7 Image Compression (GIF)

7.1 Introduction

Data communication increasingly involves the transmission of still and moving images. Compressing images into a standard form can give great savings in transmission times and storage lengths. Some of these forms are outlined in Table 7.1. The main parameters in a graphics file are:

- The picture resolution. This is defined by the number of pixels in the *x*- and *y*-directions.
- The number of colors per pixel. If *N* bits are used for the bit color then the total number of displayable colors will be 2^N. For example, an 8-bit color field defines 256 colors, a 24-bit color field gives 2^{24} or 16.7M colors. Most computer systems allow for 32-bit color, which gives over 4 billion colors.
- Palette size. Some systems reduce the number of bits used to display a color by reducing the number of displayable colors for a given palette size.

Table 7.1 Typical standard compressed graphics formats

File	*Compression type*	*Max. resolution or colors*	
TIFF	Huffman RLE and/or LZW	48-bit color	TIFF (tagged image file format) is typically used to transfer graphics from one computer system to another. It allows high resolutions and colors of up to 48 bits (16 bits for red, green and blue).
PCX	RLE	$65\,536 \times 65\,536$ (24-bit color)	Graphics file format which uses RLE to compress the image. Unfortunately, it make no provision for storing gray scale or color-correcting tables.
GIF	LZW	$65\,536 \times 65\,536$ (24-bit color, but only 256 displayable colors)	Standardized graphics file format which can be read by most graphics packages. It has similar graphics characteristics to PCX files and allows multiple images in a single file and interlaced graphics.
JPG	JPEG compression (DCT, Quantization and Huffman)	Depends on the compression	Excellent compression technique which produces lossy compression. It normally results in much greater compression than the methods outlined above.

7.2 Comparison of the different methods

This section uses example bitmapped images and shows how different techniques manage to compress them. Figure 7.1 shows an image and Table 7.2 shows the resultant file size when it is saved in different formats. It can be seen that the BMP file format has the largest storage. The two main forms of BMP files are RGB (red, green, blue) encoded and RLE encoded. RGB coding saves the bit-map in an uncompressed form, whereas the RLE coding will reduce the total storage by compressing repetitive sequences. Next is the PCX file which has limited compression abilities (the format used in this case is version 5). The GIF format manages to compress the file to around 40% of its original size and the TIF file achieves similar compression (mainly because both techniques use LZH compression). It can be seen that by far the best compression is achieved with JPEG, which in both forms has compressed the file to under 10% of its original size.

The reason that the compression ratios for GIF, TIF and BMP RLE are relatively high is that the image in Figure 7.1 contains a lot of changing data. Most images will compress to less than 10% because they have large areas which do not change much.

Figure 7.1 Sample graphics image

Table 7.2 Compression on a graphics file

Type	*Size(B)*	*Compression(%)*	
BMP	308 278	100.0	BMP, RBG encoded (640 × 480, 256 colors)
BMP	301 584	97.8	BMP, RLE encoded
PCX	274 050	88.9	PCX, Version 5
GIF	124 304	40.3	GIF, Version 89a, non-interlaced
GIF	127 849	41.5	GIF, Version 89a, interlaced
TIF	136 276	44.2	TIF, LZW compressed
TIF	81 106	26.3	TIF, CCITT Group 3, MONOCHROME
JPG	28 271	9.2	JPEG - JFIF Complaint (Standard coding)
JPG	26 511	8.6	JPEG - JFIF Complaint (Progressive coding)

Figure 7.2 shows a simple graphic of 500×500, 24-bit, which has large areas with identical colors. Table 7.3 shows that, in this case, the compression ratio is low. The RLE encoded BMP file is only 1% of the original as the graphic contains long runs of the same color. The GIF file has compressed to less than 1%. Note that the PCX, GIF and BMP RLE files have saved the image with only 256 colors. The JPG formats have the advantage that they have saved the image with the full 16.7M colors and give compression rates of around 2%.

Table 7.3 Compression on a graphics file with highly redundant data

Type	*Size (B)*	*Compression (%)*	
BMP	750 054	100.0	BMP, RBG encoded (500×500, 16.7M colors)
BMP	7 832	1.0	BMP, RLE encoded (256 colors)
PCX	31 983	4.3	PCX, Version 5 (256 colors)
GIF	4 585	0.6	GIF, Version 89a, non-interlaced (256 colors)
TIF	26 072	3.5	TIF, LZW compressed (16.7M colors)
JPG	15 800	2.1	JPEG (Standard coding, 16.7M colors)
JPG	12 600	1.7	JPEG (Progressive coding 16.7M colors)

7.3 GIF coding

The graphics interchange format (GIF) is the copyright of CompuServe Incorporated. Its popularity has increased mainly because of its wide usage on the Internet. CompuServe Incorporated, luckily, has granted a limited, non-exclusive, royalty-free license for the use of GIF (but any software using the GIF format must acknowledge the ownership of the GIF format).

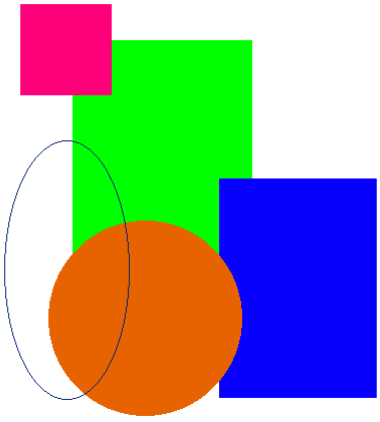

Figure 7.2 Sample graphics image

Most graphics software supports the Version 87a or 89a format (the 89a format is an update the 87a format). Both have basic specification:

- A header with GIF identification.

- A logical screen descriptor block which defines the size, aspect ratio and color depth of the image place.
- A global color table.
- Data blocks with bitmapped images and the possibility of text overlay.
- Multiple images, with image sequencing or interlacing. This process is defined in a graphic-rendering block.
- LZW compressed bitmapped images.

7.3.1 Color tables

Color tables store the color information of part of an image (a local color table) or they can be global (a global table).

7.3.2 Blocks, extensions and scope

Blocks can be specified into three groups: control, graphic-rendering and special purpose. Control blocks contain information used to control the process of the data stream or information used in setting hardware parameters. They include:

- GIF Header – which contains basic information on the GIF file, such as the version number and the GIF file signature.
- Logical screen descriptor – which contains information about the active screen display, such as screen width and height, and the aspect ratio.
- Global color table – which contains up to 256 colors from a palette of 16.7M colors (i.e. 256 colors with 24-bit color information).
- Data subblocks – which contain the compressed image data.
- Image description – which contains, possibly, a local color table and defines the image width and height, and its top left coordinate.
- Local color table – an optional block which contains local color information for an image as with the global color table, it has a maximum of 256 colors from a palette of 16.7M.
- Table-based image data – which contains compressed image data.
- Graphic control extension – an optional block which has extra graphic-rendering information, such as timing information and transparency.
- Comment extension – an optional block which contains comments ignored by the decoder.
- Plain text extension – an optional block which contains textual data.
- Application extension – which contains application-specific data. This block can be used by a software package to add extra information to the file.
- Trailer – which defines the end of a block of data.

7.3.3 GIF header

The header is six bytes long and identifies the GIF signature and the version number of the chosen GIF specification. Its format is:

- 3 bytes with the characters ‘G’, ‘I’ and ‘F’.
- 3 bytes with the version number (such as 87a or 89a). Version numbers are ordered with two digits for the year, followed by a letter (‘a’, ‘b’, and so on).

Program 7.1 is a C program for reading the 6-byte header and Sample run 7.1 shows a sample run with a GIF file. It can be seen that the file in the test run has the required signature and has been stored with Version 89a.

Program 7.1

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  fname[BUFSIZ], str[BUFSIZ];

   printf("Enter GIF file>>");
   gets(fname);

   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   fread(str,3,1,in);
   str[3]=NULL; /* terminate string */
   printf("Signature: %s\n",str);
   fread(str,3,1,in);
   str[3]=NULL; /* terminate string */
   printf("Version: %s\n",str);

   fclose(in);
   return(0);
}
```

Sample run 7.1

```
Enter GIF file>> clouds.gif
Signature: GIF
Version: 89a
```

7.3.4 Logical screen descriptor

The logical screen descriptor appears after the header. Its format is:

- 2 bytes with the logical screen width (unsigned integer).
- 2 bytes with the logical screen height (unsigned integer).
- 1 byte of a packed bit field, with 1 bit for global color table flag, 3 bits for color resolution, 1 bit for sort flag and 3 bits to give an indication of the number of colors in the global color table
- 1 byte for the background color index.
- 1 byte for the pixel aspect ratio.

Program 7.2 is a C program which reads the header and the logical descriptor field, and Sample run 7.2 shows a sample run. It can be seen, in this case, that the logic screen size is 640×480. The packed field, in this case, has a hexadecimal value of F7h, which is 1111 0111b in binary. Thus, all the bits of the packed bit field are set, apart from the sort flag. If this is set then the global color table is sorted in order of decreasing importance (the most

frequent color appearing first and the least frequent color last). The total number of colors in the global color table is found by raising 2 to the power of 1 + the color value in the packed bit field:

$$\text{Number of colors} = 2^{\text{Color value in packed bit field}+1}$$

In this case, there is a bit field of seven colors, thus the total number of colors is 2^8, or 256.

It can be seen that the aspect ratio in Sample run 7.2 is zero. If it is zero then no aspect ratio is given. If it is not equal to zero then the aspect ratio of the image is computed by:

$$\text{Aspect ratio} = \frac{\text{Pixel aspect ratio} + 15}{64}$$

where the pixel ratio is the pixel's width divided by its height.

Program 7.2

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  fname[BUFSIZ], str[BUFSIZ];
int   x,y;
char  color_index, aspect, packed;

   printf("Enter GIF file>>");
   gets(fname);

   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   fread(str,3,1,in);   str[3]=NULL; /* terminate string */
   printf("Signature: %s\n",str);
   fread(str,3,1,in);   str[3]=NULL; /* terminate string */
   printf("Version: %s\n",str);

   fread(&x,2,1,in); str[3]=NULL; /* terminate string */
   printf("Screen width: %d\n",x);
   fread(&y,2,1,in); str[3]=NULL; /* terminate string */
   printf("Screen height: %d\n",y);

   fread(&packed,1,1,in);
   printf("Packed: %x\n",packed & 0xff); /* mask-off the bottom 8 bits */
   fread(&color_index,1,1,in);
   printf("Color index: %d\n",color_index);
   fread(&aspect,1,1,in);
   printf("Aspect ratio: %d\n",aspect);

   fclose(in);
   return(0);
}
```

Sample run 7.2

```
Enter GIF file>> clouds.gif
Signature: GIF
Version: 89a
Screen width: 640
Screen height: 480
Packed: f7
Color index: 0
Aspect ratio: 0
```

7.3.5 Global color table

After the header and the logical display descriptor is the global color table. It contains up to 256 colors from a palette of 16.7M colors. Each of the colors is defined as a 24-bit color of red (8 bits), green (8 bits) and blue (8 bits). The format in memory is:

```
RRRRRRRR
GGGGGGGG
BBBBBBBB
RRRRRRRR
GGGGGGGG
BBBBBBBB
 :     :
RRRRRRRR
GGGGGGGG
BBBBBBBB
```

Thus the number of bytes that the table will contain will be:

$$\text{Number of bytes} = 3 \times 2^{\text{Size of global color table} + 1}$$

The 24-bit color scheme allows a total of 16 777 216 (2^{24}) different colors to be displayed. Table 7.4 defines some colors in the RGB (red/green/blue) strength. The format is `rrggbbh`, where `rr` is the hexadecimal equivalent for the red component, `gg` the hexadecimal equivalent for the green component and `bb` the hexadecimal equivalent for the blue component. For example, in binary:

`000000000000000000000000`	represents black (000000h)
`111111111111111111111111`	represents white (FFFFFFh)
`011101110111011101110111`	represents gray (777777h)
`111110101110010100000011`	represents yellow (FCE503h)
`001110100000101101011001`	represents purple (3A0B59h)

Program 7.3 is a C program which reads the header, the image descriptor and the color table. Sample run 7.3 shows a truncated color table. The first three are:

0111 1011 1010 1101 1101 0110 (7BADD6h)
1000 0100 1011 0101 1101 1110 (84B5DEh)
0111 0011 1010 1101 1101 0110 (73ADD6h)

Table 7.4 Hexadecimal colors for 24-bit color representation

Color	*Code*	*Color*	*Code*
White	FFFFFFh	Dark red	C91F16h
Light red	DC640Dh	Orange	F1A60Ah
Yellow	FCE503h	Light green	BED20Fh
Dark green	088343h	Light blue	009DBEh
Dark blue	0D3981h	Purple	3A0B59h
Pink	F3D7E3h	Nearly black	434343h
Dark gray	777777h	Gray	A7A7A7h
Light gray	D4D4D4h	Black	000000h

These colors have a strong blue component (D6h and DEh) and reduced strength red and green components. The image itself is a picture of clouds on a blue sky, thus, the image is likely to have strong blue colors.

Program 7.3

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  fname[BUFSIZ], str[BUFSIZ];
int   x,y,i;
char  color_index, aspect, packed,red,blue,green;

   printf("Enter GIF file>>");
   gets(fname);
   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   fread(str,3,1,in);   str[3]=NULL; /* terminate string */
   printf("Signature: %s\n",str);
   fread(str,3,1,in);   str[3]=NULL; /* terminate string */
   printf("Version: %s\n",str);

   fread(&x,2,1,in); str[3]=NULL; /* terminate string */
   printf("Screen width: %d\n",x);
   fread(&y,2,1,in); str[3]=NULL; /* terminate string */
   printf("Screen height: %d\n",y);

   fread(&packed,1,1,in);
   printf("Packed: %x\n",packed & 0xff); /* mask-off the bottom 8 bits */
   fread(&color_index,1,1,in);
   printf("Color index: %d\n",color_index);
   fread(&aspect,1,1,in);
   printf("Aspect ratio: %d\n",aspect);

   for (i=0;i<64;i++)
   {
      fread(&red,1,1,in);
```

```
      printf("Red: %x ",red & 0xff);      /* display 8 bits */
      fread(&green,1,1,in);
      printf("Green: %x ",green & 0xff);  /* display 8 bits */
      fread(&blue,1,1,in);
      printf("Blue: %x\n",blue & 0xff);   /* display 8 bits */
   }

   fclose(in);
   return(0);
}
```

Sample run 7.3

```
Enter GIF file>> clouds.gif
Signature: GIF
Version: 89a
Screen width: 640
Screen height: 480
Packed: f7
Color index: 0
Aspect ratio: 0
Red: 7b Green: ad Blue: d6
Red: 84 Green: b5 Blue: de
Red: 73 Green: ad Blue: d6
Red: 7b Green: ad Blue: de
Red: 94 Green: bd Blue: de
Red: 7b Green: b5 Blue: de
Red: 8c Green: b5 Blue: de
Red: 8c Green: bd Blue: de
Red: 9c Green: c6 Blue: de
Red: ce Green: de Blue: ef
Red: de Green: e7 Blue: ef
Red: a5 Green: c6 Blue: e7
   ::::::
Red: 8c Green: bd Blue: e7
Red: ff Green: ff Blue: f7
Red: ad Green: d6 Blue: ef
Red: 8c Green: b5 Blue: e7
Red: 84 Green: b5 Blue: e7
```

7.3.6 Image descriptor

After the global color table is the image descriptor. Its format is:

- 1 byte for the image separator (always 2Ch).
- 2 bytes for the image left position (unsigned integer).
- 2 bytes for the image top position (unsigned integer).
- 2 bytes for the image width (unsigned integer).
- 2 bytes for the image height (unsigned integer).
- 1 byte of a packed bit field, with 1 bit for local color table flag, 1 bit for interlace flag, 1 bit for sort flag, 2 bits are reserved and 3 bits for the size of the local color table.

Program 7.4 is a C program which searches for the image separator (2Ch) and displays the image descriptor data that follows. Sample run 7.4 shows a sample run. It can be seen from this sample run that the image is to be displayed at (0,0), its width is 640 pixels and its height is 480 pixels. The packed bit field contains all zeros, thus there is no local color table (and the global color table should be used).

Program 7.4

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  fname[BUFSIZ];
int   i,left,top,width,height;
char  ch,packed;

   printf("Enter GIF file>>");
   gets(fname);

   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   do
   {
      fread(&ch,1,1,in);
   } while (ch!=0x2C); /* find image seperator */

   fread(&left,2,1,in);
   printf("Image left position: %d\n",left);
   fread(&top,2,1,in);
   printf("Image top position: %d\n",top);
   fread(&width,2,1,in);
   printf("Image width: %d\n",width);
   fread(&height,2,1,in);
   printf("Image height: %d\n",height);
   fread(&packed,1,1,in);
   printf("Packed: %x\n",packed & 0xff);
   fclose(in);
   return(0);
}
```

Sample run 7.4

```
Enter GIF file>> clouds.gif
Image left position: 0
Image top position: 0
Image width: 640
Image height: 480
Packed: 0
```

7.3.7 Local color table

The local color table is an optional block which defines the color map for the image that precedes it. The format is identical to the global color map, that is, three bytes for each of the colors.

7.3.8 Table-based image data

The table-based image data follows the local color table. This table contains compressed image data. It consists of a series of subblocks of up to 255 bytes. The data consists of an index to the color table (either global or local) for each pixel in the image. As the global (or local) color table has 256 entries, the data value (in its uncompressed form) will range from 0

to 255 (8 bits). The tables format is:

- 1 byte for the LZW minimum code size, which is the initial number of bits used in the LZW coding.
- *N* bytes for the LZW compressed image data. The first block is preceded by the data size.

GIF coding uses the variable-length-code LZW technique where a variable-length code replaces image data (pixel color references). These variable-length codes are specified in a Huffman code table. The encoder replaces the data from the input and builds a dictionary with the patterns in the data. Every new pattern is entered into the dictionary and the index value of the table is added to coded data. When a previously stored pattern is encountered, its dictionary index value is added to the coded data. The decoder takes the compressed data and builds the dictionary which is identical to the encoder. It then replaces indexed terms from the dictionary.

The VLC algorithm uses an initial code size to specify the initial number of bits used for the compression codes. When the number of patterns detected by the encoder exceeds the number of patterns encodable with the current number of bits then the number of bits per LZW is increased by 1.

Program 7.5 reads the LZW code size byte. The byte after this is the block size, followed by the number of bytes of data as defined in the block size byte. Sample run 7.5 gives a sample run. It can be seen that the initial LZW code size is 8 and that the block size of the first block is 254 bytes. The dictionary entries will thus start at entry 256 (2^8).

Program 7.5

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  fname[BUFSIZ];
int   i,left,top,width,height;
char  ch,packed,code,block;

   printf("Enter GIF file>>");
   gets(fname);

   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   do
   {
      fread(&ch,1,1,in);
   } while (ch!=0x2C);
   fread(&left,2,1,in);
   printf("Image left position: %d\n",left);
   fread(&top,2,1,in);
   printf("Image top position: %d\n",top);
   fread(&width,2,1,in);
   printf("Image width: %d\n",width);
   fread(&height,2,1,in);
   printf("Image height: %d\n",height);
   fread(&packed,1,1,in);
   printf("Packed: %x\n",packed & 0xff);
```

```
   fread(&code,1,1,in);
   printf("LZW code size: %d\n",code & 0xff);
   fread(&block,1,1,in);
   printf("Block size: %d\n",block & 0xff);
   fclose(in);
   return(0);
}
```

Sample run 7.5

```
Enter GIF file>> clouds.gif
Image left position: 0
Image top position: 0
Image width: 640
Image height: 480
Packed: 0
LZW code size: 8
Block size: 254
```

7.3.9 Graphic control extension

The graphic control extension is optional and contains information on the rendering of the image that follows. Its format is:

- 1 byte with the extension identifier (21h).
- 1 byte with the graphic control label (F9h).
- 1 byte with the block size following this field and up to but not including, the end terminator. It always has a fixed value of 4.
- 1 byte with a packed array of which the first 3 bits are reserved, 3 bits define the disposal method, 1 bit defines the user input flag and 1 bit defines the transparent color flag.
- 2 bytes with the delay time for the encode wait, in hundreds of a seconds, before encoding the image data.
- 1 byte with the transparent color index.
- 1 byte for the block terminator (00h).

7.3.10 Comment extension

The comment extension is optional and contains information which is ignored by the encoder. Its format is:

- 1 byte with the extension identifier (21h).
- 1 byte with the comment extension label (FEh).
- *N* bytes, with comment data.
- 1 byte for the block terminator (00h).

7.3.11 Plain text extension

The plain text extension is optional and contains text information. Its format is:

- 1 byte with the extension identifier (21h).
- 1 byte with the plain text label (01h).
- 1 byte with the block size. This is the number of bytes after the block size field up to but not including the beginning of the plain text data block. It always contains the value 12.

- 2 bytes for the text grid left position.
- 2 bytes for the text grid top position.
- 2 bytes for the text width.
- 2 bytes for the text height.
- 1 byte for the character cell width.
- 1 byte for the character cell height.
- 1 byte for the text foreground color.
- 1 byte for the text background color.
- *N* bytes for the plain text data.
- 1 byte for the block terminator (00h).

7.3.12 Application extension

The application extension is optional and contains information for application programs. Its format is:

- 1 byte with the extension identifier (21h).
- 1 byte with the application extension label (FFh).
- 1 byte for the block size. This is the number of bytes after the block size field up to but not including the beginning of the application data. It always contains the value 11.
- 8 bytes for the application identifier.
- 3 bytes for the application authentication code.
- *N* bytes, for the application data.
- 1 byte for the block terminator (00h).

7.3.13 Trailer

The trailer indicates the end of the GIF file. Its format is:

- 1 byte identifying the trailer (3Bh).

7.4 TIFF coding

Tag image file format (TIFF) is an excellent method of transporting images between file systems and software packages. It is supported by most graphics import packages and has a high resolution that is typically used when scanning images. There are two main types of TIFF coding, baseline TIFF and extended TIFF. It can also use different compression methods and different file formats, depending on the type of data stored.

In TIFF 6.0, defined in June 1992, the pixel data can be stored in several different compression formats, such as:

- Code number 1, no compression.
- Code number 2, CCITT Group 3 modified Huffman RLE encoding.
- Code number 3, Fax-compatible CCITT Group 3.
- Code number 4, Fax-compatible CCITT Group 4.
- Code number 5, LZW compression.

Codes 1 and 2 are baseline TIFF files whereas the others are extended.

7.4.1 File structure

TIFF files have a three-level hierarchy:

- A file header.
- One or more IFDs (image file directories). These contain codes and their data (or pointers to the data).
- Data.

The file header contains 8 bytes: a byte order field (2 bytes), the version number field (2 bytes) and the pointer to the first IFD (4 bytes). Figure 7.3 shows the file header format. The byte order field defines whether Motorola architecture is used (the character sequence is 'MM', or 4D4Dh) or Intel architecture (the character sequence is 'II', or 4949h). The Motorola format defines that the bytes are ordered from the most significant to the least significant, the Intel format defines that the bytes are organized from least significant to the most significant.

The version number field always contains the decimal number 42. It is used to identify that the file is TIFF format.

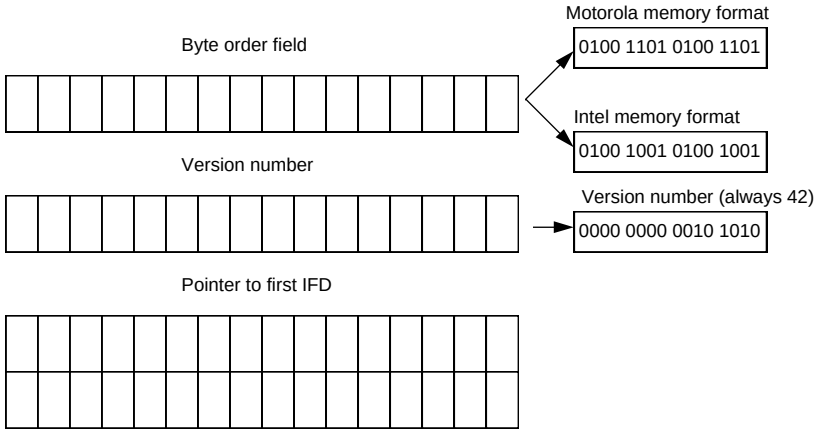

Figure 7.3 TIFF file header

The first IFD offset pointer is a 4-byte pointer to the first IFD. If the format is Intel then the bytes are arranged from least significant to most significant else they are arranged from most significant to least significant.

Program 7.6 is a C program which reads the header of a TIFF file and Sample run 7.6 shows that, in this case, it uses the Intel format and the second byte field contains 2Ah (or 42 decimal).

Program 7.6

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
char  ch1,ch2,fname[BUFSIZ];
```

```
   printf("Enter TIFF file>>");
   gets(fname);

   if ((in=fopen(fname,"r"))==NULL)
   {
      printf("Can't find file %s\n",fname);
      return(1);
   }

   ch1=fgetc(in); ch2=getc(in);
   printf("Memory model %c%c\n",ch1,ch2);
   ch1=fgetc(in); ch2=getc(in);
   printf("Version %x%x\n",ch2,ch1);

   fclose(in);
   return(0);
}
```

Sample run 7.6

```
Enter TIFF file>> image1.tif
Memory model II
Version 02a
```

7.4.2 IFD

Typically, the first IFD will be the only IFD, which is pointed to by the first IFD in the header field.

7.4.3 Compression code 2: Huffman RLE coding

TIFF compression code 2, uses the CCITT Group 3 type compression, which is a modified Huffman coding and is used in many fax transmissions. It specifies a 1-bit monochrome code with alternate black and white sequences of pixels. Tables 7.5 and 7.6 give the predefined coding table for white and black sequence runs. These tables contain codes in which the most frequent run lengths are coded with a short code. The compressed code always starts on white code. Codes themselves range from 0 to 63. Values from 64 to 2560 use two codes. The first gives the multiple of 64 followed by the normally coded remainder. There is no special end-of-line identifier because the size of the image is known by the defined ImageWidth tag field. There are thus, `ImageWidth` pixels on a line.

Table 7.5 White run-length coding

Run length	*Coding*	*Run length*	*Coding*	*Run length*	*Coding*	*Run length*	*Coding*
0	00110101	1	000111	2	0111	3	1000
4	1011	5	1100	6	1110	7	1111
8	10011	9	10100	10	00111	11	01000
12	001000	13	000011	14	110100	15	110101
16	101010	17	101011	18	0100111	19	0001100
61	00110010	62	00110011	63	00110100	64	110011

Table 7.6 Black run-length coding

Run length	*Coding*	*Run length*	*Coding*	*Run length*	*Coding*	*Run length*	*Coding*
0	0000110111	1	010	2	11	3	10
4	011	5	0011	6	0010	7	00011
8	000101	9	000100	10	0000100	11	0000101
12	0000111	13	00000100	14	00000111	15	000011000
16	0000010111	17	0000011000	18	0000001000	19	00001100111
61	000001011010	62	0000001100110	63	000001100111	64	0000001111

For example, if the data were:

16 white 4 black 16 white 2 black 63 white 10 black 63 white

it would be coded as:

```
101010   011   101010   11 00110100 0000100  00110100
```

This would take 40 bits to code, whereas it would take 184 bits if coded with pixel colors (i.e., 16+4+16+2+63+10+63). This results in a compression ratio of 4.6:1.

7.4.4 Compression code 5: LZW compression

The compression technique used by TIFF code 5 is the same as is used in GIF files, but has a fixed code size of 8. The dictionary starts with the values 0 to 255 stored in the entries 0 to 255. There are two codes for `Clear` (at 256) and `EndOfInformation` (at 257) and the dictionary is then built up from 258 to 4095. The `Clear` code is a special code which resets the dictionary entries to the original entries from 0 to 255.

A basic encoding algorithm could be:

```
Byte: byte;
Buffer, Test, String: string;
Table: array[1..4096] of string;

begin
   clear Table;   clear Buffer; clear Test; clear String;

   write ClearCode code;

   while (valid data)
   begin
      read Byte;
      Test=String+Byte;
      if (Test in Table) then String=String+Byte;
      else
      begin
         write String code;
         add Test to Table;
```

```
            String=Byte;
          end;
        end;

        write String code;
        write EndOfInformation code.
    end.
```

7.5 GIF interlaced images

GIF images can be stored in an interlaced manner. This facility is useful when receiving information over a relatively slow transmission line, as it allows an outline of an image to be displayed before the entire image has been encoded (or received). The images stored are:

Group 1: Starting at row 0, every 8th row.
Group 2: Starting at row 4, every 8th row.
Group 3: Starting at row 2, every 4th row.
Group 4: Starting at row 1, every 2nd row.

For example if the image has 16 rows (0–15) then the following would be stored:

	Scanned line displayed			
	1	2	3	4
Row 0	X			
Row 1				X
Row 2			X	
Row 3				X
Row 4		X		
Row 5				X
Row 6			X	
Row 7				X
Row 8	X			
Row 9				X
Row 10			X	
Row 11				X
Row 12		X		
Row 13				X
Row 14			X	
Row 15				X

It can be seen that the first 1/8 of the data displays an outline of the image, the next 1/8 then improves the quality. After this, the next 1/4 further improves the quality and then the final 1/2 gives the completed image.