered, they are drawn in a paint box on the form. The useful code is shown in Listing 5.

Adding the `SaveToStream` and `LoadFromStream` methods to an object allows its data to be streamed and this approach is just as applicable if you have a set number of custom objects of *different* types that need to be streamed in a particular order.

However, this is not really all that object streaming can be. So far, when reading from the stream we need to know beforehand what object is next in the stream, so that we can construct one, and then call one of its methods to get it to read its own data. Object streaming can be much more flexible. Delphi does support this extra flexibility but only for classes derived from type `TComponent`.

Design-Time Stream Requirements

One of the main uses of an object stream is to remove repetitive functionality from a program. If your program uses a set of objects each time it runs, and when it starts it initialises those objects, then your program is wasting a certain amount of time each run doing this initialisation. Much better would be to use another small utility

➤ *Listing 5*

```
procedure TForm1.ClearPoints;
begin
  while PointList.Count > 0 do begin
    TPointData(PointList[0]).Free;
    PointList.Delete(0);
  end;
end;
procedure TForm1.PaintBox1Paint(Sender: TObject);
begin
  for Loop := 0 to PointList.Count - 1 do begin
    Pt := TPointData(PointList.Items[Loop]);
    if Loop = 0 then
      PaintBox1.Canvas.MoveTo(Pt.X, Pt.Y)
    else
      PaintBox1.Canvas.LineTo(Pt.X, Pt.Y)
  end;
end;
procedure TForm1.MakeBtnClick(Sender: TObject);
begin
  ClearPoints;
  for Loop := 1 to 20 do begin
    Pt := TPointData.CreateXY(Random(PaintBox1.Width),
      Random(PaintBox1.Height));
    PointList.Add(Pt);
    PaintBox1.Invalidate;
  end;
end;
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]);
      Pt.SaveToStream(Stream);
    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;
      Pt.LoadFromStream(Stream);
      PointList.Add(Pt);
    end;
  finally
    Stream.Free;
  end;
  PaintBox1.Invalidate;
end;
procedure TForm1.SwapBtnClick(Sender: TObject);
begin
  for Loop := 0 to PointList.Count - 1 do
    TPointData(PointList.Items[Loop]).SwapXY;
  PaintBox1.Invalidate;
end;
```

program to set up the objects and write them to a stream and then your main program can just read the stream in each time it runs. This is exactly what Delphi does. It makes an object stream (a form file) to represent each of your forms at design time (when your program is not running), and then rather than you having to set up all your components each time you run your program, the streams are read in as needed when forms are created and all the objects on them are automatically created, with all their properties set up.

What is it about `TPersistent` that makes components inherently streamable? If you've ever investigated what goes into a form file, you'll have found that it is a bunch of property names and their values. How does Delphi know what to write into a form file? Well it's all to do with run-time type information (RTTI). The story goes like this.

Delphi writes property values into a form that appear on the Object Inspector. As you will know, properties appear on the Object Inspector only if they are declared in the `published` section of an object and an object can only have a `published` section if the compiler has been told to generate RTTI for it, or one of its ancestors. In Delphi 1, the undocumented compiler directive `$M+` tells the compiler to generate RTTI. `TObject` is compiled in a `$M-` state (if you make a class inherited from `TObject` and put a `published` section in, you will get *Error 200: PUBLISHED not allowed in this class*). `TPersistent` is compiled in the `$M+` state. This means that anything derived from `TPersistent` can have a `published` section and RTTI will be available for all entries in that section.

In Delphi 2, a compiler bug means that this directive is *always* on, so *any* object can now have a `published` section, not just those derived from `TPersistent`.

The Object Inspector makes use of RTTI to show the properties and allow their values to be changed. The form designer makes use of the information to save the properties to a form file and to read them back in again. This functionality is available to you as a programmer (as we will see next time).

The RTTI can be examined and manipulated using the functionality implemented in the `TypInfo` unit (undocumented but the interface section is supplied in the DOC directory in the file TYPINFO.INT). There's enough material in that and `TObject` itself to make another article on RTTI investigation, but next time we'll sate our appetites with more on streaming components, before coming back online with text file device drivers.

And Finally...

One or two comments were made about the code from last month's article. Thanks to the readers who let us know about the problems, which I'm pleased to say I have fixes for!

As I've just completely run out of space, the updates are included on the disk in the README.TXT file in the FILES directory, along with new source files too.

Brian Long is an independent consultant and trainer specialising in Delphi. His email address is 76004.3437@compuserve.com