Bitmap Documentation

This short pdf contains everything I was able to gather about Bitmaps, it's hard to find everything and it's hard to understand what some sources mean, this is due to the many updates that were happening over the years. The code will be written entirely in C/C++, most importantly this pdf has no-copyright, this means that you are able to download and edit it as if it was yours, feel free to contribute and mention me if you like ☺. Written by Ali Berro, Github: https://github.com/aliberro39109.

Contents

1. What is a Bitmap?

Bitmap file format or device independent bitmap (DIB) file format is a raster graphic image file format used to store images on system, most commonly Windows. Unlike other image structures like Jpeg or GIF, bitmap wasn't made to be portable. Instead, it was built to easily work with Windows API. As the API changed, so did the bitmap file format, Windows now has many documentation regarding the BMP file format ranging from windows 2.x to OS/2 2.x. Although this was based on Windows internal bitmap data structures, yet it is by now supported by many non-Windows and non-PC applications.

2. File Structure

The bitmap image file consists of a fixed size header, followed by a variable sized one, some may also have an Extra bit mask header and a Color table, after that we have two Gap headers, between them a pixel array header and finally the ICC color profile, which also may or may not exist.

Structure name	Optional	Size	Purpose	Comments
Bitmap file header	No	14 bytes	To store general information about the bitmap image file	Not needed after the file is loaded in memory
DIB header	No	Fixed-size, but with 7 different versions existing	To store detailed information about the bitmap image and define the pixel format	Immediately found after the **Bitmap file header**
Extra bit masks	Yes	12 or 16 bytes	To define the pixel format	Present only in case the DIB header is the *BITMAPINFOHEADER* and the Compression Method member is set to either *BI_BITFIELD* or *BI_ALPHABITFIELDS*
Color table	Semi-optional	Variable size	To define the colors used by the pixel array	Mandatory for color depths $\leq$ 8 bits
Gap1	Yes	Variable size	Structure alignment	An artifact of the File offset to Pixel array in the Bitmap file header

Pixel array	No	Variable size	To define the actual values of the pixels	The pixel format is defined by the DIB header or Extra bit masks. Each row in the Pixel array is padded to a multiple of 4 bytes in size
Gap2	Yes	Variable size	Structure alignment	An artifact of the ICC profile data offset field in the DIB header
ICC color profile		Variable size	To define the color profile for color management	Can also contain a path to an external file containing the color profile. When loaded in memory as “non-packed DIB”, it is located between the color table and Gap1

The above table was taken directly from Wikipedia, from here.

When the bitmap image is loaded into the memory, it becomes a DIB data structure, all of its data is identical to that of the bitmap image, however the Bitmap File Header (first 14 bytes) is not loaded. The DIB header contains information about the image, its width and height and many other stuff that we will explore later. Instead of defining the pixels in a pixel array, we could use numbers to denote a specific color (175 for example for red (255,0,0)) and then we would put this number in the color table, this is referred to as indexing, this can save memory in many scenarios. The color table doesn’t have any other function. The pixel array must begin at a memory address that is a multiple of 4 bytes. When the size of Gap1 and Gap2 is zero, the in memory DIB data structure is referred to as a “Packed DIB”.

2.a Bitmap File Header

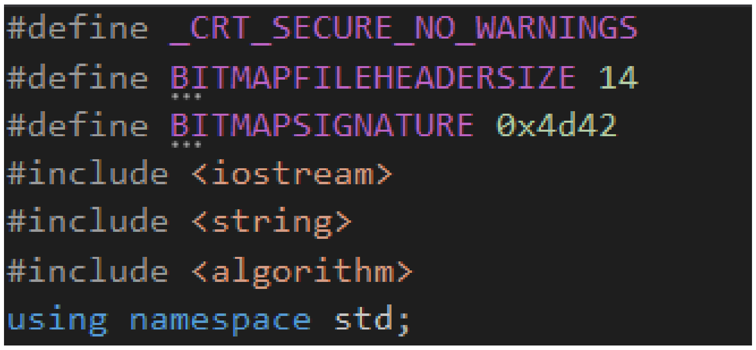
The Bitmap File Header is structured as follows:

Offset decimal	Size	Usage
0	2 bytes	This is used to identify the BMP and DIB file, most commonly is 0x42 0x4D which translates to **BM**, little endian format
2	4 bytes	The size of the BMP file in bytes
6	2 bytes	The value depends on the application in which the image was created
8		The value depends on the application in which the image was created

10	4 bytes	Starting address of the pixel array

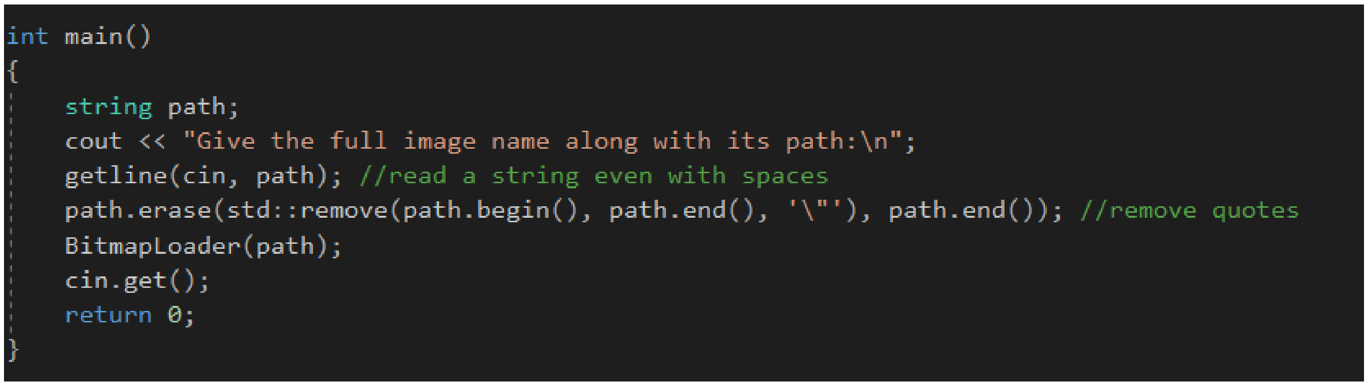
From the above table, we can now load an image in memory and check if it was corrupted or not, furthermore we have found the location of the Pixel array in memory.

Define and load the following:



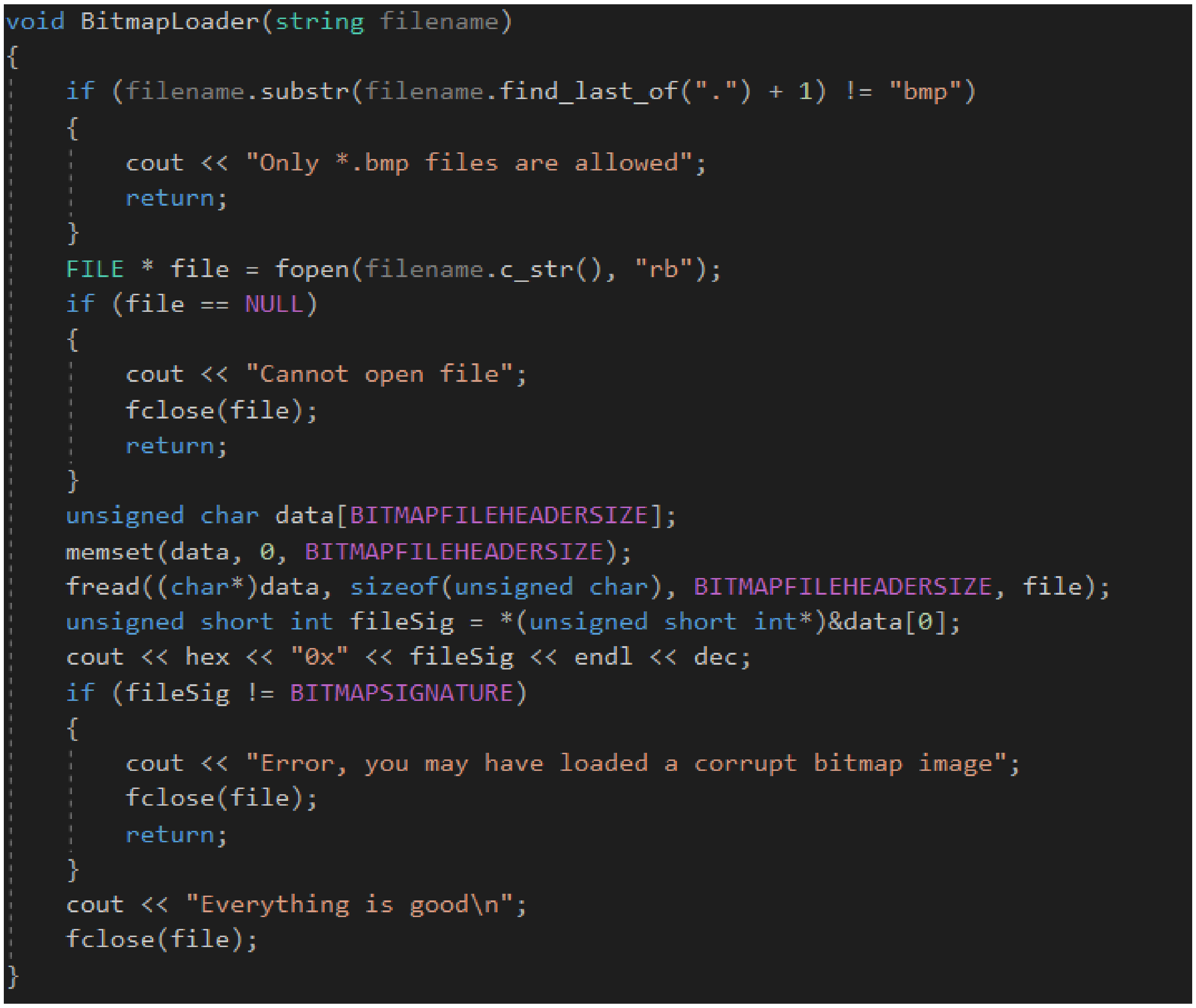
```
#define _CRT_SECURE_NO_WARNINGS
#define BITMAPFILEHEADERSIZE 14
#define BITMAPSIGNATURE 0x4d42
#include <iostream>
#include <string>
#include <algorithm>
using namespace std;
```

Our main function:



```
int main()
{
    string path;
    cout << "Give the full image name along with its path:\n";
    getline(cin, path); //read a string even with spaces
    path.erase(std::remove(path.begin(), path.end(), '\"'), path.end()); //remove quotes
    BitmapLoader(path);
    cin.get();
    return 0;
}
```

Now the BitmapLoader function would look something like this:



```cpp
void BitmapLoader(string filename)
{
    if (filename.substr(filename.find_last_of(".") + 1) != "bmp")
    {
        cout << "Only *.bmp files are allowed";
        return;
    }
    FILE * file = fopen(filename.c_str(), "rb");
    if (file == NULL)
    {
        cout << "Cannot open file";
        fclose(file);
        return;
    }
    unsigned char data[BITMAPFILEHEADERSIZE];
    memset(data, 0, BITMAPFILEHEADERSIZE);
    fread((char*)data, sizeof(unsigned char), BITMAPFILEHEADERSIZE, file);
    unsigned short int fileSig = *(unsigned short int*)&data[0];
    cout << hex << "0x" << fileSig << endl << dec;
    if (fileSig != BITMAPSIGNATURE)
    {
        cout << "Error, you may have loaded a corrupt bitmap image";
        fclose(file);
        return;
    }
    cout << "Everything is good\n";
    fclose(file);
}
```

The main function is pretty trivial, so I will explain what's happening in the BitmapLoader function. We first check if our string filename, contains a bitmap file by checking its extension if it was "bmp", then after that we open the file as read also the "b" just means that the file is a binary file, next we initialize to NULL, by the aid of memset, next we read the file data byte by byte (unsigned char is a byte), never forget that an array is just a pointer in memory, next we dereference the pointer at position 0 and cast it to short int (short int means 2 bytes), this means that we are only getting the first 2 bytes of information, then printing the file signature onto the screen, finally we perform a simple check to see if the file signature matches a normal Bitmap file signature.

2.b DIB Header

The DIB header is a little bit tricky, since Microsoft and other platforms have released many new headers over the past years. Below are some headers along with their sizes:

Size	Header name
12 bytes	BITMAPCOREHEADER OS21XBITMAPHEADER

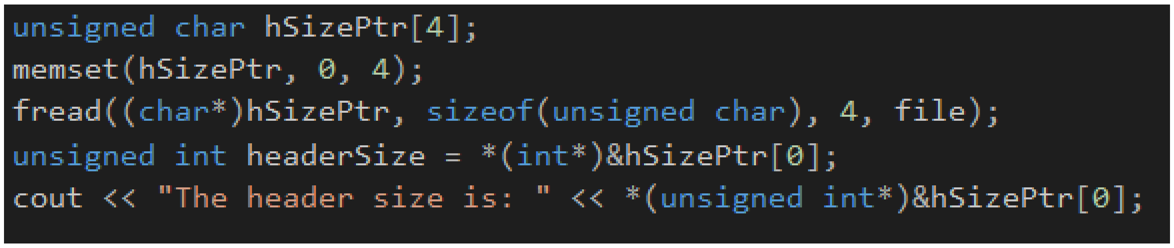
64 bytes	OS22XBITMAPHEADER
16 bytes	OS22XBITMAPHEADER this has the firs 16 bytes the others are assumed to be 0
40 bytes	BITMAPINFOHEADER
52 bytes	BITMAPV2INFOHEADER
56 bytes	BITMAPV3INFOHEADER
108 bytes	BITMAPV4HEADER
124 bytes	BITMAPV5HEADER

Luckily, all other headers just add more fields to the BITMAPCOREHEADER header. The BITMAPCOREHEADER is as follows:

Offset in decimal	**Size**	**OS/2 1.x BITMAPCOREHEADER**
14	4 bytes	Size of the header
18	2 bytes	Image width in pixels (unsigned 16-bit)
20		Image height in pixels (unsigned 16-bit)
22		The number of color planes, must be 1
24	2 bytes	The number of bits per pixel

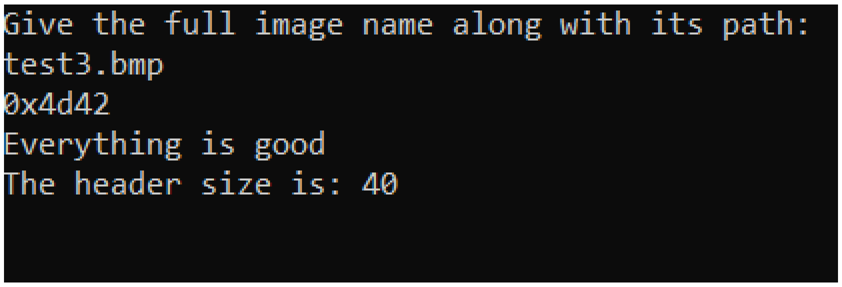
This header is not compressed and it doesn't give support to 16 or 32 bits per pixel images, it only support 1, 4, 8 or 24 according to Microsoft documents. Moreover, other headers have a signed integer for the width and height, instead of unsigned 16-bit, we will talk later about the use of negative height and its representation. Note that this header is no longer supported, many programs are still using it for backward compatibility with older images, however to Microsoft and other corporations it is dead, the oldest version that is still being used to this day is the BITMAPINFOHEADER.

After we have read the Bitmap header that was of size 14 bytes, our file pointer is now pointing to the 14 byte offset, let's read the first 4 bytes and see what's the header type of our test image, this can be done by adding this line of code.



```
unsigned char hSizePtr[4];
memset(hSizePtr, 0, 4);
fread((char*)hSizePtr, sizeof(unsigned char), 4, file);
unsigned int headerSize = *(int*)&hSizePtr[0];
cout << "The header size is: " << *(unsigned int*)&hSizePtr[0];
```

It returned:



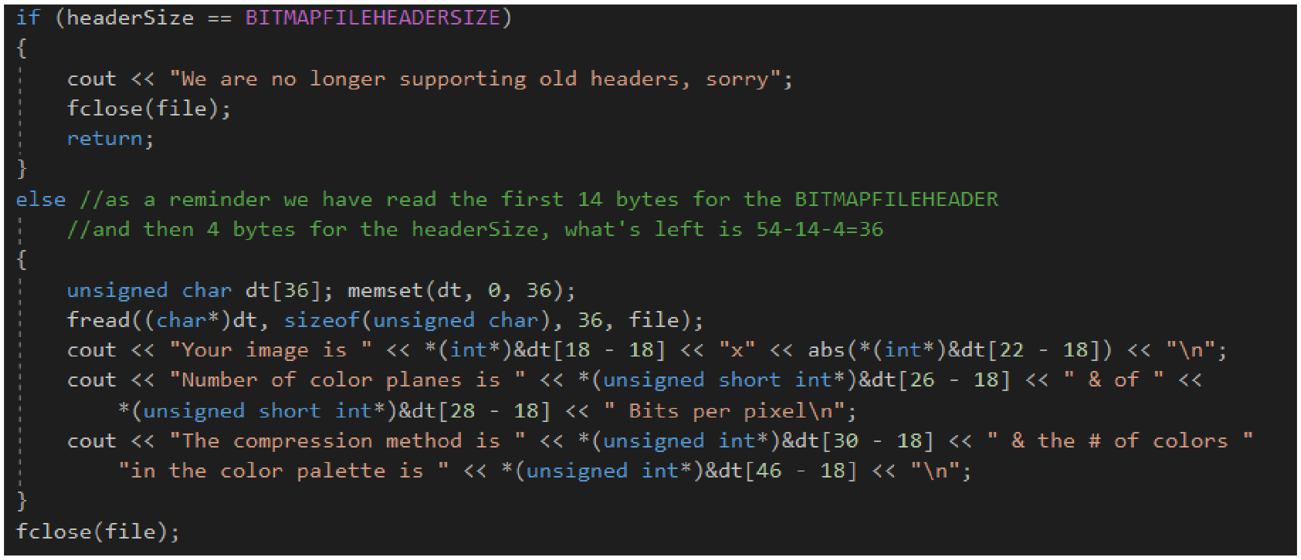
```
Give the full image name along with its path:
test3.bmp
0x4d42
Everything is good
The header size is: 40
```

From the preceding table we can deduce that the DIB header type is BITMAPINFOHEADER.

As the BITMAPINFOHEADER is the most used, we will document this header in the following table:

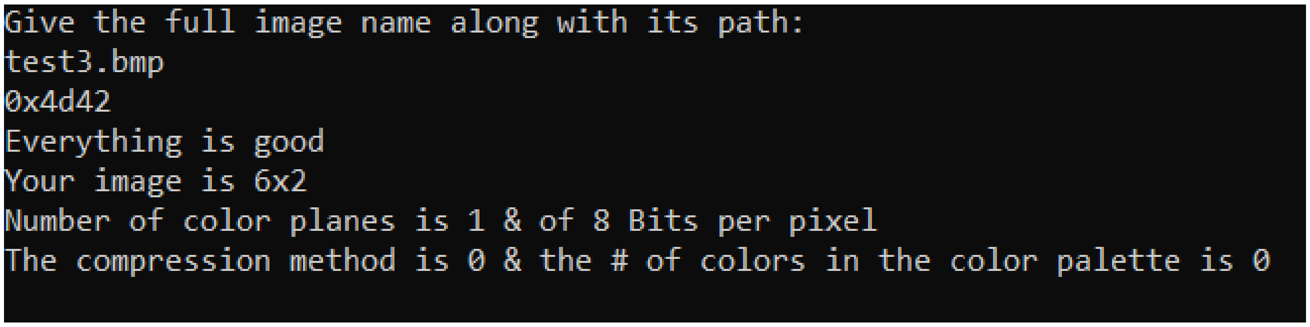
Offset in decimal	Size	BITMAPINFOHEADER
14	4 bytes	The size of the header
18		Image width in pixels, signed int
22		Image height in pixels, signed int
26	2 bytes	Number of color planes (must be 1)
28		The number of bits per pixel
30	4 bytes	Compression method used
34		The image size as raw bitmap data
38		Image horizontal resolution
42		Image vertical resolution
46		Number of colors in the color palette, the value is either 0 or 2^n
50		Number of important colors, if it was 0 then all colors are important

Now let's alter our code so that we could read all of the above information from offset 0 to offset 50 (54 bytes), our function would look something like this:



```cpp
if (headerSize == BITMAPFILEHEADERSIZE)
{
    cout << "We are no longer supporting old headers, sorry";
    fclose(file);
    return;
}
else //as a reminder we have read the first 14 bytes for the BITMAPFILEHEADER
    //and then 4 bytes for the headerSize, what's left is 54-14-4=36
{
    unsigned char dt[36]; memset(dt, 0, 36);
    fread((char*)dt, sizeof(unsigned char), 36, file);
    cout << "Your image is " << *(int*)&dt[18 - 18] << "x" << abs(*(int*)&dt[22 - 18]) << "\n";
    cout << "Number of color planes is " << *(unsigned short int*)&dt[26 - 18] << " & of " <<
        *(unsigned short int*)&dt[28 - 18] << " Bits per pixel\n";
    cout << "The compression method is " << *(unsigned int*)&dt[30 - 18] << " & the # of colors "
        "in the color palette is " << *(unsigned int*)&dt[46 - 18] << "\n";
}
fclose(file);
```

With the following output:



```
Give the full image name along with its path:
test3.bmp
0x4d42
Everything is good
Your image is 6x2
Number of color planes is 1 & of 8 Bits per pixel
The compression method is 0 & the # of colors in the color palette is 0
```

The Compression method specifies how a Bitmap image is compressed, below are the values for the compression method:

Value	Identified By	Compression Method	Comment
0	BI_RGB	Not Compressed	
1	BI_RLE8	Run-length encoding compression (RLE)	Used for bitmaps with 8bpp
2	BI_RLE4		Used for bitmaps with 4bpp
3	BI_BITFIELDS	Not Compressed	The Color table consists of three DWORD color masks that specify the red, green and blue component respectively of each pixel, valid for 16 and 32bpp

4	BI_JPEG		Indicates that the bitmap data is a JPEG
5	BI_PNG		Indicates that the bitmap data is a PNG
6	BI_ALPHABETFIELDS	RGBA bit field masks	Used previously as a back-pass for giving bitmaps alpha, (writing images as RGBA)
11	BI_CYMK	Not compressed	Color space used for printing (Cyan, Magenta, Yellow, Black). Read this
12	BI_ CMYKRLE8	Run-length encoding compression (RLE)	Used with bitmaps with 8bpp, the compression uses a 2-byte format consisting of a count byte followed by a byte containing a color index
13	BI_ CMYKRLE4		Used with bitmaps with 4bpp, the compression uses a 2-byte format consisting of a count byte followed by two word-length color indexes

When the height is positive, it means that the origin is the lower-left corner of the image, bits fill from left to right, from bottom to up, this is referred to as a bottom-up bitmap, if it was negative, then the origin is the upper-left corner of the image, bits fill from left to right, from top to bottom, this is referred to as a top-down bitmap. Note that bottom-up bitmap can be compressed but a top-down bitmap cannot.

2.c Run-length encoding Compression (RLE)

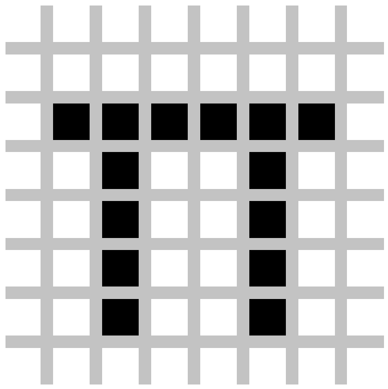
Run-length encoding (RLE for short) is a simple method of compression in which a piece of data with multiple sequences of occurring chunks is stored as a single data value along with a counter. Take as an example the following string:

AAAABCDDDDDVVVVBBBBBBBCDDDDDCBBBBBBB

With RLE we can write the first 4 repeating As as 4A which means A is written 4 times, the whole string can be written as:

4A1B1C5D4V7B1C5D7B

So we were able to write the first string consisting of 36 characters (36 bytes) as a string consisting of 18 characters (18 bytes), this is so efficient considering the fact that no loss of data has occurred. Consider the following 8x8 image, with its raw data shown adjacently:



0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	0
0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0
0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0

By the aid of RLE, we can write the raw data of the image as follows:

```
8 0
8 0
1 0 6 1 1 0
2 0 1 0 2 0 1 0 2 0
2 0 1 0 2 0 1 0 2 0
2 0 1 0 2 0 1 0 2 0
2 0 1 0 2 0 1 0 2 0
8 0
```

While there are more details regarding RLE than found here, yet this is the basic principle of how it works.

We will focus on working with BI_RGB & BI_BITFIELDS.

Moreover, in indexed color bitmaps (1, 2, 4 ,8) we may use the Huffman 1D algorithm. Above 16 bits per pixel image formats, the images are not compressed, however in 16 bits per pixel, however, the 24 bit per pixel image formats can also be compressed with the 24-bit RLE algorithm.

2.d Extra Bit Mask

This is only present under the DIB header on the following condition:

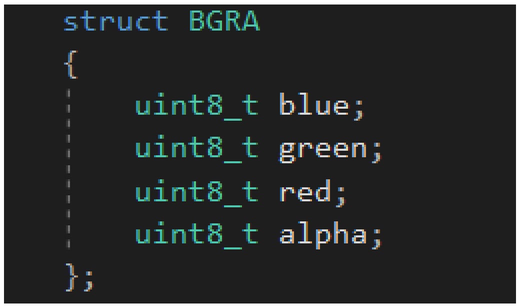
The DIB header is the BITMAPINFOHEADER or earlier version and the Compression method is set to BI_BITFIELDS or BI_ALPHABITFIELDS.

Offset in decimal	Size	Description
54	4	Red channel bit mask
68	4	Green channel bit mask
62	4	Blue channel bit mask
66	4	Alpha channel bit mask

We will furthermore investigate the use of bit mask in the pixel format section.

2.e Color Table

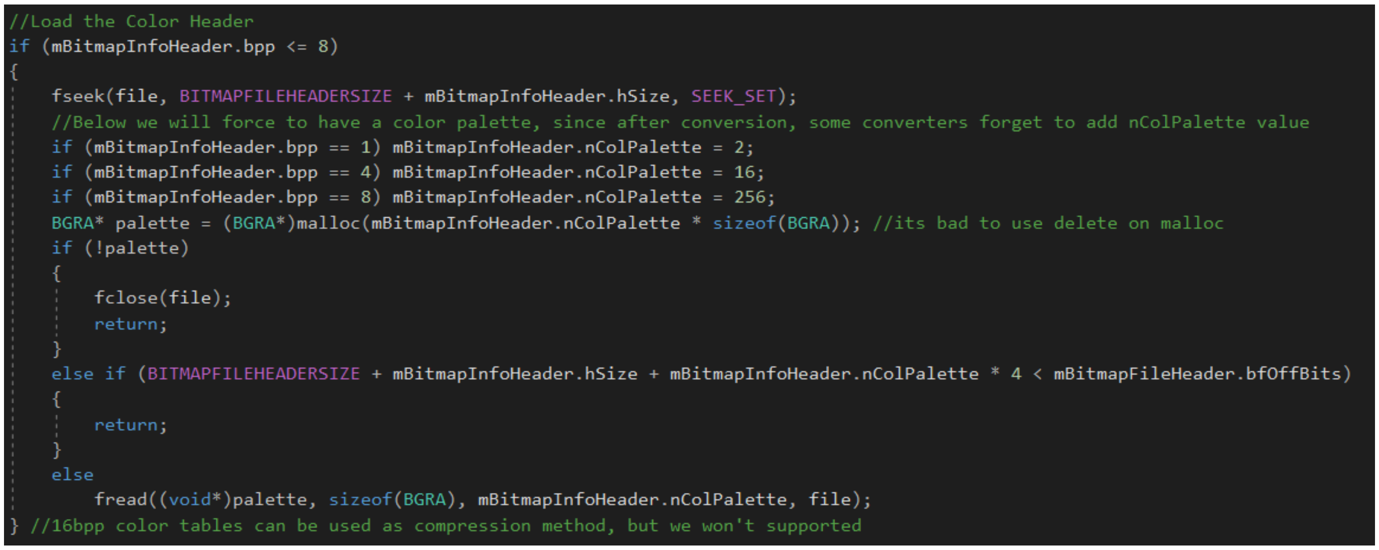
The Color table (or color palette) is found directly after the DIB header (and after the BIT mask if existed) and it's the easiest to understand. The Color table is just an array of BGR structure.



```
struct BGRA
{
    uint8_t blue;
    uint8_t green;
    uint8_t red;
    uint8_t alpha;
};
```

Notice that each variable is defined as uint8_t, this is due to the fact its range is between $0 \rightarrow 255$, and we only need a byte, so we define each as an unsigned char.

The number of elements in the Color table is either 2^n or 0, also in most cases each entry in the Color table occupies 4 bytes as red, green, blue, 0x00. The 0x00 is not used, this was said earlier in the Microsoft Developer Network (msdn) documentation, however over the years Adobe has used as a holder for alpha. The color table is a listing of bytes, each pixel in an indexed color image is described by a number of bits (1, 2, 4, or 8) which is an index to a single color described by this table. The purpose of the color table in indexed color images is to inform the application about the actual color that each of these index values corresponds to. In non-indexed color images, its purpose is to use less memory, thus good in theory for image compression. The color table always exists in images of (1, 2,4 or 8) bits per pixel, it can also be present for 16 bit per pixel images, but it's rare to be present and if it ever existed, then this is due to lower storage use. As a note every color in bitmaps is always stored in the BGRA format.



```
//Load the Color Header
if (mBitmapInfoHeader.bpp <= 8)
{
    fseek(file, BITMAPFILEHEADERSIZE + mBitmapInfoHeader.hSize, SEEK_SET);
    //Below we will force to have a color palette, since after conversion, some converters forget to add nColPalette value
    if (mBitmapInfoHeader.bpp == 1) mBitmapInfoHeader.nColPalette = 2;
    if (mBitmapInfoHeader.bpp == 4) mBitmapInfoHeader.nColPalette = 16;
    if (mBitmapInfoHeader.bpp == 8) mBitmapInfoHeader.nColPalette = 256;
    BGRA* palette = (BGRA*)malloc(mBitmapInfoHeader.nColPalette * sizeof(BGRA)); //its bad to use delete on malloc
    if (!palette)
    {
        fclose(file);
        return;
    }
    else if (BITMAPFILEHEADERSIZE + mBitmapInfoHeader.hSize + mBitmapInfoHeader.nColPalette * 4 < mBitmapFileHeader.bfOffBits)
    {
        return;
    }
    else
        fread((void*)palette, sizeof(BGRA), mBitmapInfoHeader.nColPalette, file);
} //16bpp color tables can be used as compression method, but we won't supported
```

2.f Pixel Storage

Pixels are stored in rows; each row should be a multiple of 4 bytes (32-bit DWORD), for each row if they weren't a multiple of 4 bytes, then padding is added, where padding is just 0x00, each row is of size:

$$RowSize = \left\lceil \frac{BitsPerPixel \times ImageWidth}{32} \right\rceil \times 4 = \left\lfloor \frac{BitsPerPixel \times ImageWidth + 31}{32} \right\rfloor \times 4$$

Above, we use the floor and ceiling function. The array size can be calculated as follows:

$$PixelArraySize = RowSize \times |ImageHeight|$$

The $ImageHeight$ has been passed to the absolute value function since it can be positive or negative.

2.g Pixel Format

The 1-bit per pixel format only supports $2^1 = 2$ distinct colors, each bit represents a pixel, each byte has 8 pixels. Each bit (that is either 0 or 1) found in the pixel array is an index to the Color Table, after loading the color table into a BGR or BGRA pixel array, we can then pass the value (0 or 1) to the array and get the pixel color.

The 2-bit per pixel format only supports $2^2 = 4$ distinct colors, each 2-bits represent a pixel, each byte has 4 pixels. Each 2 bits (that is either 0, 1, 2 or 3) found in the pixel array is an index to the Color Table, after loading the color table into a BGR or BGRA pixel array, we can then pass the value (0,1, 2 or 3) to the array and get the pixel color.

The 4-bit per pixel format only supports $2^4 = 16$ distinct colors, each 4-bits (nibble) represent a pixel, each byte has 2 pixels. Each 4 bits $(0 \rightarrow 15)$ found in the pixel array is an index to the Color Table, after loading the color table into a BGR or BGRA pixel array, we can then pass the value $(0 \rightarrow 15)$ to the array and get the pixel color.

The 8-bit per pixel format only supports $2^8 = 256$ distinct colors, each 8-bits (1 byte) represent a pixel, each byte has 1 pixel. Each 1 byte $(0 \rightarrow 255)$ found in the pixel array is an index to the Color Table, after loading the color table into a BGR or BGRA pixel array, we can then pass the value $(0 \rightarrow 255)$ to the array and get the pixel color.

The 16-bit per pixel format only supports $2^{16} = 65536$ distinct colors, each 2-bytes (1 WORD) represent a pixel. Here we will talk about 3 different types of 16-bit per pixel format.

The first one is the 16 bit per pixel 5:5:5:1 RGB, for the 2-bytes, the first 5 bits represent the blue intensity of the pixel, the second 5 bits represent the green intensity of the pixel, the third 5 bits represent the red intensity of the pixel and the last bit is unused. Each color ranges from 0 to 31 (maximum for a 5-bits to have), we can convert them easily from 16-bpp 555 to RGB by the following equation:

$$Red255 = Red555 \times \frac{255}{31}; Green255 = Green555 \times \frac{255}{31}; Blue255 = Blue555 \times \frac{255}{31}$$

Here comes the mask part, masks are used to get extract the red/green/blue colors from the pixel data, suppose that we have the following pixel:

$$0b1000100111101101$$

To extract the blue component from the pixel, we would have to use an AND gate in the following manner:

$$0b1000100111101101\&0b11111$$

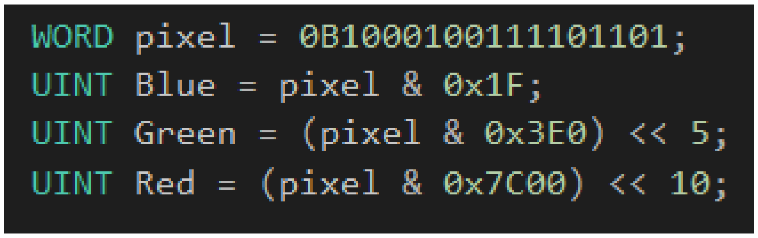
To extract the green component from the pixel, we would have to use an AND gate accompanied with bit shift in the following manner:

$$(0b1000100111101101\&0b1111100000) \ll 5$$

To extract the red component from the pixel, we would have to use an AND gate accompanied with bit shift in the following manner:

$$(0b1000100111101101\&0b111110000000000) \ll 10$$

As such, we can write this as follows:



```
WORD pixel = 0B1000100111101101;
UINT Blue = pixel & 0x1F;
UINT Green = (pixel & 0x3E0) << 5;
UINT Red = (pixel & 0x7C00) << 10;
```

The hexadecimal numbers $0 \times 1F, 0 \times 3E0,0 \times 7C00$ are called the bit masks, these values enable us to extract the pixel color components although for the 16-bpp 555, it doesn't store the masks, that's what I got from this online converter.

The second type is the 16-bpp 555 RGBA, where the last bit represents the alpha level which can be 1 or 0 (since after all we only have 1 bit), the masks are the same for the 16-bpp 555 RGBA, yet we have one extra mask, that is the alpha level, this mask is 0×8000.

As for the third type (I was able to find 3 types only), that is the 16-bpp 565 RGB, similar to the above, we have 6 bits for the green component, this is so that we can increase the accuracy for the green intensity and the explanation for this is due to the fact that our eyes are more sensitive to green light, so it's helpful for us to have the green pixel component to have more accuracy. The masks are as follows in the BGR order: $0 \times 1F, 0 \times 7E0,0 \times F800$, those masks are present the between the DIB Header and the Color Table.

The 24-bit per pixel format only supports $2^{24} = 16777216$ distinct colors, each 3-bytes represent a pixel. Each pixel component is stored in the BGR order and for each byte we have 1 component (i.e. each color component ranges from 0 to 255 since it occupies 1 byte).

The 32-bit per pixel format only supports $2^{32} = 4294967296$ distinct colors, each 4-bytes (DWORD) represent a pixel. Each pixel can define the blue, green, red and alpha components.

Similar to the 16-bpp, the 32-bpp format comes with different flavors. We have the 32-bit RGB, each byte represent a color in the BGR order and we don't have an alpha component (it is left out as 0×00 or any other value), the masks are not stored in the bitmap, yet we have the following masks in the BGR order: $0 \times FF, 0 \times FF00,0 \times FF0000$. The second one (that I was able to find) is the 32-bit RGBA (ARGB32) where the alpha component is used, thus the mask used is $0 \times FF000000$.

3. Summary

From what preceded we are now able to load a bitmap file, save a bitmap file and perform some manipulations, for example color inversion, search for a color in a picture, conversion between different bits per pixel, etc…

4. Reference

https://www.drdobbs.com/architecture-and-design/the-bmp-file-format-part-1/184409517

https://en.wikipedia.org/wiki/BMP_file_format

https://docs.microsoft.com/en-us/windows/win32/gdi/bitmaps

https://web.archive.org/web/20150127132443/https://forums.adobe.com/message/3272950

https://www.herdsoft.com/ti/davincie/imex3j8i.htm

https://www.digicamsoft.com/bmp/bmp.html

https://books.google.com.lb/books?id=-O92IIF1Bj4C&pg=PA591&lpg=PA590#v=onepage&q&f=false Starting from page 590