

The BMP File Format, Part 1

In the first installment of this two-part article, David examines how the "standard" BMP file format has been implemented differently on different platforms.

March 01, 1995

URL: <http://drdobbs.com/architecture-and-design/the-bmp-file-format-part-1/184409517>

Once only for Windows, it's now a common format for other platforms

David is a software engineer with Visix Software, makers of the Galaxy Application Framework. He can be contacted via the Internet at david@visix.com.

The standard format for storing bitmap images in Windows applications is the bitmap (BMP) file format. Sometimes known as device-independent bitmap (DIB), this format is also used by OS/2 programs and many DOS applications. Because of its popularity with Windows applications, however, it is becoming a common format for software on other operating systems as well. Unfortunately, many different file formats claim to be BMP files, some of which are incorrectly structured due to documentation errors in some early editions of the Windows SDK.

Thus, it is common to find BMP files that an otherwise well-written application cannot read. In this two-part article, I'll examine all of the different ways a BMP file can be put together, so that an application can be built that reads all known BMP file formats, plus future formats. The code I'll present assumes nothing about the system it runs on so it should work equally well on DOS, OS/2, UNIX, VMS, or any other operating system.

Throughout this article, I'll use data types such as INT16 and UINT32 instead of the usual *short*, *long*, and *int* because the standard C data types do not have specific sizes associated with them. [Table 1](#) lists these types. In an attempt to keep everything portable, I am not using structures from either the Windows SDK or the OS/2 toolkit. Instead, I have a set of structures which are supersets of both kits. This way, applications using bitmap files (DOS or UNIX programs, for example) can be compiled and run without either toolkit. [Table 2](#) lists my structures and the corresponding Windows and OS/2 structures. Note that the BITMAPARRAYHEADER structure does not exist in the Windows SDK.

In this first installment, I examine the four structures listed in [Table 2](#). Next month, I'll focus on reading and interpreting bits, and discuss how the structures fit together.

Basic Concepts of BMP

Unlike other image-file formats like GIF (CompuServe's Graphic Interchange File format) and JPEG (Joint Photographic Experts Group), the BMP file format was not designed to be portable. Instead, it was designed to easily work with the Windows API using the same structures that Windows applications use to manipulate in-memory bitmaps. As the API changed, so did the BMP file format. Windows now has three documented variants on the BMP file format: one each for Windows 2.x, Windows 3.x, and Windows NT. Additionally, OS/2 has variants designed to work easily with the OS/2 API: one each for OS/2 1.1, OS/2 1.2, and OS/2 2.x, plus one to store its icon (ICO) and pointer (PTR) files.

Fortunately, only two sets of structures are nearly identical. With a little work, you can produce a single set of structures to describe all bitmap files. [Figure 1](#) illustrates the BITMAPFILEHEADER structure; [Figure 2](#), the BITMAPARRAYHEADER structure; [Figure 3](#), BITMAPHEADER; and [Figure 4](#), the RGB structure.

All BMP formats, including the OS/2 icon and pointer variants, are built using these four structures. All use the BITMAPFILEHEADER structure, various subsets of the BITMAPHEADER structure (the *size* field indicating how much is being used), and the RGB structure. OS/2-format bitmap files containing more than one image also use the BITMAPARRAYHEADER structure.

There is one caveat: The oldest forms of the bitmap file format use 16- instead of 32-bit integers for the *width* and *height* fields of the BITMAPHEADER structure. Fortunately, it is easy to tell when such a file is being read, since the *size* field is always 12 for those structures and greater than 12 for the newer structures. Additionally, files based on the "new" format will add an extra byte to the end of every RGB structure, in order to align it on a 4-byte boundary.

Byte Ordering

The BMP format was designed to work with systems based on Intel processors with their Little-endian byte-ordering scheme. This means that whenever a multibyte field is read (such as a 16- or 32-bit integer), the first byte read will be the least significant, and the last, the most significant. While this isn't an issue when reading the files on Intel processors, it is critical on systems like Sun workstations that use Big-endian byte ordering.

To get around these problems, it is a good idea to use functions that are not dependent on byte order when reading and writing fields; see [endian.c](#) ([Listing One](#)) and [endian.h](#) ([Listing Two](#)). These functions are used throughout the sample code. They work properly without any conditional compilation statements. They read the bytes individually, combining them arithmetically to produce the required multibyte integer. The resulting integer will be correctly formatted for the host processor. Similarly, writing is performed by decomposing the integer arithmetically and writing the bytes individually, least-significant byte first.

These functions also help work around problems arising from differences between 16/32/64-bit processors. On a 64-bit processor (such as an Alpha-based system), the size of a short integer will be 32 bits instead of the usual 16. Reading and writing the files on a byte-by-byte basis avoids problems that can arise from these differences.

Reading the Structures

Each of the structures, `BITMAPFILEHEADER`, `BITMAPARRAYHEADER`, `BITMAPHEADER`, and `RGB`, is involved in reading a bitmap file. Each structure may be read by calling the appropriate Little-endian-value *read* function on each field. The fields are ordered as they appear in the structure definition.

That's all that's required to read `BITMAPFILEHEADER` and `BITMAPARRAYHEADER` structures. Both structures are of a fixed length, so reading them simply involves reading each field.

The `BITMAPHEADER` and `RGB` structures, however, are more complicated. Both are variable-length structures, and therefore require special handling when reading. The `BITMAPHEADER` structure contains its length as the first field. The `RGB` structure is often stored with an extra byte of padding when used in color tables.

The *readBitmapFileHeader*, *readBitmapArrayHeader*, *readBitmapHeader*, and *readRgb* functions in `readbmp.h` and `readbmp.c` show how to properly read these structures. (These files are available electronically; see "Availability," page 3.)

Reading the BITMAPHEADER Structure

The `BITMAPHEADER` structure contains all the information required to describe a bitmap (see [Figure 3](#)). It is a 64-byte long structure containing many fields. But most bitmap files do not use all of the fields, so how do you know how many are used and how many are not?

The answer is the *size* field, which always contains the length (in bytes) of the `BITMAPHEADER` structure. If only the first 16 bytes are used, then only they are stored in the file, and the *size* field will contain the value 16. If all 64 bytes are used, then all 64 bytes are stored in the file and the *size* field will contain the value 64.

The smallest value for the *size* field is 12 bytes. This is a special case--only the "old" format bitmap files will have a `BITMAPHEADER` structure 12 bytes long. When reading these old format bitmaps, the *width* and *height* fields must be read as 16-bit integers instead of the usual 32-bit integers.

When reading the `BITMAPHEADER` structure, you must keep count of the number of bytes read, since the structure can be any length (even greater than 64). After reading the number of bytes indicated by the *size* field, the structure has been completely read. Any fields in the structure that have not been read should be initialized to 0. If the entire structure has been read and the count is still less than the value contained in the *size* field, then there are extra bytes on the disk, since the meaning of these extra bytes is unknown.

Color Tables

Immediately following every `BITMAPHEADER` structure is a color table. For most bitmaps, it is an array of `RGB` structures. When reading the pixel values of such a bitmap, each pixel's value will be an index into this array indicating the color that value represents. This is not the case only when `BITFIELDS` compression is used; then, the table has special meaning.

For bitmaps with normal color tables (where the compression scheme is not `BITFIELDS` encoding), the length of the color table is a function of the number of colors the image can have. The number of possible colors is $2^{\text{bit depth}}$, where *bit depth* is the number of color planes, multiplied by the number of bits per plane. If bit depth is 24, however, the length of the color table is 0, since each pixel will already contain the proper RGB value.

Each entry in a normal color table is one `RGB` structure. The first byte is the blue value; the second, green; and the third, red. If this is a "new" format bitmap (indicated by the image's `BITMAPHEADER` structure having a size greater than 12 bytes), then an additional byte of padding between each `RGB` structure must be skipped over before reading the next structure in the array.

The *readColorTable* function (see `readbmp.h` and `readbmp.c`, available electronically) shows how to properly read a normal color table.

Understanding the BITMAPFILEHEADER Structure

The `BITMAPFILEHEADER` structure ([Figure 1](#)) always appears on disk immediately before a `BITMAPHEADER`. It indicates what type of image the `BITMAPHEADER` describes, points to the bitmap's pixel data, and (in the case of OS/2 pointer files) indicates where the pointer's hot spot is. For a single-image file, the first structure in the file will be a `BITMAPFILEHEADER`. For a multiple-image file, the first structure will be a `BITMAPARRAYHEADER`, and each image will have its own `BITMAPFILEHEADER`.

The first field, *type*, indicates what type of image the following `BITMAPHEADER` describes. [Table 3](#) contains the field's possible values. Note how each numeric value can also be interpreted as two characters, which are a mnemonic describing the image type.

The next field of the `BITMAPFILEHEADER`, *size*, can be safely ignored. It is a count of bytes used for the `BITMAPFILEHEADER` combined with the subsequent `BITMAPHEADER`. Using this field for anything is dangerous, though. Windows and OS/2 have different definitions for the field. Windows defines it as the size of the entire bitmap file (in `DWORD`s), while OS/2 defines it as the size of the `BITMAPFILEHEADER`, plus the size of the following `BITMAPHEADER` structure (in bytes). Many bitmap files simply store 0 in this field. Fortunately, the `BITMAPHEADER` contains its own *size* field, so this one can be ignored.

The next two fields are the *x* and *y* coordinates for the hot spot. For `TYPE_BMP` images, this value is ignored (and those two fields are documented as "reserved" in the Windows SDK, where this structure is only used for BMP files). For the other four image types, these two values contain the position where the hot spot should be if the image is used as a mouse pointer. The coordinate is relative to the lower-left corner of the image.

The fifth and last field, *offsetToBits*, contains the byte offset (from the start of the file) to the first byte of the image's color data. The interpretation of this data varies depending on the contents of the BITMAPHEADER that follows.

Understanding the BITMAPARRAYHEADER Structure

The BITMAPARRAYHEADER structure is used to establish a linked list of images in a multiple-image bitmap file. Because only OS/2 supports the multiple-image bitmap file, Windows applications often crash when a bitmap file using this structure is loaded. A BITMAPFILEHEADER always follows immediately after every BITMAPARRAYHEADER structure.

The first field of a BITMAPARRAYHEADER structure is the *type* field and always contains the hexadecimal value 0x4142 ('BA'). This value has the symbolic name TYPE_ARRAY.

Although it seems redundant with only one possible value, the *type* field serves an important purpose. Since a file can begin with either a BITMAPARRAYHEADER or a BITMAPFILEHEADER structure, the *type* field, which appears first in both structures, can be used to determine which of the two structures the first structure is.

The second field, *size*, serves a similar purpose to the *size* field in the BITMAPFILEHEADER. In other words, it can be safely ignored.

The third field, *next*, is a pointer to the next BITMAPARRAYHEADER in the linked list. Its value is the byte offset from the start of the file where the next BITMAPARRAYHEADER begins. It will contain 0 if this is the last image in the list.

The last two fields, *screenWidth* and *screenHeight*, define the device on which this image should be displayed. If they contain 0s, then the image is device independent; otherwise, it is intended for a screen with a height and width (in pixels) that matches these fields. OS/2 multiple-image bitmaps usually contain the same picture, rendered with different dimensions and/or colors. When using a multiple-image file, proper procedure is to pick the image whose *screenWidth* and *screenHeight* fields match the output device's size. If there are no matches, the first image in the file is normally used. Device-independent images (with *screenWidth* and *screenHeight* values of 0) should always be the first images in a file.

Understanding the BITMAPHEADER Structure

The BITMAPHEADER structure is at the core of every image in a bitmap file. Although it contains 19 fields, many of them can be safely ignored. As a matter of fact, most images will contain 0s (default values) for all but the first five fields.

The first field, *size*, is used to determine the size of the structure on disk. The next two fields, *width* and *height*, indicate the dimensions (in pixels) of the image. It should be noted that height can be negative, in which case it uses an inverted coordinate system (an upper-left origin instead of the usual lower-left). In this case, the actual height of the image is the absolute value of the *height* field.

The next two fields, *numBitPlanes* and *numBitsPerPlane*, indicate color depth. *numBitPlanes* is nearly always 1--Windows only supports single-plane bitmaps, and all of OS/2's standard bitmap formats are single plane. *numBitsPerPlane* is the bit depth for a single-bit plane. Multiplying these two values together yields the image's bit depth, usually 1, 4, 8, or 24. Windows 3.x supports bit depths of only 1, 4, and 8. Windows NT supports 1, 4, 8, 16, 24, and 32. OS/2's standard bit depths are 1, 4, 8, and 24, although any depth from 1 through 24 is theoretically valid.

The next field, *compressionScheme*, indicates whether the image's pixel data has been compressed. Most bitmaps are uncompressed, but some use RLE compression: [Table 4](#) lists the possible values. Note that the value 3 indicates both modified Huffman compression and BITFIELDS encoding. OS/2 interprets the value as modified Huffman compression, while NT reserves it for BITFIELDS. This is not a problem, however, since the BITFIELDS layout is only used for 16- and 32-bit-per-pixel images and modified Huffman encoding is only used for monochrome (1-bit-per-pixel) bitmaps.

The *sizeOfImageData* field is the number of bytes an image's pixel data consumes. The start of the data is pointed to by the *offsetToBits* field in the BITMAPFILEHEADER structure. This field is normally 0 if the *compressionScheme* is COMPRESSION_NONE, in which case, the size is calculated from the width, height, and bit depth of the image.

For most files, the rest of the structure will normally contain 0s. The following fields can be safely ignored without distorting the images:

- *xResolution* and *yResolution*, which contain the resolution (in pixels per meter) of the image. If these values are nonzero, they can be used to generate a scaling factor to print the image at the proper size.
- *numColorsUsed*, which indicates the number of colors in the color table actually used by the image. *numImportantColors* indicates the number of colors required for proper rendering of the image. For both fields, 0s mean "all of them." If *numColorsUsed* is nonzero, the color table is only that long--additional entries are undefined and may not exist in the file.
- *resolutionUnits*, which indicates what units are used for *xResolution* and *yResolution*. This field always contains 0, meaning pixels-per-meter and is probably a place holder for future expansion of the bitmap file format.
- *padding*, which is unused space, and serves only to align the remaining data onto a 4-byte boundary.
- *origin*, which indicates the direction in which the bits fill in the bitmap. It always contains 0, meaning the origin is the lower-left corner. Bits fill in from left-to-right and bottom-to-top. This field, however, is not the only thing that determines the origin of the image. Windows bitmaps do not use this field (the Windows structures stop after *resolutionUnits*). Windows bitmaps may be stored with an upper-left origin, indicated by storing a negative value for the image's height.
- *halftoning*, a flag that indicates one of four halftoning algorithms. The use of these algorithms is undocumented, so I can not provide any additional information. *halftoningParam1* and *halftoningParam2* are parameters used by the halftoning algorithms; [Table 5](#) lists the possible values.
- *colorEncoding*, which describes the format of each entry of the color table. It is always 0, indicating RGB encoding.
- *identifier*, the last field in the BITMAPHEADER structure, is not used in describing bitmaps. It contains a 32-bit value for application use.

Understanding the RGB Structure

The RGB structure is rather simple compared to the other structures. It contains three byte values--red, green, and blue--used to describe a single color value. Color tables are an array of these structures.

Until Next Month

That's it for now. In next month's installment, I'll focus on reading and interpreting bits, and discuss how the structures fit together.

Figure 1: The BITMAPFILEHEADER structure.

```
typedef struct BITMAPFILEHEADER
{
    UINT16    type;
    UINT32    size;
    INT16     xHotspot;
    INT16     yHotspot;
    UINT32    offsetToBits;
} BITMAPFILEHEADER;
```

Figure 2: BITMAPARRAYHEADER structure.

```
typedef struct BITMAPARRAYHEADER
{
    UINT16    type;
    UINT32    size;
    UINT32    next;
    UINT16    screenWidth;
    UINT16    screenHeight;
} BITMAPARRAYHEADER;
```

Figure 3: BITMAPHEADER structure.

```
typedef struct BITMAPHEADER
{
    UINT32 size;
    INT32  width;
    INT32  height;
    UINT16 numBitPlanes;
    UINT16 numBitsPerPlane;
    UINT32 compressionScheme;
    UINT32 sizeOfImageData;
    UINT32 xResolution;
    UINT32 yResolution;
    UINT32 numColorsUsed;
    UINT32 numImportantColors;
    UINT16 resolutionUnits;
    UINT16 padding;
    UINT16 origin;
    UINT16 halftoning;
    UINT32 halftoningParam1;
    UINT32 halftoningParam2;
    UINT32 colorEncoding;
    UINT32 identifier;
} BITMAPHEADER;
```

Figure 4: RGB structure.

```
typedef struct RGB
{
    UINT8 blue;
    UINT8 green;
    UINT8 red;
} RGB;
```

Table 1: Basic data types.

Type	Description
INT8	Integer at least 8 bits wide
INT16	Integer at least 16 bits wide
INT32	Integer at least 32 bits wide
UINT8	Unsigned INT8
UINT16	Unsigned INT16
UINT32	Unsigned INT32

Table 2: Structures used in bitmap files.

My Structure	Windows Structures	OS/2 Structures
BITMAPFILEHEADER	BITMAPFILEHEADER	BITMAPFILEHEADER, BITMAPFILEHEADER2
BITMAPARRAYHEADER	(N/A)	BITMAPARRAYFILEHEADER, BITMAPARRAYFILEHEADER2
BITMAPHEADER	BITMAPCOREINFOHEADER BITMAPINFOHEADER	BITMAPINFOHEADER BITMAPINFOHEADER2
RGB	RGBTRIPLE, RGBQUAD	RGB, RGB2

Table 3: Valid BITMAPHEADER types.

Value	Symbolic Name	Description
0x4D42 ('BM')	TYPE_BMP	Bitmap
0x4349 ('IC')	TYPE_ICO	OS/2 Icon
0x4943 ('CI')	TYPE_ICO_COLOR	OS/2 Color Icon
0x5450 ('PT')	TYPE_PTR	OS/2 Pointer
0x5043 ('CP')	TYPE_PTR_COLOR	OS/2 Color Pointer

Table 4: Compression schemes.

Value	Symbolic name	Description
0	COMPRESSION_NONE	No compression
1	COMPRESSION_RLE_8	8-bit-per-pixel RLE compression
2	COMPRESSION_RLE_4	4-bit-per-pixel RLE compression
3	COMPRESSION_HUFFMAN1D	1-bit-per-pixel modified Huffman compression
3	COMPRESSION_BITFIELDS	16- and 32-bit-per-pixel BITFIELDS encoding
4	COMPRESSION_RLE_24	24-bit-per-pixel RLE compression

Table 5: Halftoning algorithms.

Value	Symbolic name	Description
0	HALFTONING_NONE	No halftoning
1	HALFTONING_ERROR_DIFFUSION	Error-diffusion halftoning
2	HALFTONING_PANDA	Processing Algorithm for Noncoded Document Acquisition (PANDA)
3	HALFTONING_SUPER_CIRCLE	Super-circle halftoning

Listing One

```

/* These functions read and write our basic integer types from a little-endian
 * file. The endian and word-size of the host machine will not affect this
 * code. The only assumption made is that the C data type (char) is one byte
 * long. This should be a safe assumption.
 */

#include <stdio.h>
#include "bmptypes.h"
#include "endian.h"

/*****
 * Read functions. All read functions take an open file pointer as the first
 * parameter and a pointer to data as the second parameter. The return value
 * will be 0 on success, and EOF on failure. If successful, the second
 * parameter will point to the data read.
 */

/* The INT8 and UINT8 types are stored as a single byte on disk. The INT8
 * type is a signed integer with range (-128..127). The UINT8 type is an
 * unsigned integer with range (0..255).
 */
int readINT8little(FILE *f, INT8 *i)
{
    int rc;
    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    *i = (rc & 0xff);
    return 0;
}
int readUINT8little(FILE *f, UINT8 *i)
{
    int rc;
    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    *i = (rc & 0xff);
    return 0;
}

/* The INT16 and UINT16 types are stored as two bytes on disk. The INT16 type
 * is a signed integer with range (-32768..32767). The UINT16 type is an
 * unsigned integer with range (0..65535).
 */
int readINT16little(FILE *f, INT16 *i)
{
    int rc;
    INT16 temp = 0;

    temp = (fgetc(f) & 0xff);

    rc = fgetc(f);
    if (rc == EOF)

```

```

        return rc;
    temp |= ((rc & 0xff) << 8);
    *i = temp;
    return 0;
}
int readUINT16little(FILE *f, UINT16 *i)
{
    int rc;
    UINT16 temp = 0;

    temp = (fgetc(f) & 0xff);

    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    temp |= ((rc & 0xff) << 8);
    *i = temp;
    return 0;
}
/* The INT32 and UINT32 types are stored as four bytes on disk. The INT32
 * type is a signed integer with range (-2147483648..2147483647). The UINT32
 * type is an unsigned integer with range (0..4294967295).
 */
int readINT32little(FILE *f, INT32 *i)
{
    int rc;
    INT32 temp = 0;

    temp = ((long)fgetc(f) & 0xff);
    temp |= (((long)fgetc(f) & 0xff) << 8);
    temp |= (((long)fgetc(f) & 0xff) << 16);

    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    temp |= (((long)rc & 0xff) << 24);
    *i = temp;
    return 0;
}
int readUINT32little(FILE *f, UINT32 *i)
{
    int rc;
    UINT32 temp = 0;

    temp = ((long)fgetc(f) & 0xff);
    temp |= (((long)fgetc(f) & 0xff) << 8);
    temp |= (((long)fgetc(f) & 0xff) << 16);

    rc = fgetc(f);
    if (rc == EOF)
        return rc;
    temp |= (((long)rc & 0xff) << 24);
    *i = temp;
    return 0;
}
/*****
 * Write functions. All write functions take an open file pointer as the first
 * parameter and a data as the second parameter. The return value will be 0 on
 * success, and EOF on failure. If successful, the second parameter will have
 * been written to the open file.
 */
int writeINT8little(FILE *f, INT8 i)
{
    return fputc(i, f);
}
int writeUINT8little(FILE *f, UINT8 i)
{
    return fputc(i, f);
}
int writeINT16little(FILE *f, INT16 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 8) & 0xff), f);
}
int writeUINT16little(FILE *f, UINT16 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 8) & 0xff), f);
}
int writeINT32little(FILE *f, INT32 i)
{
    int rc;

```

```

    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 8) & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 16) & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 24) & 0xff), f);
}
int writeUINT32little(FILE *f, UINT32 i)
{
    int rc;
    rc = fputc((i & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 8) & 0xff), f);
    if (rc == EOF)
        return rc;
    rc = fputc(((i >> 16) & 0xff), f);
    if (rc == EOF)
        return rc;
    return fputc(((i >> 24) & 0xff), f);
}
/* Formatting information for emacs in c-mode
 * Local Variables:
 * c-indent-level:4
 * c-continued-statement-offset:4
 * c-brace-offset:-4
 * c-brace-imaginary-offset:0
 * c-argdecl-indent:4
 * c-label-offset:-4
 * End:
 */

```

Listing Two

```

/* This is the header for endian.c - functions to read/write our
 * INT8, INT16 and INT32 types from/to a little-endian file.
 */

#ifndef __ENDIAN_H_INCLUDED__
#define __ENDIAN_H_INCLUDED__

/* Read the basic types as little-endian values. The return value will be
 * zero if successful, EOF, otherwise.
 */
int readINT8little(FILE *f, INT8 *i);
int readINT16little(FILE *f, INT16 *i);
int readINT32little(FILE *f, INT32 *i);
int readUINT8little(FILE *f, UINT8 *i);
int readUINT16little(FILE *f, UINT16 *i);
int readUINT32little(FILE *f, UINT32 *i);

/* Write the basic types as little-endian values. The return value will be
 * zero if successful, EOF otherwise.
 */
int writeINT8little(FILE *f, INT8 i);
int writeINT16little(FILE *f, INT16 i);
int writeINT32little(FILE *f, INT32 i);
int writeUINT8little(FILE *f, UINT8 i);
int writeUINT16little(FILE *f, UINT16 i);
int writeUINT32little(FILE *f, UINT32 i);

#endif /* __ENDIAN_H_INCLUDED__ */

/* Formatting information for emacs in c-mode
 * Local Variables:
 * c-indent-level:4
 * c-continued-statement-offset:4
 * c-brace-offset:-4
 * c-brace-imaginary-offset:0
 * c-argdecl-indent:4
 * c-label-offset:-4
 * End:
 */

```