### [**Bitmap File Internal Structure Overview**](https://www.codeguru.com/cpp/g-m/bitmap/article.php/c4909/Exploring-the-Internal-Structure-of-a-24Bit-Uncompressed-Bitmap-File.htm)

A bitmap file consists of four different parts. The first structure is the BITMAPFILEHEADER structure. You can check the Visual Studio's Help file for this structure:

**typedef struct** tagBITMAPFILEHEADER {

**WORD** bfType;

**DWORD** bfSize;

**WORD** bfReserved1;

**WORD** bfReserved2;

**DWORD** bfOffBits;

} BITMAPFILEHEADER, \*PBITMAPFILEHEADER;

Here are explanations for these member variables:

* "**bfType**": The type of this file usually should be two letters, "B" and "M," as two bytes (combined, it is one WORD with the value "B" as the upper 8 bits and "M" as the lower 8 bits).
* "**bfSize**": The size of the file, in bytes. If you right-click a bitmap file, then select Property and check the size of the file (not the actual size on the disk), it should be the same as what this variable contains. This variable is extremely useful.
* "**bfReserved1**" and "**bfReserved2**": Useless; should be 0 at all times.
* "bfOffBits": This variable indicates how many bytes are from the beginning of the file to the actual pixels. On my computer, it always returns at 54 (14 bytes for BITMAPFILEHEADER and 40 bytes for BITMAPINFO). Other rumors say this variable should be only 40. I guess it should be right too for some cases. But, on my computer, this is always 54.

After this structure, you will encounter another structure, called BITMAPINFO. Every 24-bit, non-compressed bitmap is written to the disk like this: The first part of the bitmap is the BITMAPFILEHEADER and the next part is the BITMAPINFO. Visual Studio describes BITMAPINFO like this:

**typedef struct** tagBITMAPINFO {

**BITMAPINFOHEADER** bmiHeader;

**RGBQUAD** bmiColors[1];

} BITMAPINFO, \*PBITMAPINFO;

This is actually two structures put together with the name of BITMAPINFO. **When you process this structure, you really don't want to process them all together using a single fread(). Instead, process them one at a time (I guess any programmer would know that).**

First, let me explain simply what BITMAPINFOHEADER is. It is described like this in Visual Studio:

**typedef struct** tagBITMAPINFOHEADER {

**DWORD biSize;**

**LONG biWidth;**

**LONG biHeight;**

**WORD biPlanes;**

**WORD biBitCount;**

**DWORD biCompression;**

**DWORD biSizeImage;**

**LONG biXPelsPerMeter;**

**LONG biYPelsPerMeter;**

**DWORD biClrUsed;**

**DWORD biClrImportant;**

} BITMAPINFOHEADER, \*PBITMAPINFOHEADER;

Explanations for these member variables are in the following table:

|  |  |  |
| --- | --- | --- |
| **Variable** | **Description** | **Value** |
| biSize | The size of the structure. Basically equal to size of(BITMAPINFOHEADER). | It should be 40. |
| biWidth | The width of the image |  |
| biHeight | The height of the image |  |
| biPlanes | The number of planes of the bitmap. | In our case it is 1; I guess in most cases it should be 1. |
| biBitCount | The number of bites per pixel: 1bit (two colors black and white), 4 bits (16 colors using a color lookup table stored in RGBQUAD), 8 bits (256 colors using a color lookup table stored in RGBQUAD), and finally 24 bits (2^24 colors using 3 bytes, each for red, green, and blue). | In our case it is 24. |
| biCompression | 0 denotes no compression, 1 to 3 denote three different RLE compression methods: 1 for RLE 8 bits compression, 2 for RLE 4 bits compression, and 3 for bit fields. I don't know much about compressions (except they save spaces). | In our case it is 0. |
| biSizeImage | The total size of the image (the number of bytes from the first pixel of the file to the last pixel of the file). Note: This variable may not be equal to biWidth times biHeight. |  |
| biXPelsPerMeter | The number of pixels in one meter in x-axis. | This is 0 in our case. |
| biYPelsPerMeter | The number of pixels in one meter in y-axis. | This is 0 too in our case. |
| biClrUsed | The number of colors used in this bitmap. | In our case, it is 0. |
| biClrImportant | The number of colors that is important for this bitmap. | In our case, it is 0. |

After the BITMAPINFOHEADER structure comes a RGBQUAD variable; this is a color table. It is commonly either 16 colors or 256 colors, depending on how the file is specified in BITMAPINFOHEADER's biBitCount. In our case, it does not exist in the file because we are using a 24-bit bitmap.

After BITMAPINFO, the remainder of the file is used to store pixel points. It is a linear array of bytes occupying the rest of the file. For 24-bit uncompressed, this array of bytes seems to be nothing more than just RGB color values. That is not true at all; for certain situations, each scan line of the bitmap is separated by 1, 2, or even 3 zeroes. These are junk bytes. (I don't know what else to call them.) The bytes shouldn't be read and used as pixels. If you do use them as pixels, it will mess up the orders of pixels and generate a distorted image. Man, did I learn this lesson the hard way.

### 

### 

### 

### 

### **Just One Last Thing about Bitmaps**

I don't know who set the standard or why, but the image is usually stored in inverted order. That is, the image is stored so that the top left corner is stored at the end of the file, and the bottom right corner is stored as the first pixel after BITMAPINFO. Thus, the starting pixel is the lower-right corner of the bitmap, and the order of all pixel RGB bytes are B-G-R. The following table illustrates this idea:

Image 1

The actual image:

four colors:

1. Red (255, 0, 0)

2. Green (0, 255, 0)

3. Blue (0, 0, 255)

4. Purple Blue (128, 0, 255)

Image 2

The inverted image stored:

four colors

1. Red (255, 0, 0)

2. Green (0, 255, 0)

3. Blue (0, 0, 255)

4. Purple Blue (128, 0, 255)

Image 3

The actual inverted image:

Inside the file, all colors were written inverted:

1. Red (255, 0, 0) -> Blue (0, 0, 255)

2. Green (0, 255, 0) -> Still green

3. Blue (0, 0, 255) -> Red (255, 0, 0)

4. Purple Blue (128, 0, 255) -> Pink (255, 0, 128)

Image 1 is the actual image when you open a bitmap file with the Windows Paint program or any other image editor. Image 2 is a false image of Image 1, stored as a file. The reason for the word "false" is that the RGB bytes are not inverted. Image 3 is the actual image in the file if you read the file from the starting pixel to the end of file. It is like writing "I am an idiot!" backwards—"!toidi na ma I". So if a bitmap's pixels bits are like this:

(255, 0, 0) (0, 255, 0) (0, 0, 255)

in the real file, the pixels are stored like this:

(255, 0, 0) (0, 255, 0), (0, 0, 255) exactly backwards.

### **How the Pixels Should be Read from the File to Represent the Correct Image**

This is where most people would fall. We now know the image is inversely stored as a file. But not many people know to make the total pixels equal an even number; junk bytes (useless bytes) are used to fill the number of pixels to be an even number. I don't know if this is the correct analogy. As far as I have observed, this is true. For example:

This is a simple 31 X 31 pixels^2 24-bit uncompressed bitmap. Its total size on disk is 3,030 bytes. Let's work out some mathematics: 31 \* 31 \* 3 = 2883 bytes (Note: times 3 in the end because we are counting bytes, and each pixel consists 3 bytes; each byte denotes R, G, and B values), plus two structures, BITMAPFILEHEADER and BITMAPINFOHEADER (14 + 40 bytes = 54 bytes). The total size of file ideally should be: 2883 + 54 = 2937 bytes. Compare to 3,030 bytes, there are 93 extra bytes. These are the junk bytes. They could be found at the end of each pixel line (I would refer these pixel lines as scan lines).

Let's check another example:



This is a 7 by 7 pixels, 24-bit, uncompressed bitmap. Its total size on disk is 222 bytes. This actual size is larger than the ideal size of the bitmap. A little simple math can demonstrate this: 7 X 7 X 3 + 14 + 40 = 201. 7 and 7 is the width and height in pixels of the image. 3 means each pixel has 3 bytes to represent the values of red, green, and blue; 14 is the size of BITMAPFILEHEADER, and 40 is the size of BITMAPINFOHEADER. Compared to the actual size, there are 21 extra bytes. These are the junk bytes, which were written as 0's. By looking at the image in hexadecimal format, you can see these junk bytes, as the underlined 3 "00"s:

......

80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 80 FF 80 00 00 00

As you use various sizes of 24-bit, uncompressed bitmaps, you will see these junk bytes sometimes can be only one "00" in each scan line, or sometimes it is two "00"s in each scan line, and sometimes it is three "00"s in each scan line. I haven't seen four or a larger number of "00"s as junk bytes in each scan line. I guess it is not necessary to add such a large number of "00"s as junk bytes in each scan line. Anyway, no matter how many bytes are used as junk bytes, they must be ignored. And it is very easy to determine how many such bytes need to be ignored in each line. Again, simple math is used to determine the number of bytes that must be ignored:

(Total size of bitmap on disk - (width \* height \* 3 + 14 + 40)) / height = number of bytes needed to be ignored during each line scan.

### **How to Parse a 24-bit, Uncompressed Bitmap File—The Source Code**

We've explored every trick used to create and possibly encrypt an image into a bitmap. It is very easy to read a 24-bit uncompressed bitmap without the help of any WIN32, MFC, or other APIs to parse such a bitmap file. The following code is how you can do it:

1. BOOL CBitmaploadDoc::LoadBitmap24(const char \* fn)
2. {
3. FILE \* fp;
4. BYTE r, g, b;
5. BITMAPFILEHEADER bmFileHdr;
6. BITMAPINFO bmInfo;
7. int EndByte = 1;
8. BOOL bDone = FALSE;
9. WORD \* temp;
10. WORD \* t2;
12. // open file to read

fp = fopen(fn, "rb");

2. // read BITMAPFILEHEADER
3. fread((char \*)&bmFileHdr, sizeof(BITMAPFILEHEADER), 1, fp);
5. // you read BITMAPINFOHEADER and RGBQUAD separately or together
6. // by reading them into the BITMAPINFO structure. When
7. // RGBQUAD is not empty (bitmap is 16 colors or 256 colors),
8. // you must read these two structures separately.
9. fread((char \*)&bmInfo, sizeof(BITMAPINFO), 1, fp);
11. // check the bitmap info. If it is not 24-bit, and uncompressed,
12. //and the colors used and are important is not 0, return FALSE.
13. if (bmInfo.bmiHeader.biBitCount != 24 ||
14. bmInfo.bmiHeader.biCompression != 0 ||
15. (bmInfo.bmiHeader.biClrUsed != 0 &&
16. bmInfo.bmiHeader.biClrImportant != 0))
17. {
18. fclose(fp);
19. AfxMessageBox("This does not appear to be a 24-bit
20. uncompressedbitmap.");
21. return FALSE;
22. }
24. // get the width and height of the bitmap.
25. bm\_width = (WORD)bmInfo.bmiHeader.biWidth; // this is a class
26. // member
27. bm\_height = (WORD)bmInfo.bmiHeader.biHeight; // this is a class
28. // member
29. if (bm\_width > 800 || bm\_height > 600)
30. {
31. fclose(fp);
32. AfxMessageBox("This bitmap is way too big. Bitmap
33. shouldn't be larger than 800X600.");
34. return FALSE;
35. }
37. // set new file pointer position at 54 from the beginning of
38. // the file. 0 to 53 are used to store BITMAPFILEINFO and
39. // BITMAPINFOHEADER.
40. fseek(fp, bmFileHdr.bfOffBits, SEEK\_SET);
42. // start the mathematical calculation for junk bytes.
43. if ((bmInfo.bmiHeader.biSizeImage - bm\_width \*
44. bm\_height \* 3) == 0)
45. {
46. EndByte = 0; // only when there are no junk bytes
47. }
48. else
49. {
50. // find junk bytes.
51. EndByte = (bmInfo.bmiHeader.biSizeImage - bm\_width
52. \* bm\_height \* 3)
53. / bm\_height;
54. }
56. // create 2 empty arrays. One stores pixels from the file,
57. // the other is used to invert the array.
58. temp = new WORD[bm\_width \* bm\_height];
59. pBitmapPixels = new WORD[bm\_width \* bm\_height]; // this is a
60. // class member
61. int count = bm\_width \* bm\_height - 1;
63. // read the pixels.
64. while (!feof(fp))
65. {
66. // read scan line
67. for (int i = 0; i < bm\_width; i++)
68. {
69. // because the bitmap is stored inverted, the color
70. // byte Blue is always the first
71. if (!feof(fp))
72. {
73. fread((BYTE \*)&b, sizeof(BYTE), 1, fp);
74. }
75. else
76. {
77. // this is true only when the file is totally
78. // corrupted.
79. bDone = TRUE;
80. break;
81. }
82. // Then we read the Green byte.
83. if (!feof(fp))
84. {
85. fread((BYTE \*)&g, sizeof(BYTE), 1, fp);
86. }
87. else
88. {
89. // this is true only when the file is totally
90. // corrupted.
91. bDone = TRUE;
92. break;
93. }
94. // Finally, the Red byte.
95. if (!feof(fp))
96. {
97. fread((BYTE \*)&r, sizeof(BYTE), 1, fp);
98. }
99. else
100. {
101. // this is true only when the file is totally
102. // corrupted.
103. bDone = TRUE;
104. break;
105. }
107. // convert 24-bit color (3 bytes for R, G, B) into
108. // 16-bit, 5-6-5 format WORD pixel.
109. temp[count] = (WORD)((((WORD)r&0xf8)<<8)|
110. (((WORD)g&0xfc)<<3)|
111. (((WORD)b&0xf8)>>3));
112. count--; // counter operation for pixel index.
113. }
115. // When the for loop is done, one scan line is completed.
116. // This is where you skip the junk bytes. If you don't do
117. // that, the end result will be a distorted image.
118. // Not good.
119. if (!bDone)
120. {
121. fseek(fp, EndByte, SEEK\_CUR);
122. }
123. else
124. {
125. break;
126. }
127. }
129. // We do another inversion of the array.
130. count = 0;
131. for (int y = 0; y < bm\_height; y++)
132. {
133. for (int x = bm\_width - 1; x > -1; x--)
134. {
135. t2 = temp + y \* bm\_width + x;
136. pBitmapPixels[count] = \*t2;
137. count++;
138. }
139. }
140. delete [] temp; // when done, the old array is discarded.
142. fclose(fp); // close file
144. return TRUE; // return success.
145. }