Under Construction:
A Hex Viewer And Editor

by Bob Swart

The past few months have been about component management and workgroup management. But what about component development? We haven't seen a lot of those nice little components lately, have we? This time, I'm going to make it up to you with two new components for both versions of Delphi: a Hex Viewer and Editor, for unlimited filesizes...

A hexadecimal file viewer component (`TBHexViewer`) would have to work on files with a practically unlimited file size: that's the first problem we need to solve! Apart from that, a visual way to display the binary data, with a textual representation on the side, would be most welcome, to prepare for a hex viewer with edit or overwrite capabilities (`TBHexEditor`).

FileSize

Delphi 1.0's 16-bit `TMemo` component can only hold files up to about 32Kb in size. This is useful for small memos and a few hundred lines, but for really big files it's no good. Even Delphi 2.0 does not solve this problem, because the `TMemo` component is wrapped around the Windows edit control, which in turn is limited to 32Kb even on the supposedly "32-bits" Win95. Only when run on the true 32-bits operating system Windows NT do we finally get a `TMemo` that is capable of holding nearly 2Gb of text. And at this time, only on a small amount of Delphi users are running NT.

Since our file viewer needs to be able to view files of nearly unlimited size (I consider a filesize of 2Gb to be nearly unlimited – I don't even have the disk space to store a file of this size *[I've got 6.5Gb on my new machine gloats the Editor... <grin>]*), we cannot use a `TMemo` to derive our browsing component from. We need to do something special.

In order to be able to view files up to 2Gb in size, it is not practical to keep the entire file in memory while browsing. Nobody I know would have the required amount of memory available anyway (not on a personal computer). And while Windows NT, in theory, offers us the virtual memory needed to load a 2Gb file, it would only lead to a 2Gb swap file.

Instead, we need to use *views* on certain portions of the file. And while browsing through the file, we load a certain portion on demand. If we make sure the portions are big enough and loading them is quick enough, the end-user won't notice any difference whatsoever. This idea is not new, of course, but based on the old memory paging algorithm of operating systems such as UNIX.

Pages?

A file of 2Gb translates into 64 pages of 32Kb, 1024 pages of 2Kb, or 8K pages of 256 bytes. For a hex file viewer with a view window that holds 256 characters at a time we can set the page size to 256 bytes and have a maximum of 8K pages. Of course, file buffering can be used to make sure we don't actually have to read in a page of 256 bytes every time we scroll from page to page. We can define a type `TBlock` and use a property called `Block` to hold the data of the current page (block) of 256 bytes:

```
Const
  BlockLine = 16;
  BlockChar = 16;
  BlockSize = BlockLine *
    BlockChar; { 16x16 = 256 }
Type
  TBlock =
   Array[1..BlockSize] of Byte;
```

The page number, between 0 and 8K (or even higher), can be used to find the offset in the file where the next page starts (ie page 0 starts at offset 0, page 1 at offset 256, page 2 at offset 512, etc).

➤ *Listing 1*

```
unit first;
{$I+ will raise IO-exceptions when needed }
interface
uses
  WinTypes, WinProcs, SysUtils, Classes;
Const
  BlockLine = 16;
  BlockChar = 16;
  BlockSize = BlockLine * BlockChar; { 16 x 16 = 256 }
Type
  TBlock = Array[1..BlockSize] of Byte;
  TBFile = class(TComponent)
    private
      FFileName: TFileName;
      FFile: File;
      FOffset: LongInt; { 0, 256, 512, ... 2G }
      FBlock: TBlock; { data from FFile }
      FSize: Cardinal; { actual size of data in FBlock <= BlockSize }
    protected
      procedure SetFileName(AFileName: TFileName); virtual;
      procedure SetOffset(AnOffset: LongInt); virtual;
    public
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;
    published
      property FileName: TFileName read FFileName write SetFileName;
      property Offset: LongInt read FOffset write SetOffset default 0;
    end {TBFile};
```

The Engine

Our first attempt, called TBFile, is a component which consists of FileName and Offset properties, which together with a hidden field FFile define the contents of the FBlock field.

The type definition of the TBFile component is shown in Listing 1 (note that this is just a dummy component for now). The two properties FileName and Offset are used as the design time interface (in the Object Inspector for example), while the internal field FFile is used to hold a file pointer and FBlock contains the actual data.

In the constructor we need to set all fields to unused values, while the destructor must check to see if a filename is assigned (and hence if a file is open) to make sure to close the file before the TBFile object is destroyed. See Listing 2.

The FileName property reads from the hidden field FFileName and writes with help from a SetFileName method (Listing 3). This method not only assigns the filename to the FileName property, it also closes the previous file (if any) and opens up the FFile file field. It uses the FileMode flag $42 for this (which means Read/Write Deny None in sharing mode).

Note that Delphi 2 has a bug which ignores the sharing flags, so any file opened with this component compiled with Delphi 2 will be locked and not available for others. For a fix for this FileMode problem, please consult *Dr.Bob's Delphi Clinic* on the World Wide Web at http://www.pi.net/~drbob/

In case we can't open the file, an exception is raised and handled, after which we set the filename back to an empty ''. So, if at any time the filename is not empty we know the file is open.

Once we have an open file, we can read blocks from it. We do this by assigning values to the Offset variable (Listing 4).

Note that we use exceptions in a try-except block to check for I/O errors. Alternatively, we could specify {$I-} and check IOResult. The latter is a little bit faster, but would mix the algorithm with the error handling code itself.

Visual Hex

The non-visual component TBFile can read a file up to 2Gb in blocks of 256 bytes. However, the blocks are only kept in the hidden FBlock field and nothing useful is done with them. In fact, nothing indicates that the component is in fact reading the whole file. We now need to design and implement the component user interface.

Earlier in this column, we mentioned the sad fact that a TMemo component was limited to 32Kb of text. Now that we're limited to pages of only 256 (binary) characters, it's time to reconsider the TMemo component. We must realise, however, that not only do we need to display the hexadecimal values of the 256 byte block, we must also display the address (offset) in the

➤ *Listing 2*

```
constructor TBFile.Create(AOwner: TComponent);
var i: Integer;
begin
  inherited Create(AOwner);
  FFileName := '';
  FOffset := 0;
  FSize := 0
end {Create};
destructor TBFile.Destroy;
begin
  if FFileName <> '' then Close(FFile);
  inherited Destroy
end {Destroy};
```

➤ *Listing 3*

```
procedure TBFile.SetFileName(AFileName: TFileName);
begin
  if FFileName <> '' then begin
    FFileName := '';
    FOffset := 0;
    FSize := 0;
    System.Close(FFile)
  end;
  System.Assign(FFile,AFileName);
  try
    FileMode := $42; { read/write, deny-none }
    System.Reset(FFile,1);
    FFileName := AFileName { success! }
  except
    FFileName := ''
  end;
  Offset := 0
end {SetFileName};
```

➤ *Listing 4*

```
procedure TBFile.SetOffset(AnOffset: LongInt);
begin
  AnOffset := AnOffset AND NOT BlockLine; { skip lower bits }
  if (AnOffset <> FOffset) or (AnOffset = 0) or (FOffset = 0) then begin
    FOffset := AnOffset;
    FillChar(FBlock,SizeOf(FBlock),#0);
    if FFileName <> '' then
      try
        Seek(FFile,FOffset);
        BlockRead(FFile,FBlock,SizeOf(FBlock),FSize);
      except
        FOffset := 0;
        FSize := 0
      end
    else begin
      { FFileName = '' }
      FOffset := 0;
      FSize := 0
    end
  end
end {SetOffset};
```

file and format the output to a nice 16x16 display. Maybe a memo isn't so useful for this approach after all. We're looking for a kind of table, something like a StringGrid.

Let's consider a StringGrid with 17 rows and 18 columns (16x16 for the data, one extra row and column in for the captions and a column to display the data in text format). Using a fixed font and predefining the width of each column, we can come up with a nice display as shown in Figure 1.

So, we need to derive our TBFile from a TStringGrid class. The modified type definition of TBHexViewer, based on TBFile but derived from TStringGrid, is shown in Listing 5.

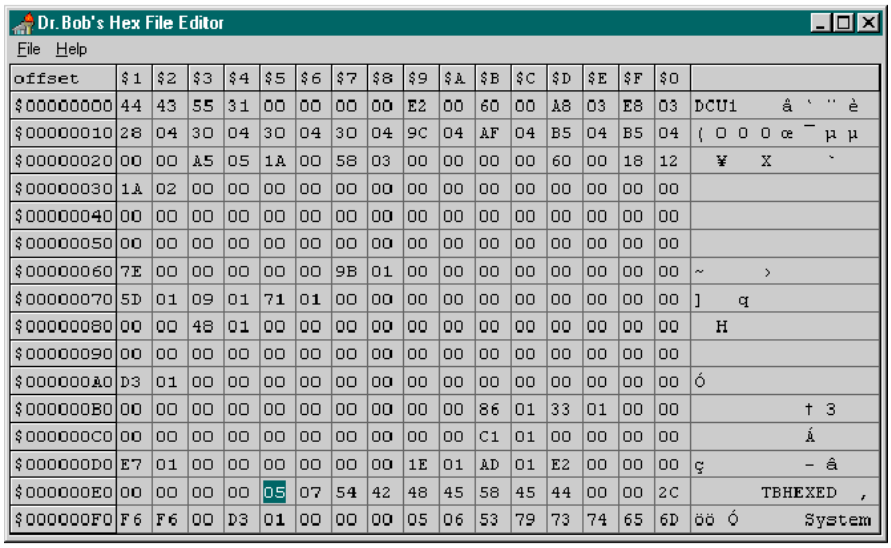
Note that all hidden fields are private (ie accessible only from within the same class or unit, and we don't put anything else in the unit of course), except for the FAbout field, since I know beforehand that I would like to get to this field from a derived class. And at this time it doesn't seem necessary to allow a derived class get to the other hidden fields.

The constructor of TBHexViewer needs to call the inherited constructor of the StringGrid, set the row and column count and sizes and set the labels for the headers (the hex values $1 through $0). See Listing 6.

The destructor is the same as the one we wrote for TBFile: just close the file if there is one open. The SetOffset, on the other hand, has changed somewhat. The first part is still the same and deals with reading in the new FBlock from the file (at the specified offset). The second part is new and consists of formatting the raw data in hex values and text representations in the Cells array-property of the StringGrid itself. See Listing 7.

We need to override the SelectCell method to indicate that column 17 (with the textual representation) cannot be selected:

```
function TBHexViewer.SelectCell(
  ACol, ARow: Longint): Boolean;
begin
  Result := inherited SelectCell(
    ACol,ARow) and (ACol <> 17)
end {SelectCell};
```



➤ *Figure 1*

```
TBHexViewer = class(TStringGrid)
  private
    FFile: File;
    FFileName: TFileName;
    FOffset: LongInt; { 0, 256, 512, ... 2G }
    FBlock: TBlock; { data from FFile }
    FSize: Cardinal; { actual size of data in FBlock }
    procedure KeyDown(var Key: Word; Shift: TShiftState); override;
  protected
    FAbout: String;
    procedure SetFileName(AFileName: TFileName); virtual;
    procedure SetOffset(AnOffset: LongInt); virtual;
    procedure SetSize(unused: Cardinal); { do nothing }
    procedure SetAbout(unused: String); { do nothing }
    function SelectCell(ACol, ARow: Longint): Boolean; override;
    { this function does *not* work when declared private... }
  public
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
  published
    property FileName: TFileName read FFileName write SetFileName;
    property Offset: LongInt read FOffset write SetOffset default 0;
    property Size: Cardinal read FSize write SetSize default 0;
    property About: String read FAbout write SetAbout;
  end {TBHexViewer};
```

➤ *Listing 5*

Finally, we need a way to browse, ie a way for the user to specify that the offsets need to be changed and the next page (or block) of the file needs to be shown. For this, we can look at the keyboard KeyDown event and simply react at the PgUp and PgDown events. In the case of a PgUp, we decrease the OffSet by the BlockSize (if possible), in the case of a PgDown we increase it by the same amount.

And we've seen before that a change to the value of the OffSet property will automatically load the new correct block, which will also set new values to the cell properties and will hence update the screen. See Listing 8.

Hex Edit

Viewing a file in hex mode is one thing. Editing the file is something else. For one thing, how do we insert or delete some bytes if the entire file is not kept in memory? Well, the techniques for this would take too long to explain here, so let's limit ourselves to a Hex Overwriter for now...

TBHexEditor

Listing 9 shows the definition and constructor for TBHexEditor. The constructor calls the inherited constructor of TBHexViewer and then sets a few properties to their special value, like the FChanged field (to False, nothing has changed so

```
constructor TBHexViewer.Create(
  AOwner: TComponent);
var i: Integer;
begin
  inherited Create(AOwner);
  FAbout :=
   'TBHexViewer (c) 1996 by Dr.Bob';
  ParentFont := False;
  Font.Name := 'Courier New';
  Font.Size := 10;
  Height := 342{+17};
  Width := 632;
  FFileName := '';
  FOffset := 0;
  FSize := 0;
  ScrollBars := ssNone;
  ColCount := 18;
  RowCount := 17;
  DefaultRowHeight := 19{+1};
  Cells[$0,0] := 'offset';
  Cells[$1,0] := '$1';
  Cells[$2,0] := '$2';
  Cells[$3,0] := '$3';
  Cells[$4,0] := '$4';
  Cells[$5,0] := '$5';
  Cells[$6,0] := '$6';
  Cells[$7,0] := '$7';
  Cells[$8,0] := '$8';
  Cells[$9,0] := '$9';
  Cells[$A,0] := '$A';
  Cells[$B,0] := '$B';
  Cells[$C,0] := '$C';
  Cells[$D,0] := '$D';
  Cells[$E,0] := '$E';
  Cells[$F,0] := '$F';
  Cells[16,0] := '$0';
  ColWidths[0] := 76;
  for i:=1 to 16 do
    ColWidths[i] := 25;
  ColWidths[17] := 136
end {Create};
```

➤ *Listing 6*

far) and the StringGrid Options to include the goEditing state so we can edit the contents of the cells.

Preparing To Write

The first thing we need to add to the TBHexViewer to make it a hex overwriter is the fact that all hidden private fields must be accessible to the derived class TBHexEditor. Note that since FFile is a private field of TBHexViewer no derived class can access the file. This means that no inherited class can ever hope to write to this file again, since the FFile property is inaccessible. So, while these fields were private in our first version, we really need to make them protected in TBHexViewer first, to make sure we can inherit from this class and re-use these fields later.

After we've done this, we can add two methods to TBHexEditor: GetBlock and SetBlock, that will work with these hidden protected fields. Note that it's the SetBlock method that actually uses the FBlock, FFile, FOffset and FSize fields to write a (changed) block back to the file. Without these hidden fields, we would have needed to re-write the entire class – which

```
procedure TBHexViewer.SetOffset(AnOffset: LongInt);
var i,j,k: Integer;
    Line: String;
begin
  AnOffset := AnOffset AND NOT BlockLine; { skip lower bits }
  if (AnOffset <> FOffset) or (AnOffset = 0) or (FOffset = 0) then begin
    FOffset := AnOffset;
    FillChar(FBlock,SizeOf(FBlock),#0);
    try
      if FFileName <> '' then
        try
          Seek(FFile,FOffset);
          BlockRead(FFile,FBlock,SizeOf(FBlock),FSize);
        except
          FOffset := 0;
          FSize := 0
        end
      else begin
        FOffset := 0;
        FSize := 0
      end;
    finally
      k := 0;
      for i:=1 to BlockLine do begin
        Cells[0,i] := '$'+IntToHex(FOffset + Pred(i) * BlockChar,8);
        for j:=1 to BlockChar do begin
          Inc(k);
          if k <= FSize then
            Cells[j,i] := IntToHex(FBlock[k],2)
          else
            Cells[j,i] := ''
        end;
        Dec(k,BlockChar);
        Line := '';
        for j:=1 to BlockChar do begin
          Inc(k);
          if k <= FSize then
            if FBlock[k] < 32 then
              Line := Line + ' '
            else
              Line := Line + Chr(FBlock[k])
        end;
        Cells[17,i] := Line
      end
    end
  end
end {SetOffset};
```

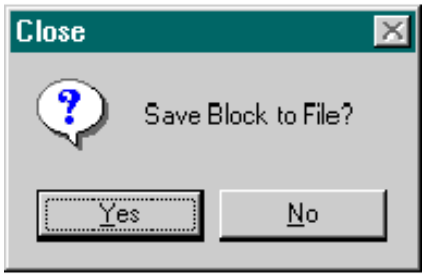
➤ *Listing 7*

```
procedure TBHexViewer.KeyDown(var Key: Word; Shift: TShiftState);
begin
  if Key = 34 then begin
    { PgDown }
    if Size = BlockSize then
      Offset := Offset + BlockSize
  end else if Key = 33 then begin
    { PgUp }
    if (Offset >= BlockSize) then
      Offset := Offset - BlockSize
    else
      Offset := 0
  end;
  inherited KeyDown(Key,Shift);
end {KeyDown};
```

➤ *Above: Listing 8*

➤ *Below Listing 9*

```
Type
  TBHexEditor = class(TBHexViewer)
    private
      FCopyBlock: TBlock;
      FChanged: Boolean;
    protected
      procedure SetOffset(AnOffset: LongInt); override;
      procedure SetFileName(AFileName: TFileName); override;
      procedure SetEditText(ACol, ARow: Longint; const Value: string);
        override;
      function GetEditLimit: Integer; override;
      function GetBlock: TBlock;
      procedure SetBlock(Const ABlock: TBlock);
    public
      constructor Create(AOwner: TComponent); override;
      destructor Destroy; override;
    published
      property Changed: Boolean read FChanged;
    end {TBHexEditor};
{...}
constructor TBHexEditor.Create(AOwner: TComponent);
begin
  inherited Create(AOwner);
  FAbout := 'TBHexEditor (c) 1996 by Dr.Bob';
  FChanged := False;
  Options := Options + [goEditing]
end {Create};
```



➤ *Figure 2*

```
function TBHexEditor.GetBlock: TBlock;
begin
  GetBlock := FBlock
end {GetBlock};
procedure TBHexEditor.SetBlock(Const ABlock: TBlock);
begin
  FBlock := ABlock;
  Seek(FFile,FOffset);
  BlockWrite(FFile,FBlock,FSize)
end {SetBlock};
```

➤ *Listing 10*

```
procedure TBHexEditor.SetEditText(ACol, ARow: Longint; const Value: string);
var i,j,k: Integer;
    Line: String;
begin
  if (ACol in [1..16]) and (ARow in [1..16]) and (Value <> '') then begin
    FChanged := True;
    k := Pred(ARow) * BlockChar;
    i := FCopyBlock[k + ACol];
    try
      if FileName = '' then
        raise Exception.Create(FileName);
      FCopyBlock[k + ACol] := StrToInt('$'+Value);
      inherited SetEditText(ACol, ARow, Value);
      Line := '';
      for j:=1 to BlockChar do begin
        Inc(k);
        if k <= Size then
         if FCopyBlock[k] < 32 then
           Line := Line + ' '
         else
           Line := Line + Chr(FCopyBlock[k])
      end;
      Cells[17,ARow] := Line
    except
      FCopyBlock[k + ACol] := i;
      MessageBeep($FFFF)
    end
  end else
    inherited SetEditText(ACol, ARow, Value)
end {SetEditText};
```

➤ *Listing 11*

is one more reason to make sure any design decision is made with care! See Listing 10.

Unfortunately, the VCL is full of examples where the designers chose to make some fields `private`, with the same horrible result. We often need to rewrite an entire class just to change its behaviour (and changing the interface won't work for the VCL, since this means a complete rebuild of the VCL and for that we need some additional files that Borland don't provide...).

Change

Since the user can now change the contents of the individual cells, we need to make sure we somehow detect that a change is made. For this we need to write code for the `SetEditText` event. First, we need to check again if the row and column are both within the range 1 to 16. Next, we need to make sure only a hex value is entered. See Listing 11.

To make sure the user can only enter two characters we write one line of code in the `GetEditLimit` function to return a value of 2.

Finally, if the user has changed some text (and hence `FChanged` is set to `True`), we need to be sure to save this text back to the `FFile` file. There are a few occasions that come to mind. Destroying the component is one (a special one), but also setting the offset of the filename to a new value. In the last two cases, we need to check the value of `Changed` and, if `True`, we simply pop up a `MessageDlg` that asks for confirmation to save the new information in the block to disk. A call to `SetBlock` (shown earlier) follows to save the new block (Listing 12).

Destructive

Showing a `MessageDlg` in the destructor is not a good idea. You'll know what I mean if you try it yourself. Here's what you get: a brief flicker of the `MessageDlg` followed by the termination of the component. And the `MessageDlg` result is a simple cancel (or close), since the block is not saved to file. The problem is that the dialog is shown for a window (the `StringGrid` component) that is being destroyed itself.

The solution is to go back to the Windows API itself, and use the good old `MessageBox` API, but this time not with the Window Handle of the `StringGrid` itself as parent, but with the current Window that has the focus (something you'll get after a call to `GetFocus`, another Windows API). See Listing 13.

Now, when destroying the `TBHexEditor` component, you will see the message box shown in Figure 2.

Property Editor

The `TBHexViewer` and `TBHexEditor` components depend on the validity of the filename: at design time we wouldn't want to supply an invalid name. That's exactly where property editors come in handy! For the `FileName` properties of type `TFileName` we can write a `paDIalog` type property editor that just fires an `OpenDialog` in its `Edit` procedure.

Check out my column in Issue 6 for more information and examples. The `TFileNameProperty` property editor we use here is in the unit FILENAME.PAS on the disk.

Register

As we've seen in the past few issues, I'm in favour of writing components in units without a `Register` procedure. I prefer to use a separate register unit where I register a collection of components and associated property and component editors all at the same time.

For the `TBHexViewer` and `TBHexEditor` components and the `TFileNameProperty` editor we can use the simple registration unit included in DRBOBREG.PAS on the disk. Place all four files in your compiler search path and install the `DrBobReg` unit.

Next Time

It's been a while since we've dealt with Delphi Experts in detail. With Delphi 2 we now have a new category: AddOn Experts.

Next time, we'll focus on this special kind of Expert where we need to do all the interfacing with the outside world by ourselves (in contrast with the standard Help, Form or Project experts we are used to). Be sure to check out our Error Report Expert next month...

Bob Swart (aka Dr.Bob, http://www.pi.net/~drbob/) is a professional software developer using Delphi and C++ for Bolesian, free-lance technical author for *The Delphi Magazine* and co-author of *The Revolutionary Guide to Delphi 2*. In his spare time, Bob likes to watch video tapes of *Star Trek Voyager* and *Deep Space Nine* with his 2.5-year old son Erik Mark Pascal.

```
procedure TBHexEditor.SetOffset(AnOffset: LongInt);
begin
  if (FileName <> '') and Changed {FCopyBlock <> GetBlock} and
    { save before exit } (MessageDlg('Save Block to File?',
    mtConfirmation, [mbYes, mbNo], 0) = mrYes) then SetBlock(FCopyBlock);
  inherited SetOffset(AnOffset);
  FCopyBlock := GetBlock;
  FChanged := False
end {SetOffset};
procedure TBHexEditor.SetFileName(AFileName: TFileName);
begin
  if (FileName <> '') and Changed {FCopyBlock <> GetBlock} and
    { save before exit } (MessageDlg('Save Block to File?', mtConfirmation,
    [mbYes, mbNo], 0) = mrYes) then SetBlock(FCopyBlock);
  FChanged := False;
  inherited SetFileName(AFileName);
  { which calls SetOffset }
end {SetFileName};
```

➤ *Listing 12*

```
destructor TBHexEditor.Destroy;
begin
  if (FileName <> '') and
    Changed {FCopyBlock <> GetBlock} and { save before exit }
    (WinProcs.MessageBox(GetFocus, 'Save Block to File?', 'Close',
    MB_ICONQUESTION or MB_YESNO) = IDYES) then
    SetBlock(FCopyBlock);
  inherited Destroy
end {Destroy};
```

➤ *Listing 13*