How to dynamically add data to an executable file (part 3 of 5)

Payload management class

Payload management class

In this section we will develop a class that lets us manage payloads. Let us first define the requirements of the class. They are:

- To check if an executable file contains a payload.
- To find the size of the payload data.
- To extract the payload from a file into a suitably sized buffer.
- To delete a payload from a file.
- To store payload data in a file.

The class is declared as follows:

```
type
  TPayload = class(TObject)
  private
    fFileName: string;
    fOldFileMode: Integer;
    fFile: File;
    procedure Open(Mode: Integer);
    procedure Close;
  public
    constructor Create(const FileName: string);
    function HasPayload: Boolean;
    function PayloadSize: Integer;
    procedure SetPayload(const Data; const DataSize: Integer);
    procedure GetPayload(var Data);
    procedure RemovePayload;
  end;
```

Listing 4

The public methods are:

- *Create* – Creates an object to work on a named file.
- *HasPayload* – Returns true if the file contains a payload.
- *PayloadSize* – Returns the size of the payload. This information is required when allocating a buffer into which payload data can be read using *GetPayload*.
- *SetPayload* – Copies a specified number of bytes from a buffer and stores them as a payload at the end of the file. Overwrites any existing payload.
- *GetPayload* – Copies the file's payload into a given buffer. The buffer must be large enough to store all the payload. The required buffer size is given by *PayloadSize*.
- *RemovePayload* – Deletes any payload from the file and removes the footer record. This method restores file to its original condition.

In addition there are two private helper methods:

- *Open* – Opens the file in a specified mode.
- *Close* – Closes the file and restores the original file mode.

The class also has three fields:

- *fFileName* – Stores the name of the file we are manipulating.

- *fOldFileMode* – Preserves the current Pascal file mode.
- *fFile* – Pascal file descriptor that records the details of an open file.

As can be seen from the discussion of the fields we will be using standard un-typed Pascal file routines to manipulate the file.

We will discuss the implementation of the class is several chunks. We begin in *Listing 5* where we look at the constructor, some required constants and the two private helper methods.

```
const
  // Untyped file open modes
  cReadOnlyMode = 0;
  cReadWriteMode = 2;

constructor TPayload.Create(const FileName: string);
begin
  // create object and record name of payload file
  inherited Create;
  fFileName := FileName;
end;

procedure TPayload.Open(Mode: Integer);
begin
  // open file with given mode, recording current one
  fOldFileMode := FileMode;
  AssignFile(fFile, fFileName);
  FileMode := Mode;
  Reset(fFile, 1);
end;

procedure TPayload.Close;
begin
  // close file and restore previous file mode
  CloseFile(fFile);
  FileMode := fOldFileMode;
end;
```

Listing 5

The two constants define the two Pascal file modes we will require. The constructor simply records the name of the file associated with the class. The *Open* method first stores the current file mode then opens the file using the required file mode. Finally, *Close* closes the file and restores the original file mode.

We next consider the two methods that provide information about a file's payload – *PayloadSize* and *HasPayload*:

```
function TPayload.PayloadSize: Integer;
var
  Footer: TPayloadFooter;
begin
  // assume no data
  Result := 0;
  // open file
  Open(cReadOnlyMode);
  try
    // read footer and if valid return data size
    if ReadFooter(fFile, Footer) then
      Result := Footer.DataSize;
  finally
    Close;
  end;
end;

function TPayload.HasPayload: Boolean;
begin
  // we have a payload if size is greater than 0
  Result := PayloadSize > 0;
end;
```

Listing 6

The only method here of any substance is *PayloadSize*. We first assume a payload size of zero in case there is no payload. Next we open the file in read mode and attempt to read the footer. The *ReadFooter* helper routine is used to do this. If the footer is read successfully we get the size of the payload from the footer record's *DataSize* field. The file is then closed.

HasPayload simply calls *PayloadSize* and checks if the payload size it returns is greater than zero.

Now we move on to consider *GetPayload* which is described in *Listing 7*. This method's *Data* parameter is a data buffer which must have a size of at least *PayloadSize* bytes.

```
procedure TPayload.GetPayload(var Data);
var
  Footer: TPayloadFooter;
begin
  // open file as read only
  Open(cReadOnlyMode);
  try
    // read footer
    if ReadFooter(fFile, Footer)
      and (Footer.DataSize > 0) then
    begin
      // move to end of exe code and read data
      Seek(fFile, Footer.ExeSize);
      BlockRead(fFile, Data, Footer.DataSize);
    end;
  finally
    // close file
    Close;
  end;
end;
```

Listing 7

GetPayload opens the file in read only mode and tries to read the footer record. If we succeed in reading a footer *and* the payload contains data we move the file pointer to the start of the payload, then read the payload into the *Data* buffer. Note that we use the footer record's *ExeSize* field to perform the seek operation and the *DataSize* field to determine how many bytes to read. The method ends by closing the file.

Finally we examine the implementation of the two methods that modify the file – *RemovePayload* and *SetPayload*.

```
procedure TPayload.RemovePayload;
var
  PLSize: Integer;
  FileLen: Integer;
begin
  // get size of payload
  PLSize := PayloadSize;
  if PLSize > 0 then
  begin
    // we have payload: open file and get size
    Open(cReadWriteMode);
    FileLen := FileSize(fFile);
    try
      // seek to end of exec code and truncate file there
      Seek(fFile, FileLen - PLSize - SizeOf(TPayloadFooter));
      Truncate(fFile);
    finally
      Close;
    end;
  end;
end;

procedure TPayload.SetPayload(const Data;
  const DataSize: Integer);
var
  Footer: TPayloadFooter;
begin
  // remove any existing payload
  RemovePayload;
  if DataSize > 0 then
  begin
```

```
    // we have some data: open file for writing
    Open(cReadWriteMode);
    try
      // create a new footer with required data
      InitFooter(Footer);
      Footer.ExeSize := FileSize(fFile);
      Footer.DataSize := DataSize;
      // write data and footer at end of exe code
      Seek(fFile, Footer.ExeSize);
      BlockWrite(fFile, Data, DataSize);
      BlockWrite(fFile, Footer, SizeOf(Footer));
    finally
      Close;
    end;
  end;
end;
```

Listing 8

RemovePayload checks the existing payload's size and proceeds only if a payload is present. If so the file is opened for writing and it's size is noted. We then seek to the end of the executable part of the file and truncate it before closing the file. We have calculated the end of the executable section by deducting the payload size and the size of the footer record from the file length. We could also have read the footer and simply used the value of its *ExeSize* field.

SetPayload takes two parameters: a data buffer (*Data*) and the size of the buffer (*DataSize*). The method begins by using *RemovePayload* to remove any existing payload, ensuring that the file contains only the executable code. If the payload contains some data we open the file for writing. A new payload record is then initialized using the *InitFooter* helper routine, then sizes of both the executable file and of the new payload are stored in the record. Finally we append the payload data followed by the footer record to the file before closing it.

Now we have created the *TPayload* class it is easy to manipulate payload data. Unfortunately we must read and write the whole of the payload at once, which is not always convenient. An improvement would be to enable random access to the data. In the *next part* of this article we develop code to do this.
