Beating the System: Manipulating 16-Bit Resources: 2

by Dave Jewell

Firstly, an apology. A couple of 'bugettes' crept into last month's code. For those seeking proof of your scribe's fallibility, look no further! In the RESFILE.PAS source file, the destructor for `TResFile` should have been declared with the `override` specifier. The fact that it wasn't meant that it never got called at all. Oops! Also, in the main application file, the code to clear the listboxes and destroy the existing `TResFile` object should only get executed if the call to the `Execute` method of the open dialog returned `True`. Oh well...

Don't worry too much about making the changes, because a revised version of the source code is included on this month's disk – I'm just trying to come clean, that's all!

You'll remember that last time I described how to programmatically examine the resources in a 16-bit executable, extract a specified resource and write it to disk. As I stated then, the problem with this approach is that you don't end up with a valid file that can be used by other applications. For example, if you grab the resource data for an icon, hope for the best and merely stuff it into a file with the .ICO extension, you don't end up with a valid file. This is because icon files have a standard header which precedes the raw icon data. The same is true of bitmaps. In this article, we're going to look at how to take the raw resource data and package it in such a way as to end up with a standard file.

Icon Madness

In his innocence, the Editor asked me if I could include the code to create kosher icon files in last month's article. In my innocence, I had originally intended to do this, but it turns out that there's a lot more to icons than meets the eye. You may not realise this, but a single .ICO file can potentially contain several different icon images, and each icon image is actually made up of two distinct bitmaps! What's the reason for this complexity? It's all to do with the device independent nature of Windows. Nowadays, almost everyone is using super VGA display hardware, but back in the early days, you could have CGA and EGA displays, with different aspect ratios and widely differing display capabilities. By having several different icon images inside a single 'logical icon', the Windows USER library can examine the capabilities of the display hardware and select the icon image which gives the closest match, thereby producing the best-looking image on the screen.

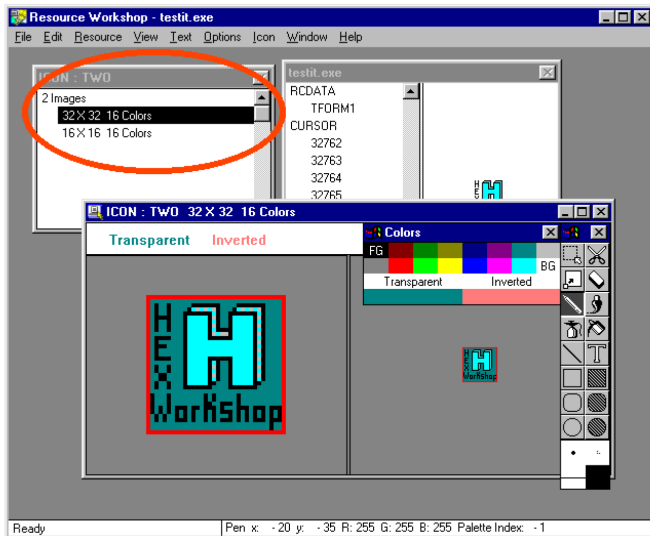
The reason that each icon is made up of two distinct bitmaps is simple. An icon consists of two parts: a foreground image (which is what you really think of as the icon) and a background image which acts as a mask. The mask is needed to 'punch a hole' in the desktop, into which the foreground image is placed. It's this mechanism which allows an icon to have a non-rectangular appearance with the background showing through in certain parts of the image. When you use an icon editor, you're not actually aware that you're creating two separate bitmaps: the editor simply analyses the parts of the image that you've specified as being transparent and subtracts those parts from the enclosing rectangle to create the mask. The foreground image is a colour bitmap and the background mask is invariably a monochrome bitmap, thus saving space in the .ICO file.

Sounds reasonably simple? Read on! What I've been talking about up until now is the format of the .ICO files, but things are more complex when we look into the .EXE file itself. Suppose that you've got an EXE file containing an icon with two different display resolutions, say, 16 by 16 pixels and 32 by 32 pixels. When you look at this file with Borland Resource Workshop, you only see a single `ICON` resource which you can double click and then select which image you want to work with. This is illustrated in Figure 1 where you can see the 32 by 32 pixel image being edited, with the image list in the background.

However, if you now look at the same EXE file with my little resource sniffer utility from last month, you'll see that two distinct `ICON` resources show up in the EXE file, along with a mysterious `GROUPICON` resource. What's going on? The answer is that Resource Workshop takes a high-level view of resources, whereas my code shows the actual, low-level, state of affairs. It's the `GROUPICON` resource (not the `ICON` resource) which corresponds to a logical icon within a Windows executable. The `GROUPICON` resource specifies how many different resolution icon images are contained within the logical icon and provides pointers to each of the individual `ICON` resources. Effectively, the `GROUPICON` resource acts as a 'directory' for each logical icon contained within the file. A little later, I'll describe the format of `GROUPICON` resources and demonstrate how to use the information contained therein to 'stitch together' a multi-part .ICO file from one or more individual icon resources.

.ICO File Format

Having briefly described where we're going, it's now time to look at the details. The format of .ICO files consists of a surprising number of parts, but it's all relatively straightforward. Firstly, there's a small



➤ *Figure 1: Resource Workshop takes a high level view of resource management; the image list in the top-left corner actually corresponds to the contents of a GROUPICON resource*

data structure which I've called `TIconHeader`. The first and second fields of this record are always zero and one respectively. In the general case, the third field specifies the number of icons contained within the file. Most .ICO files only contain one or two icons, but in principle you could have many more:

```
TIconHeader = record
  AlwaysZero: Integer;
  AlwaysOne: Integer;
  NumIcons: Integer;
end;
```

The icon header is immediately followed by an array of `TIconDirEntry` records, one for each icon in the file:

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

As you'd expect, the `Width`, `Height` and `Colors` fields of this data structure specify the dimensions and the colour format of this icon image. The next two fields are unused and should be set to zero while the `dwBytesInRes` field contains the total size of all the icon image data in the file. That is, the size of the file minus the size of the `TIconHeader` structure, minus the size of however many `TIconDirEntry` records happen to be contained in the file. Finally, the `dwImageOffset` field is an offset (relative to the start of the file) to the image data for this particular icon.

The image data for each icon begins with a standard `TBitmapInfoHeader` record. This is defined by Microsoft and shown below:

```
TBitmapInfoHeader = record
  biSize: Longint;
  biWidth: Longint;
  biHeight: Longint;
  biPlanes: Word;
  biBitCount: Word;
  biCompression: Longint;
  biSizeImage: Longint;
  biXPelsPerMeter: Longint;
  biYPelsPerMeter: Longint;
  biClrUsed: Longint;
  biClrImportant: Longint;
end;
```

The `biSize` field specifies the size of this data structure (`$28` in hex) while the `biHeight` and `biWidth` fields mirror the values given in the `TIconDirEntry` record with one notable exception: a valid icon file generally has the `biHeight` field set to twice what you'd expect it to be. This is conjecture on my part, but I believe that Microsoft originally intended to format icon files as a double-height bitmap, with the foreground image in the top half and the mask in the lower half. Early on, they realised that this would be wasteful because the mask only ever needs to be monochrome, but the double-height characteristic of `biHeight` is preserved, possibly because some system software expects it to be so.

The `TBitmapInfoHeader` record is immediately followed by an array of `TRGBQUAD` fields (again, this record type is pre-defined by Borland and Microsoft), one for each possible palette entry used by the icon. For example, if this is a 16-bit icon, there will be 16 `TRGBQUAD`s following the `TBitmapInfoHeader` record.

Next (we're nearly there!) comes the image data for the foreground bitmap image and this is immediately followed by the image data for the mask. To make this all a bit clearer, let's work through the size calculations for a 32 x 32 pixel, 16-bit icon.

Because there's only one icon in the file, there will be only one 16 byte `TIconDirEntry`, so the total size of this plus the `TIconHeader` record will be 22 bytes. Now, a `TBitmapInfoHeader` record occupies 40 bytes and since there are sixteen possible colour values, there will be sixteen `TRGBQUAD` structures (each 4 bytes long), making a total (so far) of 22+40+64 = 126 bytes. Because the icon size is 32 x 32 pixels, a total of 1024 pixels must be represented in the foreground image data, but since a 16-colour pixel can be represented by only four bits, we only need half a byte for each stored pixel. This gives 1024/2 = 512 bytes for the foreground image. Finally, the mask will also need to represent 1024 pixels, but because it is monochrome, we only need 1 bit per pixel (either on or off) meaning that we can pack eight pixels into each byte. The mask size will therefore be 1024/8 = 128 bytes. Thus, the total size of the icon file will be 126 + 512 + 128 = 766 bytes. Sure enough, we find that single icon files containing just one 32 x 32 pixel, 16-colour image are 766 bytes in size.

Yellow Pages For Icons

Returning to the executable, just how is a `GROUPICON` resource formatted? It's actually quite similar to the `TIconDirEntry` record that we're already looked at. When I wrote this article, I blissfully assumed that the format of `GROUPICON` resources would be well documented. No such luck. After a certain amount of disassembly and

head-scratching, I came up with the data structure shown below:

```
TGroupDirEntry = record
  Width, Height, Colors: Byte;
  Reserved: Byte;
  Planes: Integer;
  BitCount: Integer;
  ImageSize: LongInt;
  IconID: Word;
end;
```

A `GROUPICON` resource actually consists of an array of these data structures, preceded by a `TIconHeader` as already described. The `NumIcons` field of the `TIconHeader` indicates how many `TGroupDirEntry` structures follow. The `ImageSize` field contains the number of bytes of image data associated with each icon. This is important information, because it saves us having to work it out as each individual icon is written to disk. (Remember from last month that, within an executable, the physical size of a resource will be rounded up to a multiple of the resource alignment factor. Consequently, we can't just take the physical resource size as being the image size). Even more important is the `IconID` which maps a specific entry in the icon's *Yellow Pages* list onto a single `ICON` resource.

Armed with the above, we've now got all the information we need to write standard icon files to disk. For the sake of brevity, Listing 1 shows only the necessary additions to last month's code: you'll find a complete code listing on the disk as usual.

As you can see, `WriteResourceFile` now examines the type of resource being written and appends a suitable file extension to the file being created. It also calls one of three different methods depending on whether we're writing a group icon, a single icon resource, or a bitmap. For now, let's assume the former case and take a look at the `WriteMultiIconFile` code.

As you'll have seen from our earlier discussion, the `TIconHeader` structure which sits at the front of a `GROUPICON` resource is formatted exactly as we want it to be for writing to the icon file. Thus, the first thing we do is read the first six bytes of the resource data and then write them immediately to the `TFileStream`. Once we've done that, the count of the number of icons in the file is readily available. The code then loops once for each icon, pulling the `TGroupDirEntry` records out of the resource data and creating an equivalent `TIconDirEntry` structure which is immediately written to disk. What's very important here is the calculation of the file offset for each icon's image data. The `Offset` variable is used for this, initialising it to point immediately after the last `TIconDirEntry` in the file. `Offset` is then incremented by the value of each `TGroupDirEntry`'s `ImageSize` field each time round the loop.

➤ *Figure 2: Here's the structure of an .ICO file; although you might not expect it, .ICO files are actually more complex than .BMP files*

Once all the icon directory entries have been written, the code 'backs up' to the first `TGroupDirEntry` structure in the resource data and loops forward again, this time calling `WriteIndividualIcon` to load each icon's image into memory and write it to the file stream.

The code for `WriteIndividualIcon` needs to take the numeric ID of an icon and map this onto an index which can be passed to `GetResourceInfo`. This isn't an entirely elegant piece of code, and there's a good argument here for supplementing the interface of `TResFile` by allowing you to directly access a resource of specified type and name/number. Incidentally, the "component" icons of a `GROUPICON` resource are always identified by number rather than name, so that the number can be stored in the `IconID` field of the `TGroupDirEntry` (see last month's article for an explanation of name/number identification in resources).

Having mapped the `IconID` parameter onto the correct icon, the rest of the `WriteIndividualIcon` code is trivial. It simply opens the executable file and loads the resource data into memory using a `TFileStream` as done elsewhere in the program. The icon data is then appended to the multi-part icon file, taking care to write the number of bytes specified in the `TGroupDirEntry`, rather than the physical resource size. And that's it! Once all the individual icon images have been written and the file closed, you then end up with an industry standard .ICO file.

This isn't quite the end of the story as far as icons go. Up to now, I've only described how to extract multi-part `GROUPICON` resources into .ICO files. But, you might also want to extract just a specific icon image from a file, or you might find yourself with a very old executable that doesn't contain any `GROUPICON`s, only `ICON` resources.

For maximum flexibility, I've therefore added code to extract single `ICON` resource files as well. This is handled by the `WriteSingleIconFile` method. The code is pretty straightforward with the exception of the call to `GetImageSize` which tries to calculate an appropriate image size for the icon. The code in this routine effectively works through the sort of sizing calculation that we've already discussed, but also tries to cater for a special case that apparently applies to 16 x 16 pixel, 16 colour icons. The rest of the code is relatively simple.

And Finally, Bitmaps

Funnily enough, creating .BMP files is considerably simpler than the preceding machinations involving icons. This is mainly due to the fact that a .BMP file contains exactly one bitmap, while as we've seen an icon file can contain several. The only new data structure here is `TBitmapFileHeader`, another of the standard API record types which have already been predefined by Borland/Microsoft:

```
TBitmapFileHeader = record
  bfType: Word;
  bfSize: Longint;
  bfReserved1: Word;
  bfReserved2: Word;
  bfOffBits: Longint;
end;
```

The first field, `bfType`, must be set to `$4D42` which identifies the file as a bitmap. The next field, `bfSize`, must be equal to the total size of the file (irrespective of the image size of the bitmap itself) and is used as a further check that the file is valid. The next two fields are reserved and should be set to zero. Lastly, the `bfOffBits` field is an offset (relative to the top of the file) which points to the actual image data for the bitmap. Within the .BMP file, the `TBitmapFileHeader` data structure is immediately followed by a `TBitmapInfoHeader`, an array of `TRGBQuads`, and the image data itself: this is all just as for icon files, except that, since this is a straight bitmap, there is no monochrome mask following the main image.

➤ *Listing 1*

```
function GetIconImageSize(ImageData: PByte): LongInt;
  var bm: TBitmapInfoHeader;
begin
  Result := 0;
  if ImageData^ = sizeof (bm) then begin
    bm := PBitmapInfoHeader (ImageData)^;
    bm.biHeight := bm.biHeight div 2;
    Result := sizeof (bm);
    Inc (Result, sizeof (TRGBQuad) *
      LongInt(1 shl bm.biBitCount));
    { Allocate another 1 bit/pixel for mask }
    Inc (bm.biBitCount);
    { Hack: Older 16*16, 16-color icons use 2 bits/pixel for mask! }
    if (bm.biHeight = 16) and (bm.biWidth = 16) and
      (bm.biBitCount = 5) then Inc (bm.biBitCount);
    Inc(Result, ((bm.biWidth * bm.biHeight + 7) div 8) *
      bm.biBitCount);
  end;
end;
procedure TForm1.WriteIndividualIcon(fs: TFileStream;
  IconID, ImageSize: Integer);
var
  Info: TResInfo;
  ResData: PByte;
  IconList: TStringList;
  IconStream: TFileStream;
begin
  IconList := rf.GetResList ('ICON');
  IconID := IconList.IndexOf ('#' + IntToStr (IconID));
  rf.GetResourceInfo ('ICON', IconID, Info);
  GetMem (ResData, Info.rLength);
  try
    IconStream :=
      TFileStream.Create(OpenDialog.FileName, fmOpenRead);
    try
      IconStream.Position := Info.rOffset;
      IconStream.Read (ResData^, Info.rLength);
    finally
      IconStream.Free;
    end;
    fs.Write (ResData^, ImageSize);
  finally
    FreeMem (ResData, Info.rLength);
  end;
end;
procedure TForm1.WriteMultiIconFile(fs: TFileStream;
  ImageData: PByte; Len: Word);
var
  Idx: Integer;
  DirStart: PByte;
  Hdr: TIconHeader;
  Offset: LongInt;
  Dir: TIconDirEntry;
  GroupDir: TGroupDirEntry;
begin
  { Get the file header and write it now }
  Move (ImageData^, Hdr, sizeof (Hdr));
  fs.Write (Hdr, sizeof (Hdr));
  Inc (ImageData, sizeof (Hdr));
  { Save current position and calculate initial image offset}
  DirStart := ImageData;
  Offset :=
    sizeof(Hdr)+(Hdr.NumIcons * sizeof(TIconDirEntry));
  { Write all the TIconDirEntry entries to disk }
  for Idx := 0 to Hdr.NumIcons - 1 do begin
    Move (ImageData^, GroupDir, sizeof (GroupDir));
    { Build a TIconDirEntry for this icon }
    Dir.Width := GroupDir.Width;
    Dir.Height := GroupDir.Height;
    Dir.Colors := GroupDir.Colors;
    Dir.Reserved := 0;
    Dir.dwReserved := 0;
    Dir.dwBytesInRes := GroupDir.ImageSize;
    Dir.dwImageOffset := Offset;
    fs.Write (Dir, sizeof (Dir));
    Inc (ImageData, sizeof (GroupDir));
    Inc (Offset, GroupDir.ImageSize);
  end;
  { Now loop again, writing the image data for each icon }
  ImageData := DirStart;
  for Idx := 0 to Hdr.NumIcons - 1 do begin
    Move (ImageData^, GroupDir, sizeof (GroupDir));
    WriteIndividualIcon(fs, GroupDir.IconID,
      GroupDir.ImageSize);
    Inc (ImageData, sizeof (GroupDir));
  end;
end;
procedure TForm1.WriteSingleIconFile (fs: TFileStream;
  ImageData: PByte; Len: Word);
var
  ImageSize: LongInt;
  bm: TBitmapInfoHeader;
  Hdr: TIconHeader;
  Dir: TIconDirEntry;
begin
  bm := PBitmapInfoHeader (ImageData)^;
  ImageSize := GetIconImageSize (ImageData);
  with Hdr do begin      { Firstly, write file header }
    AlwaysZero := 0;
    Hdr.AlwaysOne := 1;
    Hdr.NumIcons := 1;
    fs.Write (Hdr, sizeof (Hdr));
  end;
  with Dir do begin      { Next, write icon directory entry }
    Width := bm.biWidth;
    Height := bm.biHeight div 2;
    Colors := 1 shl bm.biBitCount;
    Reserved := 0;
    dwReserved := 0;
    dwBytesInRes := ImageSize;
    dwImageOffset := sizeof (Hdr) + sizeof (Dir);
    fs.Write (Dir, sizeof (Dir));
  end;
  { Finally, write the data itself }
  fs.Write (ImageData^, ImageSize);
end;
procedure TForm1.WriteResourceFile (const ResName, TypName:
  String; Info: TResInfo; ResData: Pointer);
var
  fName: String;
  Ext: String [5];
  NumIcons: Integer;
  fs: TFileStream;
begin
  fName := ExtractFilePath (Application.ExeName) + ResName;
  if (TypName = 'ICON') or (TypName = 'GROUPICON') then
    Ext := '.ICO'
  else if (TypName = 'BITMAP') then
    Ext := '.BMP'
  else
    Ext := '.BIN';
  fName := fName + Ext;
  fs := TFileStream.Create (fName, fmCreate);
  try
    if TypName = 'ICON' then
      WriteSingleIconFile (fs, ResData, Info.rLength)
    else if TypName = 'GROUPICON' then
      WriteMultiIconFile (fs, ResData, Info.rLength)
    else if TypName = 'BITMAP' then
      WriteBitmapFile (fs, ResData, Info.rLength)
    else
      fs.Write(ResData^, Info.rLength);
    MessageDlg('Resource has been written to ' + fName,
      mtInformation, [mbok], 0);
  finally
    fs.Free;
  end;
end;
```

The complete code to `WriteBitmapFile` is shown in Listing 2. You'll notice that rather than trying to calculate the exact size of the image data, I 'cheat' and just use the resource length. This adds a few bytes to the end of the bitmap file but, in my experience, I've found it safer to do this than to precisely calculate the image size for a wide variety of different resolutions and colour depths.

Conclusions

In last month's column I showed you how to programmatically examine the resources contained within a 16-bit executable file (EXE or DLL) while this month I've demonstrated how to convert some of those resource types back into industry standard bitmap and icon files. Using the information supplied here, you could have a crack at writing your own icon editor application (naturally, I'll expect a share in the profits!), or maybe come up with a utility to prowl your hard disk for executables, extracting their resources into a directory for later browsing. Using the code I've given you, you could even write a program to alter icons 'in-place' within an existing executable, provided, of course, that the new icon resource wasn't larger than what's already there. The possibilities are endless...

What I haven't done is describe how to do the same sort of thing with 32-bit executables. Although I mentioned this last month, I later remembered that Borland have already provided the code to do this in the shape of their RESXPLOR demo program that comes with Delphi 2 and Delphi 3. If you wanted to, you could take my code and amalgamate with Borland's, to create a unit which transparently handles resources across both 16-bit and 32-bit executables.

Dave Jewell is a freelance consultant/programmer and technical journalist specialising in system-level Windows and DOS work. He is the author of *Instant Delphi Programming* published by Wrox Press. You can contact Dave as DaveJewell@msn.com, DSJewell@aol.com or even DaveJewell@compuserve.com

➤ *Listing 2*

```
procedure TForm1.WriteBitmapFile (fs: TFileStream; ImageData: PByte; Len: Word);
var Hdr: TBitmapFileHeader;
    pbm: PBitmapInfoHeader absolute ImageData;
begin
  { Initialise and write the file header }
  Hdr.bfType := $4D42;
  Hdr.bfSize := sizeof (Hdr) + Len;
  Hdr.bfReserved1 := 0;
  Hdr.bfReserved2 := 0;
  Hdr.bfOffbits := sizeof (Hdr) + sizeof (TBitmapInfoHeader) +
    (sizeof(TRGBQuad) * LongInt (1 shl pbm^.biBitCount));
  fs.Write (Hdr, sizeof (Hdr));
  { Then write the data }
  pbm^.biClrImportant := LongInt (1 shl pbm^.biBitCount);
  fs.Write (ImageData^, Len);
end;
```