How to dynamically add data to an executable file (part 4 of 5)

Random payload access

Random payload access

The "Delphi way" of providing random access to data is to derive a class from *TStream* and to override its abstract methods – and this is what we will do here. Our new class will be called *TPayloadStream*. It will detect payload data and provide read write random access to it.

Not only does this approach provide random access but it also has the added advantage of hiding the details of how the payload is implemented from the user of the class. All the user sees is the familiar *TStream* interface while all the gory details are hidden in *TPayloadStream*'s implementation.

Listing 9 shows the definition of the new class, along with an enumeration – *TPayloadOpenMode* – that is used to determine whether a payload stream object is to read or write the payload data. Note that in addition to overriding *TStream*'s abstract methods, *TPayloadStream* also overrides the virtual *SetSize* method to enable the user to change the size of the payload. This is necessary because, by default, *SetSize* does nothing.

```
type
  TPayloadOpenMode = (
    pomRead,     // read mode
    pomWrite     // write (append) mode
  );

  TPayloadStream = class(TStream)
  private
    fMode: TPayloadOpenMode;  // stream open mode
    fOldFileMode: Integer;    // preserves old file mode
    fFile: File;              // handle to exec file
    fDataStart: Integer;      // start of payload data in file
    fDataSize: Integer;       // size of payload
  public
    constructor Create(const FileName: string;
      const Mode: TPayloadOpenMode);
      // opens payload of file in given open mode
    destructor Destroy; override;
      // close file, updating data in write mode
    function Seek(Offset: LongInt;
      Origin: Word): LongInt; override;
      // moves to specified position in payload
    procedure SetSize(NewSize: LongInt); override;
      // sets size of payload in write mode only
    function Read(var Buffer;
      Count: LongInt): LongInt; override;
      // Reads count bytes from payload
    function Write(const Buffer;
      Count: LongInt): LongInt; override;
      // Writes count bytes to payload in write mode only
  end;
```

Listing 9

The public methods of *TPayloadStream* are:

- *Create* – Creates a *TPayloadStream* and opens the named file either in read or write mode.
- *Destroy* – Updates the payload footer, closes the file and destroys the object.
- *Seek* – Moves the stream's pointer to the specified position in the payload data, ensuring that the pointer remains within the payload.

- *SetSize* – Sets the size of the payload in write mode only. Raises an exception when used in read mode. Note that setting the size to zero will remove the payload and the associated footer record.
- *Read* – Attempts to read a specified number of bytes into a buffer. If there is insufficient data in the payload just the remaining bytes are read.
- *Write* – Writes a specified number of bytes from a buffer to the payload, extending the payload if required. Works only in write mode – an exception is raised in read mode.

The class also uses the following private fields:

- *fMode* – Records whether the stream is open for reading or writing.
- *fOldFileMode* – Preserves the current Pascal file mode.
- *fFile* – Pascal file descriptor that records the details of an open file.
- *fDataStart* – Offset of the start of payload data from the start of the executable file.
- *fDataSize* – Size of payload data.

We begin our review of the class's implementation by examining *Listing 10* which shows the constructor and destructor. Once again we are using classic Pascal un-typed files to perform the underlying physical access to the executable file. However this could easily be changed to use some other file access techniques.

```
constructor TPayloadStream.Create(const FileName: string;
  const Mode: TPayloadOpenMode);
var
  Footer: TPayloadFooter; // footer record for payload data
begin
  inherited Create;
  // Open file, saving current mode
  fMode := Mode;
  fOldFileMode := FileMode;
  AssignFile(fFile, FileName);
  case fMode of
    pomRead: FileMode := 0;
    pomWrite: FileMode := 2;
  end;
  Reset(fFile, 1);
  // Check for existing payload
  if ReadFooter(fFile, Footer) then
  begin
    // We have payload: record start and size of data
    fDataStart := Footer.ExeSize;
    fDataSize := Footer.DataSize;
  end
  else
  begin
    // There is no existing payload: start is end of file
    fDataStart := FileSize(fFile);
    fDataSize := 0;
  end;
  // Set required file position per mode
  case fMode of
    pomRead: System.Seek(fFile, fDataStart);
    pomWrite: System.Seek(fFile, fDataStart + fDataSize);
  end;
end;

destructor TPayloadStream.Destroy;
var
  Footer: TPayloadFooter; // payload footer record
begin
  if fMode = pomWrite then
  begin
    // We're in write mode: we need to update footer
    if fDataSize > 0 then
    begin
      // We have payload, so need a footer record
      InitFooter(Footer);
      Footer.ExeSize := fDataStart;
      Footer.DataSize := fDataSize;
      System.Seek(fFile, fDataStart + fDataSize);
```

```
      Truncate(fFile);
      BlockWrite(fFile, Footer, SizeOf(Footer));
    end
    else
    begin
      // No payload => no footer
      System.Seek(fFile, fDataStart);
      Truncate(fFile);
    end;
  end;
  // Close file and restore old file mode
  CloseFile(fFile);
  FileMode := fOldFileMode;
  inherited;
end;
```

Listing 10

In the constructor the first thing we do is to record the open mode and then open the underlying file in the required mode. Next we try to read a payload footer record, using the *ReadFooter* function we developed in *Listing 3*.

If we have a footer we a get the start of the payload data (*fDataStart*) and the size of the payload (*fDataSize*) from the footer's *ExeSize* and *DataSize* fields respectively. If there is no footer record we have no payload so we set *fDataStart* to refer to just beyond the end of the file and set *fDataSize* to zero.

Setting *fDataStart*

fDataStart is the same as the size of the executable file because payloads always start immediately after the executable code.

Finally the constructor sets the file pointer according to the file mode – in read mode we set it to the start of the payload while in write mode we set it to the end.

In the destructor we proceed differently according to whether we a are in read mode or write mode:

In read mode all there is to do is to close the file and restore the previous Pascal file mode.

In write mode we first check if we actually have a payload (`fDataSize > 0`). If so we create a footer record using the *InitFooter* routine we defined in *Listing 2* and record the payload size and start position in the record. We then seek to the end of the new payload data, truncate any data that falls beyond its end (as will be the case when the data size has shrunk), then write the footer. If there is no payload data we truncate the file at the end of the executable code. Finally we close the file and restore the file mode.

That completes the discussion of the class constructor and destructor. Let us now consider how we override the abstract *Seek*, *Read* and *Write* methods. *Listing 11* has the details:

```
function TPayloadStream.Seek(Offset: Integer;
  Origin: Word): LongInt;
begin
  // Calculate position in payload after move
  // (this is result value)
  case Origin of
    soFromBeginning:
      // Moving from start of stream: ignore -ve offsets
      if Offset >= 0 then Result := Offset
      else Result := 0;
    soFromEnd:
      // Moving from end of stream: ignore +ve offsets
      if Offset <= 0 then Result := fDataSize + Offset
      else Result := fDataSize;
    else  // soFromCurrent and other values
      // Moving from current position
      Result := FilePos(fFile) - fDataStart + Offset;
  end;
  // Result must be within payload: make sure it is
  if Result < 0 then Result := 0;
  if Result > fDataSize then Result := fDataSize;
  // Perform actual seek in underlying file
  System.Seek(fFile, fDataStart + Result);
end;

function TPayloadStream.Read(var Buffer;
  Count: Integer): LongInt;
var
```

```
  BytesRead: Integer;   // number of bytes read
  AvailBytes: Integer;  // number of bytes left in stream
begin
  // Work out how many bytes we can read
  AvailBytes := fDataSize - Position;
  if AvailBytes < Count then
    Count := AvailBytes;
  // Read data from file and return bytes read
  BlockRead(fFile, Buffer, Count, BytesRead);
  Result := BytesRead;
end;

function TPayloadStream.Write(const Buffer;
  Count: Integer): LongInt;
var
  BytesWritten: Integer;  // number of bytes written
  Pos: Integer;           // position in stream
begin
  // Check in write mode
  if fMode <> pomWrite then
    raise EPayloadStream.Create(
      'TPayloadStream can''t write in read mode.');
  // Write the data, recording bytes read
  BlockWrite(fFile, Buffer, Count, BytesWritten);
  Result := BytesWritten;
  // Check if stream has grown
  Pos := FilePos(fFile);
  if Pos - fDataStart > fDataSize then
    fDataSize := Pos - fDataStart;
end;
```

Listing 11

Seek is the most complicated of the three methods. This is because the *FilePos* and *Seek* routines (from the `System` unit) that we use to get and set the file pointer operate on the whole file, while our stream positions must be relative to the start of the payload data. We must also ensure that the file pointer cannot be set outside the payload data. The *case* statement contains the code that calculates the required offset within the payload, depending on the seek origin. The two lines following the *case* statement constrain the offset within the payload data. Finally we perform the actual seek operation on the underlying file, offset from the start of the payload data. The method returns the new offset relative to the payload.

The *Read* method must ensure that the read falls wholly within the payload data. We can't assume that all the remaining bytes in the stream can be read, because the payload may be followed by a footer record that is not part of the data. Therefore we calculate the number of available bytes by subtracting the current position in the payload from the size of the payload data. If there is insufficient data to meet the request the number of bytes to be read is reduced to the number of available bytes.

Note that *Read* uses *TStream*'s *Position* property to get the current position in the payload data. This property calls the *Seek* method which, as we have seen, ensures that the position returned falls within the payload data.

Write is quite simple – we just check we are in write mode and output the data to the underlying file at the current position if so. The number of bytes written is returned. The only complication is that we must check if the write operation took us beyond the end of the current data and record the new data size if so. Should the stream be in read mode *Write* raises an exception.

All that remains to do now is to override the *SetSize* method. *Listing 12* provides the implementation.

```
procedure TPayloadStream.SetSize(NewSize: Integer);
var
  Pos: Integer; // current position in stream
begin
  // Check for write mode
  if fMode <> pomWrite then
    raise EPayloadStream.Create(
      'TPayloadStream can''t change size in read mode.');
  // Update size, adjusting position if required
  if NewSize < fDataSize then
  begin
    Pos := Position;
    fDataSize := NewSize;
```

```
    if Pos > fDataSize then
      Position := fDataSize;
  end;
end;
```

Listing 12

Obviously, we can't change the stream size in read mode, so we raise an exception in this case. In write mode we only record the new size if it is less than the current payload size. In this case we must also check if the current stream position falls beyond the end of the reduced payload and move the position to the end of the truncated data if so. The *Position* property is used to get and set the stream position. As noted earlier, this property calls our overridden *Seek* method.

This completes our presentation of the *TPayloadStream* class. In the *final part* of the article you can download some demo code and provide feedback to the author.
