The .bmp file format

Introduction:

The .bmp file format (sometimes also saved as .dib) is the standard for a Windows 3.0 or later DIB(device independent bitmap) file. It may use compression (though I never came across a compressed .bmp-file) and is (by itself) not capable of storing animation. However, you can animate a bitmap using different methods but you have to write the code which performs the animation. There are different ways to compress a .bmp-file, but I won't explain them here because they are so rarely used. The image data itself can either contain pointers to entries in a color table or literal RGB values (this is explained later).

Basic structure:

A .bmp file contains of the following data structures:

```
BITMAPFILEHEADER    bmfh;
BITMAPINFOHEADER    bmih;
RGBQUAD             aColors[];
BYTE                aBitmapBits[];
```

bmfh contains some information about the bitmap file (about the file, not about the bitmap itself). *bmih* contains information about the bitmap such as size, colors,... The *aColors array* contains a color table. The rest is the image data, which format is specified by the *bmih* structure.

Exact structure:

The following tables give exact information about the data structures and also contain the settings for a bitmap with the following dimensions: size 100x100, 256 colors, no compression. The *start*-value is the position of the byte in the file at which the explained data element of the structure starts, the *size*-value contains the nuber of bytes used by this data element, the *name*-value is the name assigned to this data element by the Microsoft API documentation. *Stdvalue* stands for standard value. There actually is no such a thing as a standard value but this is the value Paint assigns to the data element if using the bitmap dimensions specified above (100x100x256). The *meaning*-column gives a short explanation of the purpose of this data element.

The BITMAPFILEHEADER:

start	size	name	stdvalue	purpose
1	2	bfType	19778	must always be set to 'BM' to declare that this is a .bmp-file.
3	4	bfSize	??	specifies the size of the file in bytes.
7	2	bfReserved1	0	must always be set to zero.
9	2	bfReserved2	0	must always be set to zero.
11	4	bfOffBits	1078	specifies the offset from the beginning of the file to the bitmap data.

The BITMAPINFOHEADER:

start	size	name	stdvalue	purpose
15	4	biSize	40	specifies the size of the BITMAPINFOHEADER structure, in bytes.
19	4	biWidth	100	specifies the width of the image, in pixels.
23	4	biHeight	100	specifies the height of the image, in pixels.
27	2	biPlanes	1	specifies the number of planes of the target device, must be set to zero.
29	2	biBitCount	8	specifies the number of bits per pixel.
31	4	biCompression	0	Specifies the type of compression, usually set to zero (no compression).
35	4	biSizeImage	0	specifies the size of the image data, in bytes. If there is no compression, it is valid to set this member to zero.
39	4	biXPelsPerMeter	0	specifies the the horizontal pixels per meter on the designated targer device, usually set to zero.
43	4	biYPelsPerMeter	0	specifies the the vertical pixels per meter on the designated targer device, usually set to zero.
47	4	biClrUsed	0	specifies the number of colors used in the bitmap, if set to zero the number of colors is calculated using the biBitCount member.
51	4	biClrImportant	0	specifies the number of color that are 'important' for the bitmap, if set to zero, all colors are important.

Note that *biBitCount* actually specifies the color resolution of the bitmap. The possible values are: 1 (black/white); 4 (16 colors); 8 (256 colors); 24 (16.7 million colors). The biBitCount data element also decides if there is a color table in the file and how it looks like. In 1-bit mode the color table has to contain 2 entries (usually white and black). If a bit in the image data is clear, it points to the first palette entry. If the bit is set, it points to the second. In 4-bit mode the color table must contain 16 colors. Every byte in the image data represents two pixels. The byte is split into the higher 4 bits and the lower 4 bits and each value of them points to a palette entry. There are also standard colors for 16 colors mode (16 out of Windows 20 reserved colors (without the entries 8, 9, 246, 247)). Note that you do not need to use this standard colors if the bitmap is to be displayed on a screen which support 256 colors or more, however (nearly) every 4-bit image uses this standard colors. In 8-bit mode every byte represents a pixel. The value points to an entry in the color table which contains 256 entries (for details see Palettes in Windows. In 24-bit mode three bytes represent one pixel. The first byte represents the red part, the second the green and the third the blue part. There is no need for a palette because every pixel contains a literal RGB-value, so the palette is omitted.

The RGBQUAD array:

The following table shows a single RGBQUAD structure:

start	size	name	stdvalue	purpose
1	1	rgbBlue	-	specifies the blue part of the color.
2	1	rgbGreen	-	specifies the green part of the color.
3	1	rgbRed	-	specifies the red part of the color.
4	1	rgbReserved	-	must always be set to zero.

Note that the term *palette* does not refer to a RGBQUAD array, which is called *color table* instead. Also note that, in a color table (RGBQUAD), the specification for a color starts with the blue byte. In a palette a color always starts with the red byte. There is no simple way to map the whole color table into a LOGPALETTE structure, which you will need to display the bitmap. You will have to write a function that copies byte after byte.

The pixel data:

It depens on the BITMAPINFOHEADER structure how the pixel data is to be interpreted (see above).
It is important to know that the rows of a DIB are stored upside down. That means that the uppest row which appears on the screen actually is the lowest row stored in the bitmap, a short example:

pixels displayed on the screen

pixels stored in .bmp-file

You do not need to turn around the rows manually. The API functions which also display the bitmap will do that for you automatically.
Another important thing is that the number of bytes in one row must always be adjusted to fit into the border of a multiple of four. You simply append zero bytes until the number of bytes in a row reaches a multiple of four, an example:

```
 6 bytes that represent a row in the bitmap:  A0 37 F2 8B 31 C4
                         must be saved as:  A0 37 F2 8B 31 C4 00 00
```

to reach the multiple of four which is the next higher after six (eight). If you keep these few rules in mind while working with .bmp files it should be easy for you, to master it.

Back to the main page
