Bitmap File Format and Manipulation

Application Note AN004

Introduction

There are many different graphics file formats that can be created using various imaging programs, such as .bmp, .jpg, .gif, .tif, etc. However, understanding the internal structure of each image file format is necessary when manipulating and downloading images to OSRAM OLED display.

The objectives of this application note are to explain the fundamental concepts of the internal structure of a 24-bit, uncompressed bitmap file and provide instructions on how to convert a bitmap file from 24-bit to 4-bit that still retains its highest resolution possible along with the C/C++ source code.

Internal Structure of a 24-bit Uncompressed Bitmap File

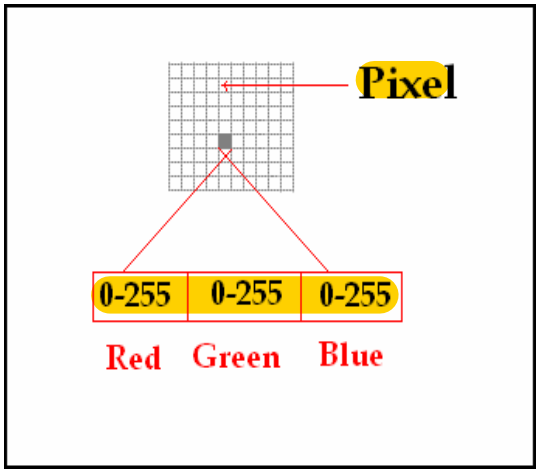

Figure 1. Bitmap Pixel Properties

Windows BMP Bitmap file are characterized by the number of pixels and the color depth per pixel. The color depth of each pixel is always the same in a file. Figure 1 demonstrates the properties of a bitmap pixel, which consists of 24 bits in 3 contiguous bytes: 1 byte (8 bits) for Red, 1 byte for Green, and 1 byte for Blue. When reading bitmap data pixel information, designer has to first read the File Header, the Image Header Information, and then the Bitmap Pixel Data.

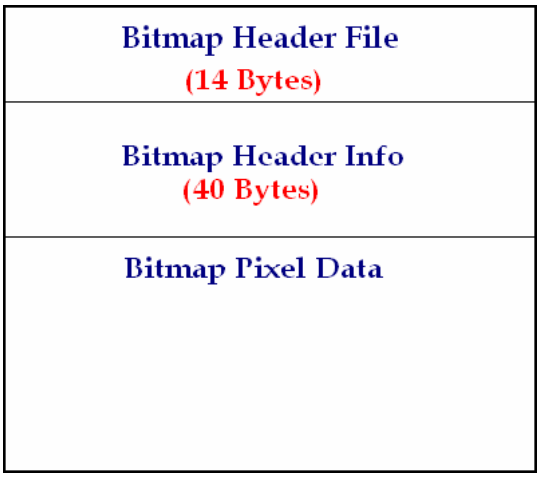

Figure 2. Bitmap File Format

Header File

A Windows Bitmap Header File consists of 14 bytes of information:

bfType: This field occupies **2 bytes** which are allocated for the ASCII characters “BM”.
bfSize: This field occupies **4 bytes** which indicate the size of the Bitmap file.
bfReserved: This field occupies **4 bytes** which are reserved for future use and should be 0.
bfOffBits: This field occupies **4 bytes** which specify the number of bytes from the beginning of the file to the starting Pixel Data byte.

The following code written in C/C++ that can be used to read the Windows Bitmap's Header File:

```
typedef struct BMP1
{
      unsigned short bfType;
      unsigned long  bfSize;
      unsigned short bfReserved1;
      unsigned short bfReserved2;
      unsigned long  bfOffBits;

      BMP1() { bfSize = 0; }
};
```

Header Information

The Header Information of a Windows BMP format occupies at least 40 bytes:

biSize : 4 Bytes – This field indicates the size of the Header Information structure given in bytes
biWidth : 4 Bytes – indicates the width of the bitmap in pixels.
biHeight : 4 Bytes – indicates the height of the bitmap in pixels
biPlanes : 2 Bytes – indicates the number of planes contained in the bitmap and must be set to 1.
biBitCount: 2 Bytes – indicates the bit-depth of the bitmap pixel or bits per pixel.
biCompression: 4 Bytes – indicates whether the bitmap data is Run Length Encoding or not. It should be 0 for uncompressed images
biSizeImage: 4 Bytes – contains the length of the bitmap image data in bytes. This value is not necessary equal to the width multiplied by the height.
biXPelsPerMeter: 4 Bytes – describes the resolution of the bitmap in pixels per meter.
biYPelsPerMeter: 4 Bytes – describes the resolution of the bitmap in pixels per meter.
biClrUsed: 4 Bytes – indicates the number of colors that is used in the bitmap.
biClrImportant: 4 Bytes – signifies the number of significant colors.

```
typedef struct BMP2

{
      unsigned long  biSize;
      unsigned long  biWidth;
      unsigned long  biHeight;
      unsigned short biPlanes;
      unsigned short biBitCount;
      unsigned long  biCompression;
      unsigned long  biSizeImage;
      unsigned long biXPelsPerMeter;
      unsigned long biYPelsPerMeter;
      unsigned long  biClrUsed;
      unsigned long  biClrImportant;

      BMP2()
       {
            biWidth
            = biHeight
            = biCompression
            =  biSizeImage
            = biXPelsPerMeter
            = biYPelsPerMeter
            = biClrUsed
            = biClrImportant = 0;
            biSize = sizeof(BMP2);
            biPlanes = biBitCount
            = 0;
      }
};
```

ReadBitmapFileHeader() and **ReadBitmapInfoHeader()** functions are to read the Image Header and Image Information in C/C++.

```
void ReadBmpFileHeader( BITMAPFILEHEADER *FILEHEADER, FILE *fp )
{
        fread( &FILEHEADER->bfType,      sizeof( FILEHEADER->bfType ),          1, fp );
        fread( &FILEHEADER->bfSize,      sizeof( FILEHEADER->bfSize ),          1, fp );
        fread( &FILEHEADER->bfReserved1, sizeof( FILEHEADER->bfReserved1 ),     1, fp );
        fread( &FILEHEADER->bfReserved2, sizeof( FILEHEADER->bfReserved2 ),     1, fp );
        fread( &FILEHEADER->bfOffBits,   sizeof( FILEHEADER->bfOffBits ),       1, fp );

}
void ReadBmpInfoHeader( BITMAPINFOHEADER *INFOHEADER, FILE *fp )
{
        fread( &INFOHEADER->biSize,      sizeof( INFOHEADER->biSize ),    1, fp );
        fread( &INFOHEADER->biWidth,     sizeof( INFOHEADER->biWidth ),    1, fp );
        fread( &INFOHEADER->biHeight,    sizeof( INFOHEADER->biHeight ),  1, fp );
        fread( &INFOHEADER->biPlanes,    sizeof( INFOHEADER->biPlanes ),  1, fp );
        fread( &INFOHEADER->biBitCount,  sizeof( INFOHEADER->biBitCount ),       1, fp );
        fread( &INFOHEADER->biCompression,  sizeof( INFOHEADER->biCompression ),1,fp );
        fread( &INFOHEADER->biSizeImage,        sizeof( INFOHEADER->biSizeImage ),
        1, fp );
        fread( &INFOHEADER->biXPelsPerMeter,    sizeof( INFOHEADER->biXPelsPerMeter ),
        1, fp );
        fread( &INFOHEADER->biYPelsPerMeter,    sizeof( INFOHEADER->biYPelsPerMeter ),
        1, fp );
        fread( &INFOHEADER->biClrUsed,          sizeof( INFOHEADER->biClrUsed ), 1, fp );
        fread( &INFOHEADER->biClrImportant,sizeof( INFOHEADER->biClrImportant ),1, fp );
}
```

Pixel Data

Bitmap pixel data is stored upside down from its actual image as demonstrated in Figure 3 below. Therefore, when reading or writing an image to a file, data starts at the bottom left corner and goes from left to right, bottom to top in pixel unit. Since the image is stored in inverted order, each RGB values are written in backward starting with blue, green, and then red.

One important issue that we have to pay careful attention to when manipulating bitmap image file is how to manage junk bytes. The red square below is used as an example to demonstrate the junk byte concept.

When saving this red square as a bitmap image file to any local directory in the computer, the file is 15 by 15 pixels, 24-bit

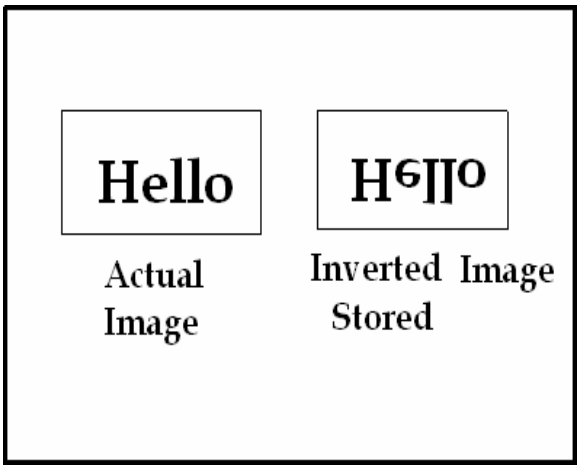

Figure 3. Image Orientation

uncompressed bitmap with its total size on disk of 774 bytes. As mentioned earlier, each 24-bit pixel consists of 3 bytes: 1 byte for red, one byte green, and 1 byte for blue plus 14 bytes for Header file and 40 bytes for Header information, the total of (15x15x3) + 14 + 40 = 729 bytes.

Compared to its actual size on disk, there are 45 bytes extra.

The reason for junk bytes is that the number of bytes per row of a bitmap file is grouped into 4-byte block. Therefore, it has to be divisible by 4. When the amount of bytes per row in not divisible by 4, then the junk bytes are needed to add in at the end of the array of bytes to make it divisible by four. Therefore, there could be no junk byte, one "00", two "00 00", or three "00 00 00" junk bytes per row.

Refer to the red square above again; there are 15 pixels per row. Therefore, each row has 3 junk bytes because 15%4 = 3. There are 15 rows. That is why there are 45 bytes extra on disk comparing to its actual size (15 * 3 = 45).

Note that % is the modulo operator. It equals to the remainder of the division. For example, 7 / 3 = 2 with 1 remainder. In other words, 7 % 3 = 1.

Let's use another example to illustrate the concept: we have a bitmap file of 5 columns and 4 rows. There are 3 bytes of RGB values for each pixel which are arbitrarily chosen below.

FF00FF FF00FF 00FFFF 00FF00 FFFF00

Pixel per column * 3 bytes = Total number of bytes per column
5 columns * 3 = 15 bytes

If we arrange these 15 bytes into 4-byte block, we have:

FF00FFFF 00FF00FF FF00FF00 FFFF00

Notice that the last group has only 3 bytes and is not evenly divisible by 4. Therefore, one junk byte is needed to pad in to make it divisible by 4.

FF00FFFF 00FF00FF FF00FF00 FFFF0000

The following formula can be used to calculate junk bytes:

4 – ((pixel per row * 3) % 4) = junk bytes

*4 – ((5 * 3) % 4) = 4 – (15 % 4) = 4 – 3 = 1 junk byte*

```
fseek(fp, (BMPFILEHEADER.bfOffBits), SEEK_SET);

if (BMPINFOHEADER.biSizeImage ==0)
      BMPINFOHEADER.biSizeImage = BMPFILEHEADER.bfSize;

int bm_width = (WORD)BMPINFOHEADER.biWidth;
int bm_height = (WORD)BMPINFOHEADER.biHeight;

if ((BMPINFOHEADER.biSizeImage - bm_width * bm_height * 3) == 0)
       EndByte = 0;
else EndByte = (BMPINFOHEADER.biSizeImage - bm_width * bm_height * 3) /
                bm_height;
```

EndByte in the block of C/C++ code above is a variable used to hold the number of junk bytes of the image file. Sometimes, the File Header and File Information are longer than 54 bytes, the following line of code is to set the pointer to the first data pixel of the bitmap file before reading its first RGB value of the image.

```
fseek(fp, (BMPFILEHEADER.bfOffBits), SEEK_SET);
```

Converting 24-bit Pixels into 4-bit Pixels

Since OSRAM OLED display is 4-bit, we have to convert 24-bit pixels into 4-bit pixels before writing an image to the display.

A 24-bit pixel is formatted as the following:

Red (8 bits)	Green (8 bits)	Blue (8 bits)
11011011	11110110	01111110

RGB values for each pixel can be any value.
(Red + Green + Blue) / 3 = 8-bit pixel. Therefore, (11011011 + 11110110 + 01111110) / 3 = 11000101. Since these are still 8 bits long, we have to eliminate the last 4 bits to make it a 4-bit pixel. Therefore, the new value is 1100.

ReadBmpImage() function below reads the entire bitmap file starting the xStart, yStart to xEnd, yEnd, which specifies the image window size in pixel units. After this function being executed, GrayVal should equal to the average value of pixel data Blue, Red, and Green. DisplayDataRam is used as an internal memory buffer to store the new image of 8 bits per pixel.

```
void ReadBmpImage( FILE *fp, const USHORT xStart, const USHORT xEnd, const
USHORT yStart, const USHORT yEnd )
{
      for( USHORT y=yStart; y < yEnd; y++ )
      {
            for( USHORT x=xStart; x < xEnd; x++ )
            {
                  Blue = getc(fp);
                  Green = getc(fp);
                  Red = getc(fp);
                  GrayVal = (Blue + Green + Red)/3;
                  DisplayRamData[y][x] = GrayVal;
            }
      }
}
```

After formatting the new bitmap image into 8 bits per pixel, the lower 4 bits will be truncated as shown in function **Combine2PixInto1Byte()** as shown below:

```
void Combine2PixInto1Byte (bool debug_mode)
{
	union
	{
		unsigned short int byte;
		struct
		{
			unsigned nib0:4;
			unsigned nib1:4;
		}Nib;
	} OddVal, EvenVal;

	int FlipY = YMax - 1;
	for (unsigned int E = YMin;   E < YMax;   E++)
	{
		for (unsigned int i = XMin; i < XMax;   i = i + 2)
		{

			EvenVal.byte  = DisplayRamData[FlipY][i];
			OddVal.byte   = DisplayRamData[FlipY][i + 1];
			EvenVal.Nib.nib0 = OddVal.Nib.nib1;
			WrDrBuf(EvenVal.byte, debug_mode);
		}
		if (FlipY >= YMin)
			FlipY--;
	}
}
```

Author: Paulina T. Nguyen
OLED Display Engineering
San Jose, USA

About Osram Opto Semiconductors

Osram Opto Semiconductors GmbH, Regensburg, is a wholly owned subsidiary of Osram GmbH, one of the world's three largest lamp manufacturers, and offers its customers a range of solutions based on semiconductor technology for lighting, sensor and visualisation applications. The company operates facilities in Regensburg (Germany), San José (USA) and Penang (Malaysia). Further information is available at www.osram-os.com.