Beating the System: Animated Icons And Cursors, 1

by Dave Jewell

Many moons ago (Issue 22 to be precise) I described the internal anatomy of an icon file. This month I'll describe the format of the animated cursor files (.ANI) which Microsoft first introduced in Windows NT. Animated cursors are now supported under NT, Windows 95 and 98, and (this month and next) I'll be presenting code which allows you to programmatically implement animated cursors in Delphi, and allows you to display the animations in fixed locations on a form. Moreover, the implementation described here will stream the animation data into the form file, thus eliminating the need to deploy extra files or load special resource data at runtime *[A special round of applause seems appropriate here! Ed]*.

Anatomy Of An ANI File

An ANI file is a variation of Microsoft's so-called RIFF file format as used for various different types of multimedia files. If you want to get the definitive specification for RIFF and ANI files, you'll find there are several relevant documents at www.wotsit.org/, a website which is devoted specifically to accumulating and disseminating file format information.

The RIFF file format is 'chunk' based. A chunk begins with a four byte signature which identifies the type of the chunk. This is then followed by a 32-bit length specifier which indicates how much data is contained in the chunk, and this is then followed by the chunk data itself. Just to make things a little more interesting, chunks can contain 'sub-chunks' and, in the case of animated icon files, some chunks are optional.

The actual image data within an ANI file is stored as a series of icons. In fact, the data is stored in exactly the same format as if the various icons were individual .ICO files. If you look back to my article on .ICO files (Issue 22), you'll see that the icon data begins with a six-byte header that doesn't do much besides identify the number of icons in a file. In case you don't have that issue (or your *Collection '98 CD-ROM*) to hand, this data structure is reproduced below:

```
TIconHeader = packed record
  AlwaysZero : Word;
  CursorType : Word;
  NumIcons : Word;
end;
```

This is then followed by a sixteen byte header that contains more important information, such as the width and height of the icon, the number of colours in the icon and so forth:

```
TIconDirEntry = packed record
  Width, Height, Colors: Byte;
  Reserved: Byte;
  dwReserved: LongInt;
  dwBytesInRes: LongInt;
  dwImageOffset: LongInt;
end;
```

This second header is then followed by the icon data itself. I've seen some shareware implementations of animated icon components which make the assumption that an animated icon is only ever going to contain sixteen colour images. However, this isn't a valid assumption. If you're running a recent implementation of Windows such as Windows 98 *[Oh, sorry Dave, I thought you said* decent *for a moment there... Ed]*, you'll probably find that there's a \CURSORS directory hanging off your main WINDOWS directory. If you look in there, you'll find an assortment of ANI files, some of which (eg S_BUSY.ANI) contain full 256 colour images. My code tries not to make any assumptions about the size of embedded icon files.

As you'll no doubt appreciate, each embedded icon represents a single 'frame' of the movie. By displaying these frames one after another, the animation effect is achieved. Aside from the image data itself, the most important sub-chunk in the ANI file is the `anih` sub-chunk. This contains the `TAniIconHeader` data structure shown near the top of Listing 1.

In the usual way, the first field, `dwSizeof`, contains the total size of the data structure in bytes and is there to provide backwards compatibility in case Microsoft ever change this data structure. They probably wouldn't need to, because most of the fields in `TAniIconHeader` are already reserved for future use and will be found to contain zero in the file on the disk. This includes `dwCX`, `dwCY`, `dwBitCount` and `dwPlanes`. The `dwFrames` field stores the total number of frames in the movie, the number of distinct icon images stored within the ANI file. The `dwSteps` field, on the other hand, stores the number of discrete 'steps' within the animation.

Playing The Animation

In the simplest possible case, you might have an animation of (say) ten frames and exactly ten steps. The first step will therefore display the first icon, the second step will show the next icon and so on. Each step of the animation should be displayed for an amount of time defined by the `dwJIFRate` field in the `TAniIconHeader` structure. This value is given in 60th parts of a

➤ *Facing page: Listing 1*

```
unit UCAniIcon;
interface
uses
  Windows, SysUtils, Consts, Classes, Graphics;
type
  TAniIconHeader = record
    dwSizeof: LongInt;
    dwFrames: LongInt;
    dwSteps: LongInt;
    dwCX: LongInt; { use this to store icon width }
    dwCY: LongInt; { use this to store icon height }
    dwBitCount: LongInt;
    dwPlanes: LongInt;
    dwJIFRate: LongInt;
    dwFlags: LongInt;
  end;
  TAniIcon = class (TGraphic)
  private
    Rates: TList; { Optional JIFRate info for each step }
    FrameOffsets: TList; { Stream offsets into each frame }
    SequenceMap: TList; { Optional frame sequence mapping }
    Image: TMemoryStream; { Memory Image of .ANI file }
    fAuthor: String; { Optional author information }
    fTitle: String; { Optional title information }
    fHeader: TAniIconHeader; { ANI header extracted from file }
    fCurrentJIFs: Integer; { current JIF count for this step }
    fCurrentStep: Integer; { current step number }
    fCurrentFrame: Integer; { currently displaying frame number }
    fCurrentIcon: hIcon; { currently displaying icon }
    fTransparent: Boolean; { for transparent blitting }
    fBackColor: TColor; { background color when not transparent }
    procedure Clear;
    procedure SetFrame (Index: Integer);
  public
    constructor Create; override;
    destructor Destroy; override;
    procedure Assign (Source: TPersistent); override;
    procedure LoadFromStream (Stream: TStream); override;
    procedure SaveToStream (Stream: TStream); override;
    procedure Animate;
    procedure LoadFromClipboardFormat (AFormat: Word; AData:
      THandle; APalette: HPalette); override;
    procedure SaveToClipboardFormat (var Format: Word;
      var Data: THandle; var APalette: HPalette); override;
    procedure Draw (ACanvas: TCanvas; const Rect: TRect);
      override;
    property Author: String read fAuthor;
    property Title: String read fTitle;
    property Icon: hIcon read fCurrentIcon;
    property Transparent: Boolean read fTransparent
      write fTransparent default False;
    property BackgroundColor: TColor read fBackColor
      write fBackColor default clBtnFace;
  protected
    function GetEmpty: Boolean; override;
    function GetHeight: Integer; override;
    function GetWidth: Integer; override;
    procedure SetHeight (Value: Integer); override;
    procedure SetWidth (Value: Integer); override;
  end;
implementation
constructor TAniIcon.Create;
begin
  Inherited Create;
  fTransparent := False;
  fBackColor := clBtnFace;
  Rates := TList.Create;
  FrameOffsets := TList.Create;
  SequenceMap := TList.Create;
  Image := TMemoryStream.Create;
end;
destructor TAniIcon.Destroy;
begin
  Clear;
  Image.Free;
  Rates.Free;
  FrameOffsets.Free;
  SequenceMap.Free;
  Inherited Destroy;
end;
procedure TAniIcon.Clear;
begin
  fAuthor := '--unavailable--';
  fTitle := '--unavailable--';
  Image.Clear;
  Rates.Clear;
  FrameOffsets.Clear;
  SequenceMap.Clear;
  if fCurrentIcon <> 0 then DestroyIcon (fCurrentIcon);
  fCurrentIcon := 0;
end;
procedure TAniIcon.Assign (Source: TPersistent);
begin
  if Source = Nil then
    Clear
  else if Source is TAniIcon then
    LoadFromStream(TAniIcon(Source).Image)
  else
    Inherited Assign (Source);
end;
```

```
{ ** SOME CODE OMITTED HERE: SEE DISK FOR FULL LISTING ** }
procedure TAniIcon.LoadFromStream (Stream: TStream);
const
  sig_RIFF = $46464952; { RIFF header }
  sig_ACON = $4E4F4341; { ACON form type }
  sig_LIST = $5453494C; { LIST sub-chunk }
  sig_INFO = $4F464E49; { INFO sub-chunk }
  sig_INAM = $4D414E49; { INAM - title information }
  sig_IART = $54524149; { IART - author information }
  sig_anih = $68696E61; { anih - header information }
  sig_rate = $65746172; { optional JIF rates sub-chunk }
  sig_fram = $6D617266; { fram - list of icon frames }
  sig_icon = $6E6F6369; { icon - start of actual frame }
  sig_seq  = $20716573; { seq - opt sequence information }
var
  ChunkLen: LongInt;
  EncounteredHeader: Boolean;
  procedure InvalidFile;
  begin
    raise EInvalidGraphic.Create(
      'Animated icon image is not valid');
  end;
  function ReadByte: Byte;
  begin
    Image.ReadBuffer (Result, sizeof (Result));
  end;
  function ReadLong: LongInt;
  begin
    Image.ReadBuffer (Result, sizeof (Result));
  end;
  function ReadString: String;
  var
    p: PChar;
    Len: LongInt;
  begin
    Len := ReadLong;
    if (Len and 1) <> 0 then
      Inc (Len);
    GetMem (p, Len + 1);
    p[Len] := #0;
    Image.ReadBuffer (p^, Len);
    Result := StrPas (p);
    FreeMem (p, Len + 1);
  end;
  { Process an optional info header sub-chunk.
    Contains Title/Author }
  procedure ParseTitleAuthor;
  var ChunkEnd: LongInt;
  begin
    ChunkEnd := ReadLong;
    Inc(ChunkEnd, Image.Position);
    if ReadLong <> sig_INFO then
      InvalidFile;
    while Image.Position < ChunkEnd do
      case ReadLong of
         sig_INAM : fTitle  := ReadString;
         sig_IART : fAuthor := ReadString;
      end;
  end;
  { Parse ANI header information }
  procedure ParseAniHeader;
  begin
    if ReadLong <> sizeof (fHeader) then
      InvalidFile;
    Image.ReadBuffer (fHeader, sizeof (fHeader));
    EncounteredHeader := True;
  end;
  { Parse optional JIFRates chunk OR }
  { optional Sequence Map }
  procedure ParseList (List: TList);
  var Len: LongInt;
  begin
    Len := ReadLong div sizeof (LongInt);
    if Len <> fHeader.dwSteps then
      InvalidFile;
    while Len > 0 do begin
      List.Add (Pointer (ReadLong));
      Dec (Len);
    end;
  end;
  { Parse the actual icon data itself }
  procedure ParseIconList;
  var
    Idx: Integer;
    Len, Next: LongInt;
  begin
    ReadLong; { Discard chunk length }
    if ReadLong <> sig_fram then
      InvalidFile;
    { Store frame offsets for later consumption }
    for Idx := 0 to fHeader.dwFrames - 1 do begin
      if ReadLong <> sig_icon then
        InvalidFile;
      { Save position from beginning of length dword }
      FrameOffsets.Add(Pointer (Image.Position));
      { Read Length of this frame }
      Len := ReadLong;
      Next := Len + Image.Position;
{ ** CONTINUED ON FOLLOWING PAGE... ** }
```

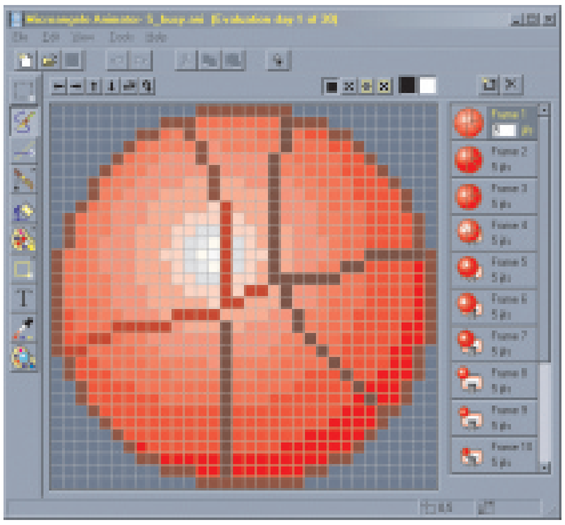
second. Thus, if dwJIFRate happens to be 3, then we know that the animation should 'step' twenty times per second.

That's the simplest case, but things can be more complex than this. For starters, there's an optional rate sub-chunk which specifies the length of time that each step of the animation ought to be displayed. Thus, rather than having each step last for the same amount of time, you can create animations where the same frame persists for a long time, and then a lot of action takes place very quickly. The rate sub-chunk is encoded simply as a list of 32-bit values, one for each step in the animation. If this sub-chunk isn't present, then the dwJIFRate field of the header is used throughout.

In a similar way, the size of an ANI file can be substantially reduced by re-using the same frame at different points in an animation. Almost all animations will make use of this feature, since very many cursor animations display a 'to and fro' characteristic: a complete animation cycle consisting of the action moving in one direction and then moving in the other, back to the starting point. It should be obvious that if we had to store a new icon for each frame we'd end up with many frames being duplicated within the file. The optional seq sub-chunk eliminates this problem by specifying the order in which frames are shown: it provides a mapping from a particular step to the frame which needs to be displayed during that step. As before, this chunk consists of a series of 32-bit integer values, one for each step in the animation. Each value within the array is an index into the list of icon images.

Introducing TAniIcon

In order to most easily work with animated icons, and incorporate animated icon capabilities into other components, I decided to write a new class, TAniIcon, which derives from the abstract class TGraphic. If you're well up on the VCL design philosophy, you'll know that this is the correct class to derive from when implementing a new graphic file format, it's used for TIcon, TMetaFile, TBitmap and so forth.



➤ *Figure 1: MicroAngelo is the definitive editor for creating and editing ANI files. Visit www.impactsoft.com for the latest version.*

Nearly all the code for TAniIcon is given in Listing 1 (and is of course in full on this month's disk). Amongst other things, this class implements all the methods that are defined as being abstract in the TGraphic class. If you've ever derived from an abstract class before, and you then write code which creates a new instance of the derived class, you'll know that the compiler will keep whingeing at you until it's happy that all the abstract methods have been replaced with real code!

Unlike some other implementations that I've seen, TAniIcon does not create all the necessary API-level icons for the animation at the same time. In other words, if you've got (for instance) an animation of ten, twenty or even forty different icons, some animated icon implementations create all those different icon handles together and store them as a handle array within the class. I didn't go down this route because I considered it to be a terrible waste of GDI system resources. Instead, each icon is created on the fly as required and then disposed of when it is no longer needed. Modern hardware is perfectly capable of providing the necessary horsepower to do this while still giving a smooth animation effect.

When a new ANI file is opened, the code loads the entire file into a TMemoryStream object and stores it there for later use, revisiting the contents of the stream whenever a new icon needs to be created. This is an approach that has minimal runtime overhead (even the largest ANI files on my machine are a mere 30Kb) and gives us a big pay-off when it's time to send the file data somewhere else, such as might be required through a call to Assign or SaveToStream. If we didn't have all the file data to hand, then we would have to painfully reconstruct the ANI file image on a byte-by-byte basis which would add very considerably to the complexity of the code.

As you'll see from the code listing, the Image field is used to store an image of the file data. The FrameOffsets field is used to maintain a list of 32-bit offsets into the image data, each offset representing the start of a particular frame. This list is constructed at the time the data is first parsed. As I mentioned earlier, the rate and seq sub-chunks are optional; these correspond to the Rates and SequenceMap fields within the TAniIcon class. If the corresponding information isn't present in the image data, then these lists will be empty. The private fAuthor and fTitle fields correspond to an optional info block stored within the file, as we shall see.

If you're not familiar with the TGraphic class, it's worth pointing out that there's no need to directly implement LoadFromFile and SaveToFile methods in TAniIcon. These methods are already implemented in TGraphic, and they simply call down to the abstract LoadFromStream and SaveToStream abstract methods which we *do* have to implement. From this, and bearing in mind my previous comments about storing the entire file image in memory, you should

```
      { Dig a little deeper to get the icon size info }
      if Idx = 0 then begin
        Image.Position := Image.Position + 6;
        fHeader.dwCX := ReadByte;
        fHeader.dwCY := ReadByte;
      end;
      Image.Position := Next;
    end;
  end;
begin { LoadFromStream }
  Clear;
  Image.LoadFromStream (Stream);
  EncounteredHeader := False;
  { Validate initial eight-byte header. Some .ANI files have
    filesize > header (eg appstart.ani) }
  if (ReadLong <> sig_RIFF) or (ReadLong > Image.Size) then
    InvalidFile;
  { Next item must be an ACON chunk }
  if ReadLong <> sig_ACON then
    InvalidFile;
  while Image.Position < Image.Size do
    { Case out on the sub-chunk we find }
    case ReadLong of
      sig_LIST : if not EncounteredHeader then
        ParseTitleAuthor else ParseIconList;
      sig_anih : ParseAniHeader;
      sig_rate : ParseList (Rates);
      sig_seq :  ParseList (SequenceMap);
    else begin
      { Unknown chunk - just skip it }
      ChunkLen := ReadLong;
      Image.Position := Image.Position + ChunkLen;
    end;
  end;
  SetFrame (0);
end;
procedure TAniIcon.SaveToStream (Stream: TStream);
begin
  if GetEmpty then
    raise EInvalidGraphicOperation.Create(sInvalidImage);
  with Image do
    Stream.WriteBuffer (Memory^, Size);
end;
procedure TAniIcon.Draw(
  ACanvas: TCanvas; const Rect: TRect);
var bm: TBitmap;
begin
  if fCurrentIcon <> 0 then begin
    if not fTransparent then begin
      bm := TBitmap.Create;
      bm.Width := fHeader.dwCX;
      bm.Height := fHeader.dwCY;
      bm.Canvas.Brush.Color := fBackColor;
      bm.Canvas.FillRect(
        Classes.Rect (0, 0, bm.Width, bm.Height));
      DrawIcon(bm.Canvas.Handle, 0, 0, fCurrentIcon);
      ACanvas.Draw(Rect.Left, Rect.Top, bm);
      bm.Free;
    end else
      DrawIcon(ACanvas.Handle, Rect.Left, Rect.Top,
        fCurrentIcon);
  end;
end;
```

```
procedure TAniIcon.SetFrame(Index: Integer);
type
  TIconHeader = packed record
    AlwaysZero: Word;
    CursorType: Word;
    NumIcons: Word;
  end;
  TIconDirEntry = packed record
    Width, Height, Colors: Byte;
    Reserved: Byte;
    dwReserved: LongInt;
    dwBytesInRes: LongInt;
    dwImageOffset: LongInt;
  end;
var
  p: PByte;
  ChunkLen: LongInt;
  IconHeader: TIconHeader;
begin
  if (FrameOffsets.Count <> 0)
    and (Index < fHeader.dwFrames) then begin
    fCurrentFrame := Index;
    if fCurrentIcon <> 0 then   // Delete any existing icon
      DestroyIcon (fCurrentIcon);
    // Seek to wanted position in stream data
    Image.Position := Integer (FrameOffsets [Index]);
    Image.ReadBuffer (ChunkLen, sizeof (ChunkLen));
    Image.ReadBuffer (IconHeader, sizeof (IconHeader));
    Image.Position := Image.Position +
      (sizeof(TIconDirEntry)*IconHeader.NumIcons);
    Dec(ChunkLen, sizeof(IconHeader) +
      (sizeof (TIconDirEntry) * IconHeader.NumIcons));
    p := Image.Memory;
    Inc(p, Image.Position);
    fCurrentIcon :=
      CreateIconFromResource(p, ChunkLen, True, $30000);
    Changed (Self);
  end;
end;
procedure TAniIcon.Animate;
var JifRate, NextFrame: Integer;
begin
  if Rates.Count = 0 then
    JifRate := fHeader.dwJIFRate
  else
    JifRate := Integer (Rates [fCurrentStep]);
  Inc (fCurrentJIFs, 4);
  if fCurrentJIFs >= JifRate then begin
    { Time to move on to next step }
    fCurrentJIFs := 0;
    Inc (fCurrentStep);
    if fCurrentStep >= fHeader.dwSteps then
      fCurrentStep := 0;
    if SequenceMap.Count = 0 then
      NextFrame := fCurrentFrame + 1
    else
      NextFrame := Integer (SequenceMap [fCurrentStep]);
    if NextFrame >= fHeader.dwFrames then
      NextFrame := 0;
    if NextFrame <> fCurrentFrame then
      SetFrame (NextFrame);
  end;
end;
end.
```

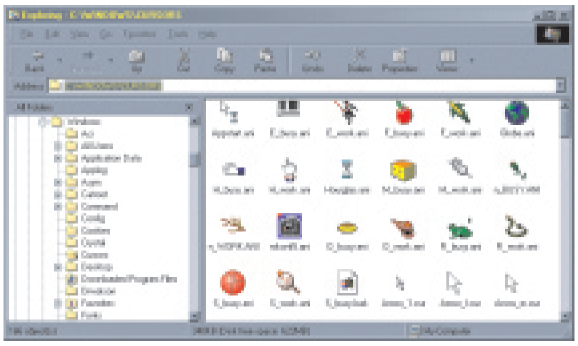
➤ *Listing 1, continued*

readily appreciate that the only non-trivial routine in the entire class is going to be the LoadFromStream method. It's this routine which is really the heart of the entire unit, it's here that the incoming stream is parsed and all the important data structures are initialised. The code begins by calling the Clear method to remove any traces of the existing animation. This re-initialises all lists and (if present) destroys the current API-level icon handle corresponding to the current frame. Next, the entire file image is read into the TMemoryStream object.

As ever, Microsoft have added a few little 'gotchas' to the ANI file format, and the first of these is the possible presence of *two* different INFO sub-chunks within the file. The actual image data is contained within one INFO chunk, and the presence of this chunk type is obviously mandatory: no image data, no comment! However, the other INFO chunk is optional: it only contains title and author information which, for the sake of completeness, I decided to expose as public properties of the class. If the optional chunk isn't present both these property strings are initialised to –unavailable–).

This raises the question of how to distinguish the two INFO chunk types? Fortunately, it turns out that the image data will only ever appear after the mandatory anih chunk whereas the author/title data will only ever appear *before* anih. Thus, we can easily discriminate between the two simply by noting whether or not we've parsed the anih chunk yet. Purists and frustrated compiler writers(!) may prefer to implement a state machine, look-ahead facility, or whatever, but there's really nothing to be gained from such an approach in this case.

The initial four-byte file signature (RIFF) should immediately be followed by a 32-bit value which specifies the size of the remainder of the file. Thus, conceptually, the RIFF chunk is the outermost, global, chunk. Another gotcha I discovered here was that some Microsoft-authored ANI files



➤ *Figure 2: Windows comes with a number of animated icon files which you can use with the TAniIcon class developed here.*

contain extraneous data which follows the RIFF chunk. To put it another way, some ANI files appear to be longer than they should be. Fortunately, this is easy to address in validation checks that I perform.

The code then simply loops round, reading successive chunks, until the end of the `Image` stream is reached. The Microsoft documentation is vague about the order in which certain chunks appear, and I've therefore tried to make the code as 'context-free' as possible. Chunk types which aren't recognised by the scanner are simply skipped over. If any validation code fails, then a show-stopping `EInvalidGraphic` exception is raised.

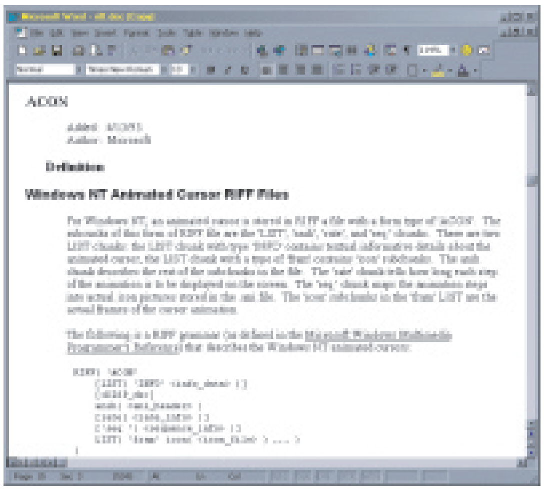
The `anih` chunk fills in the `fHeader` data structure with the important file information that I discussed earlier when explaining the `TAniIconHeader` structure. Later, when the actual image data is being parsed, the `ParseIconList` routine checks to see if the image entry being parsed is the first entry. If so, it digs a little deeper into the stream data and extracts the width and height image information for the icon. These values are then stored into the `dwCX` and `dwCY` fields of the header, saving a couple of bytes of memory. I once earned a living programming games for Apple II computers fitted with 4Kb RAM and old habits die hard! Finally, the `LoadFromStream` method finishes by calling `SetFrame` with an argument of zero, which creates an API-level icon handle for the first frame in the animation.

Creating The Icon

The `SetFrame` code ensures the passed frame number is valid and then deletes any existing icon handle. Next, it seeks to the relevant position in the stream data and reads in the length of the sub-chunk. The 32-bit length field is immediately followed by a `TIconHeader` data structure. For some reason which I don't fully understand, some ANI files contain multiple icons within each frame, and this situation has to be addressed by examining the `NumIcons` field of the `TIconHeader`. There's one `TIconDirEntry` in the data stream for each icon present, and the code uses the `NumIcons` count to skip ahead to the start of the image data proper. So far, I haven't found an ANI file which breaks this code, but if you wanted to add a little more validation code here, you could check that the next 32-bit long-word in the data stream (after skipping the various headers) is equal to `$28`. Referring back to my article in Issue 22, this corresponds to the size of the `TBitmapInfoHeader` data structure.

Rather than allocating a buffer and reading all the image data into the buffer prior to creating the bitmap, we can just use the `Memory` property of the `Image` stream to directly retrieve a pointer to the data (at least I knew about *that* property! ☺ – see *A Slice of Humble Pie*). Having got a pointer to the `TBitmap InfoHeader` data and the following image information, we call the routine `CreateIconFromResource` to create an actual icon handle.

➤ *Figure 3: The Microsoft RIFF specification (which includes notes on the .ANI file format) can be downloaded from the file format repository at www.wotsit.org*



ACON

Added: 4/1/95
Author: Microsoft

Definition

Windows NT Animated Cursor RIFF Files

For Windows NT, an animated cursor is stored in RIFF a file with a form type of 'ACON'. The subchunks of this form of RIFF file are the 'LIST', 'anih', 'rate', and 'seq' chunks. There are two LIST chunks: the LIST chunk with type 'INFO' contains textual information details about the animated cursor, the LIST chunk with a type of 'fram' contains 'icon' subchunks. The anih chunk describes the rest of the subchunks in the file. The 'rate' chunk tells how long each step of the animation is to be displayed on the screen. The 'seq' chunk maps the animation steps into actual icon pictures stored in the .ani file. The 'icon' subchunks in the 'fram' LIST are the actual frames of the cursor animation.

The following is a RIFF grammar (as defined in the Microsoft Windows Multimedia Programmer's Reference) that describes the Windows NT animated cursors:

```
RIFF( 'ACON'
    [LIST( 'INFO' <info_data> )]
    [<DISP_ck>]
    anih( <ani_header> )
    [rate( <rate_info> )]
    ['seq '( <sequence_info> )]
    LIST( 'fram' icon( <icon_file> ) ... )
)
```

If you're feeling adventurous, you might like to modify my code so that it uses the `CreateIconFromResourceEx` routine. The advantage of this is that it will enable you to create animations which are larger (in terms of width and height) than the icon data contained within the ANI file. This is done by internally stretching the bitmap information to the specified dimensions. By adopting such a strategy, you could provide a more complete implementation of the `SetWidth` and `SetHeight` methods, maintaining internal variables which indicate the desired size of the animation, and then using them when it's time to actually create the icon handle. I haven't gone down this route myself because I'm not a great fan of stretched bitmaps, but if you want to do it, the necessary code changes are relatively trivial.

One Jiffy = Three Shakes Of A Dead Lamb's Tail...

There are only two other routines in the `TAniIcon` class which are worthy of mention. Firstly, the `Animate` method. This method must be called periodically by whatever code has created the `TAniIcon` object in order to 'drive' the animation. Each time it's called, it bumps the jiffy count that pertains to the currently displayed frame. When this count exceeds the 'display time' that's specified for this particular frame, then the animation moves on to the next step. This typically involves calling the `SetFrame` routine in order to render the next icon handle. The `Animate`

A Slice Of Humble Pie

There are times when technical journalism can be something of a sweet and sour experience. The sweet part (as you no doubt suspect) has to do with being able to get your hands on all manner of programming goodies without actually having to pay for anything. The sour part, on the other hand, means that when one screws up, one does so in a very public manner!

A couple of weeks ago, I got an email message from a reader of *The Delphi Magazine*, Andrew McLellan (andrew@cix.compulink.co.uk), who gently pointed out a rather embarrassing oversight in last month's article on the system image list. You may remember that I devoted some time to stressing that you could read the system image list information, but that you weren't allowed to change it. In particular, I noted the dangers of 'wrapping' a normal VCL-level `TImageList` component around the system image list, explaining that when the `TImageList` object is destroyed, the system image list will be destroyed as well, with devastating consequences for the Windows desktop.

That's certainly true as far as it goes, but Andrew pointed out that by setting the `SharedImages` property of the `TImageList` object to `True`, the aforementioned problem could be overcome. The `SharedImages` property is there specifically to cater for this sort of situation, and it prevents the underlying API-level system list from being destroyed when the VCL control is released. It's important to set the `TImageList` property *after* setting the `Handle` property of the control to the system image list. If you set if before, then any pre-existing image list would not be freed.

I guess the moral of the story is that every VCL component always has at least one more feature than you think it has! I haven't done a great deal of work with image list components (at least, not at the VCL level) and my mind had never registered the presence of the `SharedImages` property let alone what it does. So there you have it, a cigar for Andrew and a large slice of humble pie for me!

method should be fairly self-explanatory, apart from the odd way in which I've incremented the Jiffy count by 4. This is explained a little further on.

The `Draw` method is where we finally get to draw the current frame image on a canvas. It should be obvious that icons are inherently transparent images which means that, when calling `DrawIcon`, the background still 'shows through'. This is what we want when using icons on the taskbar, on the Windows desktop, or wherever, but it's a deeply bad idea when using animated icons because it means that each frame will be drawn on top of the preceeding frame, creating a sort of unintentional 'time-lapse' effect! In order to address this problem, I implemented `Transparent` and `BackgroundColor` properties of the `TAniIcon` class. By default, the `Transparent` property will be `False`, which is almost always what we want. When used in this way, the `Draw` method creates a small, off-screen, icon-sized bitmap, pre-fills it with the currently selected background colour and then draws the current frame on top of this. The resulting bitmap is what then gets drawn on the display canvas. By doing things in this way, we avoid the unpleasant flicker effects that would result from calling `FillRect` directly on the display canvas, and then immediately overwriting it with the current icon image.

The sample program in Listing 2 shows one possible arrangement for making use of `TAniIcon`. Here, a `TTimer` object is used to drive the `Animate` method, calling the `Draw` method to blit the current frame onto the screen at the same time. As an alternative, you could use the `OnChange` event of the `TAniIcon` object to redraw the icon only when the frame has actually changed.

The source code to both units is, as ever, included on this month's disk, and there is also a 'packaged' version of the compiled executable which you will be able to run if you have the Delphi 3 runtime packages installed. Bear in mind, though, that nothing much will happen if you don't have the actual ANI file which the program is expecting to find.

➤ *Listing 2*

```
unit Unit1;
interface
uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls,UCAniIcon;
type
  TForm1 = class(TForm)
    Timer1: TTimer;
    procedure FormCreate(Sender: TObject);
    procedure FormDestroy(Sender: TObject);
    procedure Timer1Timer(Sender: TObject);
  private
    ani: TAniIcon;
  public
  end;
var Form1: TForm1;
implementation
{$R *.DFM}
procedure TForm1.FormCreate(Sender: TObject);
begin
  ani := TAniIcon.Create;
  ani.LoadFromFile ('c:\windows\cursors\s_busy.ani');
  ani.BackgroundColor := Self.Color;
  Caption := ani.Title;
  Timer1.Interval := 50;
  Timer1.Enabled := True;
end;
procedure TForm1.FormDestroy(Sender: TObject);
begin
  ani.Free;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
  ani.Animate;
  ani.Draw (Canvas, Rect (0, 0, ani.Width, ani.Height));
end;
end.
```

I mentioned earlier the question of updating the Jiffy count by 4 each time the `Animate` method is called. The author of MicroAngelo has a lot to say about his software being the standard by which others are judged, and I therefore tried to match the animation speed of `TAniIcon` to the animation speed of the same ANI file in MicroAngelo. By trial and error, I found that incrementing the Jiffy count by 4 inside the `Animate` method and using a timer interval of 50 milliseconds gave a reasonably close match. This gives a 'logical Jiffy' value of one 80th of a second. In theory, one should be aiming for a 'logical Jiffy' value of one 60th of a second, but in practice this seemed rather on the slow side. It's really a case of 'how long is a piece of string?' or perhaps, how many shakes of a dead lamb's tail correspond to a Jiffy? Before leaving the subject of animation speed, it's also important to point out that `WM_TIMER` messages are inherently inaccurate, so it's therefore better to go for a relatively long timer interval (commensurate with a smooth animation) and compensate for it inside the `Animate` code as I've done here. This will tend to iron out minor irregularities in the `WM_TIMER` repetition rate.

Next month, I'll provide a couple of drop-in Delphi components which make use of `TAniIcon`, one for displaying an animation on a form, and another for creating an animated icon in the tray area. We'll also look at a property editor for the `TAniIcon` class, and I'll be showing you how to associate an animated icon with the mouse cursor itself. See you then!

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level work. He is Technical Editor of *Developers Review* which is also published by iTec. Contact Dave at Dave@HexManiac.com