<https://www.cnblogs.com/wainiwann/p/7086844.html>

**BMP文件格式详解（BMP file format）**

BMP文件格式，又称为[Bitmap](https://so.csdn.net/so/search?q=Bitmap&spm=1001.2101.3001.7020" \t "_blank)（位图）或是DIB(Device-Independent Device，设备无关位图)，是Windows系统中广泛使用的图像文件格式。由于它可以不作任何变换地保存图像像素域的数据，因此成为我们取得RAW数据的重要来源。Windows的图形用户界面（graphical user interfaces）也在它的内建图像子系统GDI中对BMP格式提供了支持。

下面以Notepad++为分析工具，结合Windows的[位图](https://so.csdn.net/so/search?q=%E4%BD%8D%E5%9B%BE&spm=1001.2101.3001.7020" \t "_blank)[数据结构](http://lib.csdn.net/base/31)对BMP文件格式进行一个深度的剖析。

BMP文件的数据按照从文件头开始的先后顺序分为四个部分：

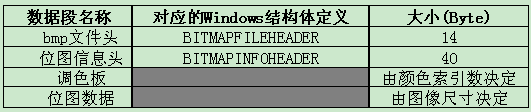
Ø         **bmp文件头(bmp file header)：**提供文件的格式、大小等信息

Ø         **位图信息头(bitmap information)：**提供图像数据的尺寸、位平面数、压缩方式、颜色索引等信息

Ø         **调色板(color palette)：**可选，如使用索引来表示图像，调色板就是索引与其对应的颜色的映射表

Ø         **位图数据(bitmap data)：**就是图像数据啦^\_^

下面结合Windows结构体的定义，通过一个表来分析这四个部分。



我们一般见到的图像以24位图像为主，即R、G、B三种颜色各用8个bit来表示，这样的图像我们称为真彩色，这种情况下是不需要调色板的，也就是所位图信息头后面紧跟的就是位图数据了。因此，我们常常见到有这样一种说法：**位图文件从文件头开始偏移54个字节就是位图数据了，这其实说的是24或32位图的情况。这也就解释了我们按照这种程序写出来的程序为什么对某些位图文件没用了。**

 下面针对一幅特定的图像进行分析，来看看在位图文件中这四个数据段的排布以及组成。

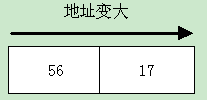
  我们使用的图像显示如下：



   这是一幅16位的位图文件，因此它是含有调色板的。

   在拉出图像数据进行分析之前，我们首先进行几个约定：

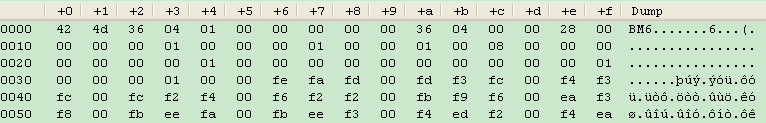
   1. 在BMP文件中，如果一个数据需要用几个字节来表示的话，那么该数据的存放字节顺序为“低地址村存放低位数据，高地址存放高位数据”。如数据0x1756在内存中的存储顺序为：



这种存储方式称为小端方式(little endian) , 与之相反的是大端方式（big endian）。对两者的使用情况有兴趣的可以深究一下，其中还是有学问的。

2.  以下所有分析均以字节为序号单位进行。

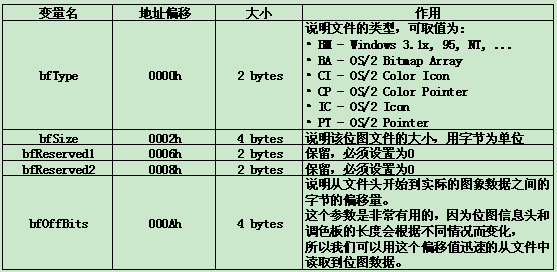
   下面我们对从文件中拉出来的数据进行剖析：



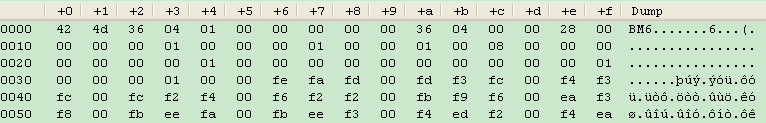
一、**bmp文件头**  
Windows为bmp文件头定义了如下结构体：

typedef struct tagBITMAPFILEHEADER   
{    
UINT16 bfType;      
DWORD bfSize;   
UINT16 bfReserved1;   
UINT16 bfReserved2;   
DWORD bfOffBits;  
} BITMAPFILEHEADER;

其中：



 对照文件数据我们看到：



1-2  ：424dh = 'BM',表示这是Windows支持的位图格式。有很多声称开头两个字节必须为'BM'才是位图文件，从上表来看应为开头两个字节必须为'BM'才是Windows位图文件。

3-5  ：00010436h = 66614 B = 65.05 kB，通过查询文件属性发现一致。

6-9  ：这是两个保留段，为0。

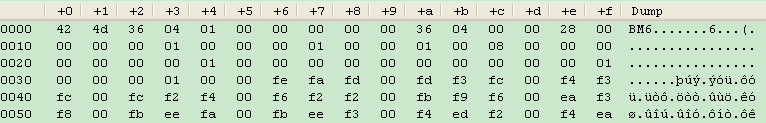
A-D：00000436h = 1078。即从文件头到位图数据需偏移1078字节。我们稍后将验证这个数据。

共有14个字节。

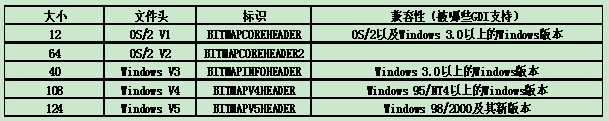
二、**位图信息头**  
同样地，Windows为位图信息头定义了如下结构体：

****

**对照数据**文件：

****

0E-11：00000028h = 40,这就是说我这个位图信息头的大小为40个字节。前面我们已经说过位图信息头一般有40个字节，既然是这样，为什么这里还要给一个字段来说明呢？这里涉及到一些历史,其实位图信息头原本有很多大小的版本的。我们看一下下表：



    出于兼容性的考虑，大多数应用使用了旧版的位图信息头来保存文件。而 OS/2 已经过时了,因此现在最常用的格式就仅有V3 header了。因此，我们在前面说位图信息头的大小为40字节。

12-15：00000100h = 256，图像宽为255像素，与文件属性一致。

16-19：00000100h = 256，图像高为255像素，与文件属性一致。这是一个正数，说明图像数据是从图像左下角到右上角排列的。

1A-1B：0001h, 该值总为1。

1C-1D：0008h = 8, 表示每个像素占8个比特，即该图像共有256种颜色。

1E-21：00000000h，BI\_RGB， 说明本图像不压缩。

22-25：00000000h，图像的大小，因为使用BI\_RGB，所以设置为0。

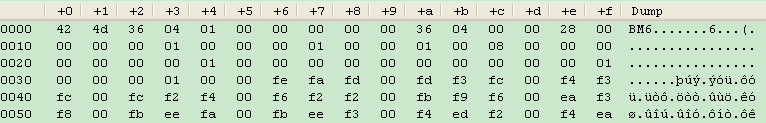
26-29：00000000h，水平分辨率，缺省。

2A-2D：00000000h，垂直分辨率，缺省。

2E-31：00000100h = 256,说明本位图实际使用的颜色索引数为256，与1C-ID得到的结论一致。

32-35：00000100h = 256,说明本位图重要的颜色索引数为256，与前面得到的结论一致。

三、**调色板**  
下面的数据就是调色板了。前面也已经提过，调色板其实是一张映射表，标识颜色索引号与其代表的颜色的对应关系。它在文件中的布局就像一个二维数组palette[N][4],其中N表示总的颜色索引数，每行的四个元素分别表示该索引对应的B、G、R和Alpha的值，每个分量占一个字节。如不设透明通道时，Alpha为0。因为前面知道，本图有256个颜色索引，因此N = 256。索引号就是所在行的行号，对应的颜色就是所在行的四个元素。这里截取一些数据来说明：

****

**索引：(蓝，绿，红，Alpha)**

0号：(fe，fa，fd，00)

1号：(fd，f3，fc，00)

2号：(f4，f3，fc，00)

3号：(fc，f2，f4，00)

4号：(f6，f2，f2，00)

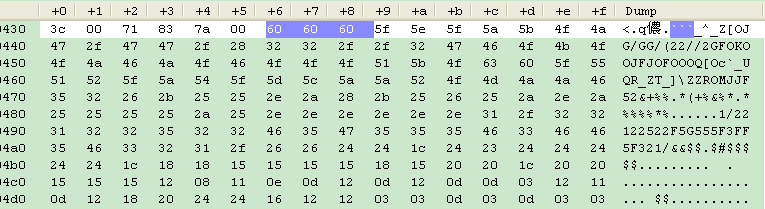
5号：(fb，f9，f6，00) 等等。

 一共有256种颜色，每个颜色占用4个字节，就是一共1024个字节，再加上前面的文件信息头和位图信息头的54个字节加起来一共是1078个字节。也就是说在位图数据出现之前一共有1078个字节，与我们在文件信息头得到的信息：文件头到文图数据区的偏移为1078个字节一致！

四、**位图数据**

下面就是位图数据了，每个像素占一个字节，取得这个字节后，以该字节为索引查询相应的颜色，并显示到相应的显示设备上就可以了。

注意：由于位图信息头中的图像高度是正数，所以位图数据在文件中的排列顺序是从左下角到右上角，以行为主序排列的。



也即我们见到的第一个像素60是图像最左下角的数据，第二个人像素60为图像最后一行第二列的数据，…一直到最后一行的最后一列数据，后面紧接的是倒数第二行的第一列的数据，依此类推。

 如果图像是24位或是32位数据的位图的话，位图数据区就不是索引而是实际的像素值了。下面说明一下，此时位图数据区的每个像素的RGB颜色阵列排布：

24位RGB按照BGR的顺序来存储每个像素的各颜色通道的值，一个像素的所有颜色分量值都存完后才存下一个下一个像素，不进行交织存储。

32位数据按照BGRA的顺序存储，其余与24位位图的方式一样。

像素的排布规则与前述一致。

**对齐规则**

讲完了像素的排列规则以及各像素的颜色分量的排列规则，最后我们谈谈数据的对齐规则。我们知道Windows默认的扫描的最小单位是4字节，如果数据对齐满足这个值的话对于数据的获取速度等都是有很大的增益的。因此，BMP图像顺应了这个要求，要求每行的数据的长度必须是4的倍数，如果不够需要进行比特填充（以0填充），这样可以达到按行的快速存取。这时，位图数据区的大小就未必是 图片宽×每像素字节数×图片高 能表示的了，因为每行可能还需要进行比特填充。

填充后的每行的字节数为：

，其中BPP（Bits Per Pixel）为每像素的比特数。

在程序中，我们可以表示为：

**int iLineByteCnt = (((m\_iImageWidth \* m\_iBitsPerPixel) + 31) >> 5) << 2;**

这样，位图数据区的大小为：

**m\_iImageDataSize = iLineByteCnt \* m\_iImageHeight;**

我们在扫描完一行数据后，也可能接下来的数据并不是下一行的数据，可能需要跳过一段填充数据：

**skip = 4 - ((m\_iImageWidth \* m\_iBitsPerPixel)>>3) & 3;**

**五、拾遗**

至此，我们通过分析一个具体的位图文件例子详细地剖析了位图文件的组成。需要注意的是：我们讲的主要是PC机上的位图文件的构成，对于[嵌入式](http://lib.csdn.net/base/embeddeddevelopment)平台，可能在调色板数据段与PC机的不同。如在嵌入式平台上常见的16位r5g6b5位图实际上采用的掩模的方式而不是索引的方式来表示图像。此时，在调色板数据段共有四个部分，每个部分为四个字节，实际表示的是彩色版规范。即：

  第一个部分是红色分量的掩模

  第二个部分是绿色分量的掩模

 第三个部分是蓝色分量的掩模

 第四个部分是Alpha分量的掩模（缺省为0）

典型的调色板规范在文件中的顺序为为：

  00F8 0000 E007 0000 1F00 0000 0000 0000

其中

    00F8 0000为FB00h=1111100000000000（二进制），是蓝红分量的掩码。   
　　E007 0000为 07E0h=0000011111100000（二进制），是绿色分量的掩码。   
　  1F00 0000为001Fh=0000000000011111（二进制），是蓝色分量的掩码。   
　   0000 0000设置为0。

将掩码跟像素值进行“与”运算再进行移位操作就可以得到各色分量值。看看掩码，就可以明白事实上在每个像素值的两个字节16位中，按从高到低取5、6、5位分别就是r、g、b分量值。取出分量值后把r、g、b值分别乘以8、4、8就可以补齐每个分量为一个字节，再把这三个字节按BGR组合，放入存储器，就可以转换为24位标准BMP格式了。

这样我们假设在位图数据区有一个像素的数据在文件中表示为02 F1。这个数据实际上应为F102：

 r = (F102 AND F800) >> 8 = F0h = 240

 g= (F102 AND 07E0）>> 3 = 20h = 32   
　 b=(F102 AND 001F) << 3 = 10h =16

至此我们就可以显示了。（本文结束）

**前言**

记得本科时候讲《计算机体系结构》的老师（很遗憾忘了他姓名）评价过中外教材的差别，他说按照老外的体系结构教材，你就真的能够做出一个CPU来（虽然只能做出很老很老的CPU），但国内的教材就很难教到这个程度。

几个月前我从零开始写了一个简单的bmp解码库，现在用一篇[文章](http://www.it-crazy.com/)把其中的关键内容记录下来，希望能够达到让别人照着文章就可以开发出任何语言绑定的bmp解码库的程度，以便日后我可以放心地忘却，因为我的脑子里总是不能同时记住太多的东西。

**BMP简介**

BMP是一种与硬件设备无关的图像文件格式，使用非常广。BMP是Windows环境中交换与图有关的数据的一种标准，在Windows环境中运行的图形图像软件都支持BMP图像格式。

相对来讲，BMP格式比较简单，它只包含两个重要参数：编码格式（Encoding）和像素位数（bpp, bit-per-pixel）。到目前为止，BMP格式所支持的所有像素位数与编码格式的组合如下：

|  |  |  |
| --- | --- | --- |
| 序号 | 像素位数(bpp) | 编码格式(Encoding) |
| 1 | 1 | bit |
| 2 | 4 | bgr(blue-green-red) |
| 3 | 4 | rle(run-length encode) |
| 4 | 8 | bgr |
| 5 | 8 | rle |
| 6 | grayscale | bgr |
| 7 | grayscale | rle |
| 8 | 16 | bgr |
| 9 | 16 | bitfields-555 |
| 10 | 16 | bitfields-565 |
| 11 | 16 | bitfields-customized |
| 12 | 24 | bgr |
| 13 | 32 | bgr |
| 14 | 32 | bitfields-888 |
| 15 | 32 | bitfields-customized |

其中24bpp称为真彩（true-color）图像，应用最为广泛。16bpp的bmp图像拥有存储空间小，解析速度快，仿真彩效果好等特点，经常出现在游戏软件中。grayscale（灰度）图像其实是8bpp的一种情况。

**文件结构**

典型的BMP图像文件由四部分组成：  
1：文件头，它包含BMP图像文件的类型、内容尺寸和起始偏移量等信息；  
2：图像参数，它包含图像的宽、高、压缩方法，以及颜色定义等信息；  
3：调色板，可选部分，bpp较小的位图需要调色板；有些位图，比如24bpp（真彩色）图就不需要调色板；  
4：位图数据，这部分的内容因位图实际像素位数和编码格式而不同，在真彩位图中直接使用RGB真彩色值；而有调色板的位图则使用调色板中颜色索引值。

**数据类型**

先定义几个数据类型以便于描述。  
byte —— 8 bits  
word —— 16 bits  
int/uint/dword —— 32 bits

**文件头**

BMP的文件头共14个字节。

|  |  |  |
| --- | --- | --- |
| 字节顺序 | [数据结构](http://lib.csdn.net/base/datastructure) | 描述 |
| 1,2 | word | 高8位为字母’B’，低8位为字母’M’ |
| 3,4,5,6 | uint | 文件尺寸 |
| 7,8 | word | 保留字1 |
| 9,10 | word | 保留字2 |
| 11,12,13,14 | uint | 位图数据部分相对于文件首的起始偏移量 |

数据部分偏移量的存在，说明图像数据部分并不一定要紧随图像参数或调色板之后放置，BMP图片的制作者其实可以在调色板之后、数据部分之前填充任何内容，只要正确地设置偏移量即可。

**图像参数信息**

这一个数据块共40字节或56字节。前40字节的内容如下：

|  |  |  |
| --- | --- | --- |
| 字节顺序 | 数据结构 | 描述 |
| 15,16,17,18 | uint | 当前结构体的大小，通常是40或56 |
| 19,20,21,22 | int | 图像宽度（像素） |
| 23,24,25,26 | int | 图像高度（像素） |
| 27,28 | word | 这个字的值永远是1 |
| 29,30 | word | 每像素占用的位数，即bpp |
| 31,32,33,34 | uint | 压缩方式 |
| 35,36,37,38 | uint | 图像的尺寸（字节数） |
| 39,40,41,42 | int | 水平分辨率，pixels-per-meter |
| 43,44,45,46 | int | 垂直分辨率，pixels-per-meter |
| 47,48,49,50 | uint | 引用色彩数 |
| 51,52,53,54 | uint | 关键色彩数 |

水平分辨率和垂直分辨率我从来没用过。看上去应该是与设备相关的参数。

如果你是一个有优化癖的[程序](http://www.xuebuyuan.com/)员，你一定会问，图像的宽和高为什么是int型而不是uint型呢？因为想象中负数宽和高似乎没有意义。比较滑稽的是，在BMP格式中，负数的高是有意义的。为了与高搭配，因此图像的宽也定义为int型。负数高的意义我们将在图像数据块中讨论。

第31-34字节存储着一个uint型参数，它代表图像数据的压缩方式。该参数的取值范围是0、1、2或3等等。这些取值的含义分别是：

0 —— RGB方式  
1 —— 8bpp的run-length-encoding方式  
2 —— 4bpp的run-length-encoding方式  
3 —— bit-fields方式

只有压缩方式选项被置为bit-fields时，当前结构体的大小为56字节，否则为40字节。

**调色板**

当bpp小于等于8时，BMP使用调色板记录色彩信息，调色板中每条数据（即每种色彩值）都是一个uint型数据。当调色板存在时，图像数据块中存储的只是各个像素的色彩在调色板中的索引值，必须通过在调色板中查表，才能获知各个像素的真实颜色。若引入调色板，则调色板数据块紧随在图像参数数据块之后。

当bpp == 1时，调色板合法索引值只有0和1。因此调色板中只有2个色彩值，分别表示索引值为0和1时的色彩信息。

当bpp == 4或bpp == 8时，合法索引值范围扩大为[0,15]和[0,255]。但图像中不一定使用到了全部16种或256种颜色。第47-50字节存储的uint型数据指出图像中实际应用的色彩数，也即调色板中的色彩值数目。当然，它不应超出调色板的合法索引值的范围。

当bpp == 4或bpp == 8时，可以采用Run-Length-Encoding方式压缩图像的存储空间，即压缩方式选项的值为1或2（当选项值为0时，不压缩）。这种编码格式所考虑的情况是，若4bpp或8bpp位图的尺寸较大时，由于色彩总数非常有限，所以图像中必然会有很多颜色重复的像素。因此BMP图像格式的设计者决定采取一个简单的措施来挽回一些被浪费掉的存储空间。这个简单的措施就是RLE压缩方法。

**RLE, Run-Length-Encoding**

如果你是一个有优化癖的程序员，当你发现一张8bpp、宽300像素的位图中有一行像素只有两种颜色：前100个像素是红色，后200个像素是蓝色，然而你的位图却忠实地用100字节来存储前半行重复的红色，又用200字节来存储后半行重复的蓝色，那么你一定会抓狂到大骂BMP格式的设计者是白痴。

为了避免被骂，BMP格式的设计者想出了这样的办法：先用一个字节来存储重复色彩的数量，再用一个字节来存储这个色彩的值，即用两个字节代表一段颜色重复的像素。并且，他们给重复色彩的数量起了个名字，叫做Run-Length，可能是因为只有在运行时我们才能知道这段重复色彩的长度。由于runlength为0时没有意义，因此设计者把runlength=0当做每行的终止符。于是，同样存储一行300个像素，原先需要300字节，现在只用5个字节就搞定！

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字节 | 1 | 2 | 3 | 4 | 5 |
| 内容 | 100 | red | 200 | blue | 0 |

由于你是一个有着严重优化癖的程序员，所以你对这样粗制滥造的优化方法并不满足，因为你很快发现，如果一张位图中没有连续重复的像素（例如红蓝像素点阵），那么用刚才发明的这个办法存储300个像素，居然要用601字节！当然，这种情况下最好的办法是不用压缩[算法](http://www.xuebuyuan.com/category/%E7%AE%97%E6%B3%95)。可是，如果既有重复像素，又有点阵的情况呢？比如前150像素是重复的绿色，后150像素是红蓝相间的像素点阵。

为了满足你变态的优化癖，BMP格式的设计者只好继续发展这个[算法](http://lib.csdn.net/base/datastructure)。首先，他们保留了“用一个字节来存储重复色彩的数量，再用一个字节来存储这个色彩的值”的设计思路，然后修改了runlength为0的含义。设计者规定，当遇到runlength==0时，我们要继续读取下一个字节，若该字节值为n，意味着后面的n个像素将采用“逐字翻译”的方式来解析，也就是说，这n个像素的前面没有runlength这个字节。用这种方法压缩“前150像素是重复的绿色，后150像素是红蓝相间的像素点阵”的300个像素，只需要154个字节。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 字节 | 1 | 2 | 3 | 4 | 后150个字节 |
| 内容 | 150 | green | 0 | 150 | red,blue,…,red,blue |

这个近似完美的结果中有个小问题：设计BMP格式的天才们把runlength==0的含义修改后，我们就没有行终结符了。不过天才终归是天才，他们发现“逐字翻译”的像素数必须大于2才有意义（想想这是为什么？），因此runlength==0之后的那个字节的值为0、1或2时，目前还没有意义。于是天才们规定，当这个值为0时，表示行结束符；当这个值为1时，表示文件结束符；当这个值为2时，似乎仍然没有什么意义；只有当该值大于等于3时，才是“逐字翻译”。完整的压缩结果是：

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 字节 | 1 | 2 | 3 | 4 | 后150个字节 | 155 | 156 |
| 内容 | 150 | green | 0 | 150 | red,blue,…,red,blue | 0 | 0 |

这就是传说中的Run-Length-Encoding for 8bpp。4bpp的RLE跟8bpp时没有本质差别。

**RGB与Bit-Fields**

当图像中引用的色彩超过256种时，我们就需要16bpp或更高bpp的位图。调色板不适合bpp较大的位图，因此16bpp以上的位图都不使用调色板。不使用调色板的位图图像有两种编码格式：RGB和Bit-Fields（下称BF）。

RGB编码格式是一种均分的思想，使Red、Green、Blue三个颜色分量所包含的信息容量尽可能一样大。

16bpp-RGB：在每个像素所占的16bits中，低5位表示Blue分量；中5为表示Green分量；高5位表示Red分量；最高1位无意义（后来有些应用程序将其视为透明度Alpha分量，但这并不是标准）。所以从低到高的顺序实际上是B-G-R，这也是我在BMP简介的表格里，把RGB的编码方式都写成BGR的原因。

24bpp-RGB：24bpp的位图又称为真彩位图，它通常只有这一种编码格式，在24bits中，低8位表示Blue分量；中8为表示Green分量；高8位表示Red分量。

32bpp-RGB：在32bits中，低24位的编码方式与24bpp位图相同，最高8位用来表示透明度Alpha分量。32bpp的位图尺寸太大，一般只有在图像处理的中间过程中使用。对于需要半透过效果的图像，更好的选择是PNG格式。

BF编码格式与RGB不同，它利用位域操作，人为地确定RGB三分量所包含的信息容量。在图像参数信息模块的介绍中提及，当压缩方式选项置为BF时，图像参数结构体将比平时多出16字节。这16字节实际上是4个dword的位域掩码。按照先后顺序，它们分别是R、G、B、A四个分量的位域掩码。当然如果没有Alpha分量，则Alpha掩码没有实际意义。

位域掩码的作用是：指出像素色彩值中RGB分量，就像子网掩码指出子网网段一样。

16bpp-BF-565：这是BF编码格式最著名和最普遍的应用。它的Red、Green和Blue分量的位域掩码分别是0xF800、0x07E0和0x001F。

我们平时所能够见到的位图中使用BF编码格式的并不多，因为它看上去比较麻烦，而效果也不见得比RGB要好（你能用肉眼分辨出16bpp-RGB和16bpp-BF-565之间的区别吗？）。

BF编码格式的重要应用在于游戏软件。游戏软件通常包含数量庞大的小尺寸图片。如果一张图片中几乎没有Blue分量，那么使用16bpp-RGB格式显然会浪费掉B分量所占的5位。此时若采用16bpp-BF-772格式，只给B分量2位，那么R与G分量都拥有7位的容量，几乎接近真彩图像。因此存储空间小、仿真彩能力强的特点使BF编码格式仍然有着独特的用武之地。

32bpp-BF-xxx：我一直不明白为什么会存在32bpp的位图。如果说32bpp-RGB格式的存在是因为在图像处理过程中存储起来比较高效（不用压缩），那么32bpp-BF又是为什么存在呢？

**图像数据块**

图像数据块从文件头中起始偏移量字段所指出的位置开始，其中存放着位图图像的数据，数据格式由图像参数信息块中的压缩方式选项的取值决定。操作图像数据块时，有一些注意事项：

当压缩方式为RGB时，图像数据块以“行”为单位双字对齐。例如一张宽度为5像素的8bpp的图像，其实际使用的存储空间是每行8个字节。又如一幅4bpp、宽度为5像素的位图图像，其实际使用的存储空间是每行4个字节。

当bpp < 8时，每个字节将存放多个像素的色彩索引，则先出现的像素存放在高位中。例如某4bpp图像第一行像素的顺序是red, green, blue, yellow, …则图像数据块中第一个字节的高4位值为red，低4位值为green；第二字节高4位值为blue，低4位值为yellow。1bpp时的情况以此类推。

还记得前面提到，图像参数里，高度有可能是负值吗？这看上去很逗，但事实是，你见过的大多数位图，其图像参数里的高度都是负值。BMP格式设计者规定，当高度为正值时，图像数据块中记录的第一行像素数据是图像的最后一行；而数据块中最后一行数据才是实际图像的第一行，也就是说，数据块中的行记录与实际图像反序。而当高度为负值时，数据块中的行记录与实际图像才是同序的。