8 Image Compression (JPEG)

8.1 Introduction

JPEG is an excellent compression technique which produces lossy compression (although in one mode it is lossless). As seen from the previous chapter, it has excellent compression ratio when applied to a color image. This chapter introduces the JPEG standard and the method used to compress an image. It also discusses the JFIF file standard which defines the file format for JPEG encoded images. Along with GIF files, JPEG is now one of the most widely used standards for image compression.

8.2 JPEG coding

A typical standard for image compression has been devised by the Joint Photographic Expert Group (JPEG), a subcommittee of the ISO/IEC, and the standards produced can be summarized as follows:

It is a compression technique for gray-scale or color images and uses a combination of discrete cosine transform, quantization, run-length and Huffman coding.

It has resulted from research into compression ratios and the resultant image quality. The main steps are:

- Data blocks — Generation of data blocks
- Source-encoding — Discrete cosine transform and quantization
- Entropy-encoding — Run-length encoding and Huffman encoding

Unfortunately, compared with GIF, TIFF and PCX, the compression process is relatively slow. It is also lossy in that some information is lost in the compression process. This information is perceived to have little effect on the decoded image.

GIF files typically take 24-bit color information (8 bits for red, 8 bits for green and 8 bits for blue) and convert it into an 8-bit color palette (thus reducing the number of bits stored to approximately one-third of the original). It then uses LZW compression to further reduce the storage. JPEG operates differently in that it stores changes in color. As the eye is very sensitive to brightness changes and less on color changes, then if these changes are similar to the original then the eye perceives the recovered image as very similar to the original.

8.2.1 Color conversion and subsampling

The first part of the JPEG compression separates each color component (red, green and blue) in terms of luminance (brightness) and chrominance (color information). JPEG allows more losses on the chrominance and less on the luminance as the human eye is less sensitive to color changes than to brightness changes. In an RGB image, all three colors carry some brightness information but the green component has a stronger effect on brightness than the blue component.

A typical scheme for converting RGB into luminance and color is CCIR 601, which converts the components into Y (can be equated to brightness), C_b (blueness) and C_r (redness). The Y component can be used as a black and white version of the image.

Each component is computed from the RGB components as:

$$Y = 0.299R + 0.587G + 0.114B$$
$$C_b = 0.1687\mathrm{R} - 0.3313G + 0.5B$$
$$C_r = 0.5R - 0.4187G + 0.0813B$$

For the brightness (Y) it can be seen that green has the most effect and blue has the least effect. For the redness (C_r), the red color (of course) has the most effect and green the least. For blueness (C_b), the blue color has the most effect and green the least. Note that the YC_bC_r components are often known as YUV (especially in TV systems).

A subsampling process is then samples the C_b and C_r components at a lower rate than the Y component. A typical sampling rate is four samples of the Y component for a single sample of the C_b and C_r components (4:1:1). This sampling rate is normally set with the compression parameters, the lower the sampling, the smaller the compressed data and the shorter the compression time. The JPEG header contains all the information necessary to properly decode the JPEG data.

8.2.2 DCT coding

The DCT (discrete cosine transform) converts intensity data into frequency data, which can be used to tell how fast the intensities vary. In JPEG coding the image is segmented into 8×8 pixel rectangles, as illustrated in Figure 8.1. If the image contains several components (such as Y,C_b,C_r or R,G,B), then each of the components in the pixel blocks is operated on separately. If an image is subsampled, there will be more blocks of some components than of others. For example, for 2×2 sampling there will be four blocks of Y data for each block of C_b or C_r data.

The data points in the 8×8 pixel array starts at the upper right at $(0,0)$ and finish at the lower right at $(7,7)$. At the point (x,y) the data value is $f(x,y)$. The DCT produces a new 8×8 block $(u \times v)$ of transformed data using the formula:

$$F(u,v) = \frac{1}{4} C(u)C(v) \left[\sum_{x=0}^{7} \sum_{y=0}^{7} f(x,y) \cos \frac{(2x+1)u\pi}{16} \cos \frac{(2y+1)v\pi}{16} \right]$$

where $C(z) = \dfrac{1}{\sqrt{2}}$ if $z = 0$

or $= 1$ if $z \neq 0$

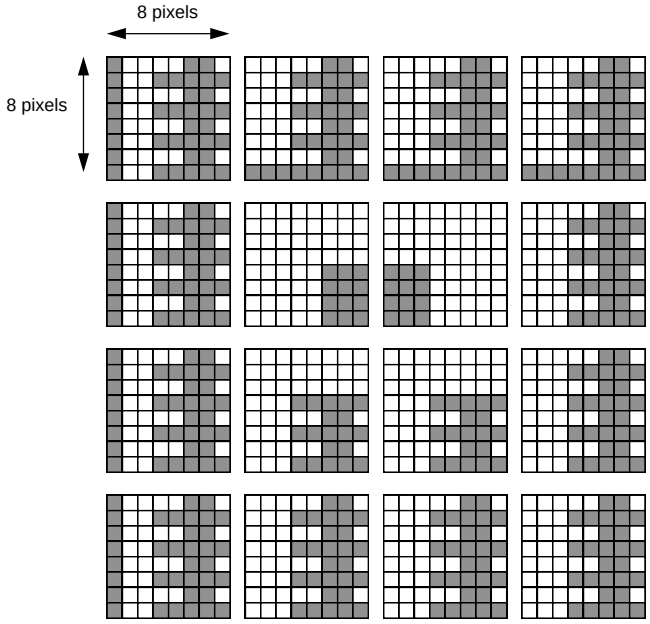

Figure 8.1 Segment of an image in 8x8 pixel blocks

This results in an array of space frequency $F(u,v)$ which gives the rate of change at a given point. These are normally 12-bit values which give a range of 0 to 1024. Each component specifies the degree to which the image changes over the sampled block. For example:

- F(0,0) gives the average value of the 8×8 array.
- $F(1,0)$ gives the degree to which the values change slowly (low frequency).
- $F(7,7)$ gives indicates the degree to which the values change most quickly in both directions (high frequency).

The coefficients are equivalent to representing changes of frequency within the data block. The value in the upper left block $(0,0)$ is the DC or average value. The values to the right of a row have increasing horizontal frequency and the values to the bottom of a column have increasing vertical frequency. Many of the bands though, end up having zero, or almost zero terms.

Program 8.1 gives a C program which determines the DCT of an 8×8 block and Sample run 8.1 shows a sample run with the resultant coefficients.

Program 8.1

```
#include <stdio.h>
#include <math.h>
#define  PI 3.1415926535897

int   main(void)
{
int   x,y,u,v;
float in[8][8]=   {{144,139,149,155,153,155,155,155},
            {151,151,151,159,156,156,156,158},
```

```
            {151,156,160,162,159,151,151,151},
            {158,163,161,160,160,160,160,161},
            {158,160,161,162,160,155,155,156},
            {161,161,161,161,160,157,157,157},
            {162,162,161,160,161,157,157,157},
            {162,162,161,160,163,157,158,154}};
float out[8][8],sum,Cu,Cv;

   for (u=0;u<8;u++)
   {
      for (v=0;v<8;v++)
      {
         sum=0;
         for (x=0;x<8;x++)
            for (y=0;y<8;y++)
            {
               sum=sum+in[x][y]*cos(((2.0*x+1)*u*PI)/16.0)*
                  cos(((2.0*y+1)*v*PI)/16.0);
            }
         if (u==0) Cu=1/sqrt(2); else Cu=1;
         if (v==0) Cv=1/sqrt(2); else Cv=1;

         out[u][v]=1/4.0*Cu*Cv*sum;
         printf("%8.1f ",out[u][v]);
      }
      printf("\n");
   }
   printf("\n");
   return(0);
}
```

The program uses a fixed 8×8 block of:

```
144   139   149   155   153   155   155   155
151   151   151   159   156   156   156   158
151   156   160   162   159   151   151   151
158   163   161   160   160   160   160   161
158   160   161   162   160   155   155   156
161   161   161   161   160   157   157   157
162   162   161   160   161   157   157   157
162   162   161   160   163   157   158   154
```

Sample run 8.1

```
  1257.9      2.3     -9.7     -4.1      3.9      0.6     -2.1      0.7
   -21.0    -15.3     -4.3     -2.7      2.3      3.5      2.1     -3.1
   -11.2     -7.6     -0.9      4.1      2.0      3.4      1.4      0.9
    -4.9     -5.8      1.8      1.1      1.6      2.7      2.8     -0.7
     0.1     -3.8      0.5      1.3     -1.4      0.7      1.0      0.9
     0.9     -1.6      0.9     -0.3     -1.8     -0.3      1.4      0.8
    -4.4      2.7     -4.4     -1.5     -0.1      1.1      0.4      1.9
    -6.4      3.8     -5.0     -2.6      1.6      0.6      0.1      1.5
```

Notice that the values of the most significant values are in the top left-hand corner and that many terms are near to zero. It is this property which allows many values to become zeros when quantized. These zeros can then be compressed using run-length coding and Huffman codes.

8.2.3 Quantization

The next stage of the JPEG compression is quantization where bias is given to lower-frequency components. JPEG divides each of the DCT values by a quantization factor, which is then rounded to the nearest integer. As the DCT factors are 8×8 then a table of 8×8 of quantization factors are used, corresponding to each term of the DCT output. The JPEG file then stores this table so that the decoding process may use this table or a standard quantization table. Note that files with multiple components must have multiple tables, such as one each for the Y, C_b and C_r components.

For example the values of the quantized high-frequency term (such as $F(7,7)$) could have a term of around 100, while the low-frequency term could have a factor of 16. These values define the accuracy of the final value. When decoding, the original values are (approximately) recovered by multiplying by the quantization factor.

Figure 8.2 shows that, for a factor of 100, the values between 50 and 150 would be quantized as a 1, thus the maximum error would be ±50. The maximum error for the factor of 16 is ±8. Thus the maximum error of the final unquantized value for a scale factor of 100 is 1.22% (5000/4096), while a factor of 16 gives a maximum error of 0.20% (800/4096). So, using the factors of 100 for $F(7,7)$ and 16 for $F(0,0)$, and a 12-bit DCT, the $F(0,0)$ term would range from 0 to 256 and the $F(7,7)$ term would range from 0 to 41. The $F(0,0)$ term could be coded with 8 bits (0 to 255) and the $F(7,7)$ term with 6 bits (0 to 63).

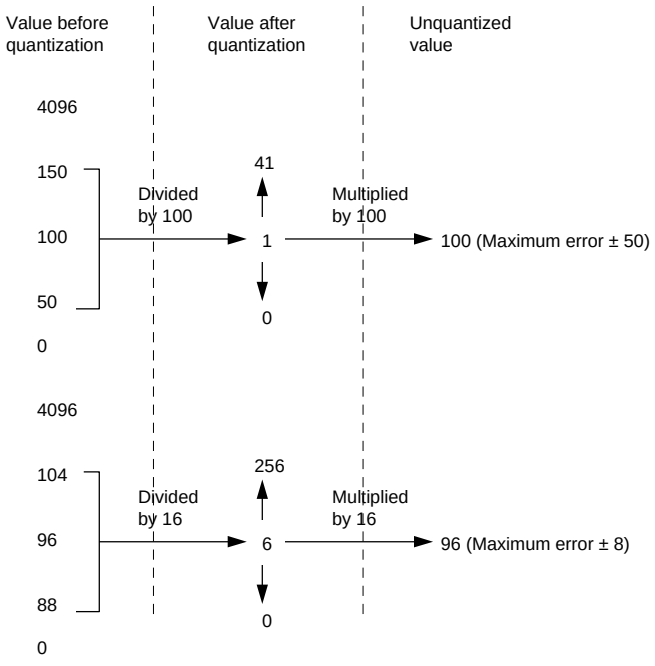

Figure 8.2 Example of quantization

Program 8.2 normalizes and quantizes (to the nearest integer) the example given previously. To recap, the input 8×8 block is:

```
144   139   149   155   153   155   155   155
151   151   151   159   156   156   156   158
```

```
151   156   160   162   159   151   151   151
158   163   161   160   160   160   160   161
158   160   161   162   160   155   155   156
161   161   161   161   160   157   157   157
162   162   161   160   161   157   157   157
162   162   161   160   163   157   158   154
```

The applied normalization matrix is:

```
5     3     4     4     4     3     5     4
4     4     5     5     5     6     7     12
8     7     7     7     7     15    11    11
9     12    13    15    18    18    17    15
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
```

Program 8.2

```
#include <stdio.h>
#include <math.h>
#define  PI 3.1415926535897

int   main(void)
{
int   x,y,u,v;
float in[8][8]=   {
      {144,139,149,155,153,155,155,155},
      {151,151,151,159,156,156,156,158},
      {151,156,160,162,159,151,151,151},
      {158,163,161,160,160,160,160,161},
      {158,160,161,162,160,155,155,156},
      {161,161,161,161,160,157,157,157},
      {162,162,161,160,161,157,157,157},
      {162,162,161,160,163,157,158,154}};

float norm[8][8]= {
   {5,3,4,4,4,3,5,4},
   {4,4,5,5,5,6,7,12},
   {8,7,7,7,7,15,11,11},
   {9,12,13,15,18,18,17,15},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20}};

int   out[8][8];
float sum,Cu,Cv;

   for (u=0;u<8;u++)
   {
      for (v=0;v<8;v++)
      {
         sum=0;
         for (x=0;x<8;x++)
            for (y=0;y<8;y++)
            {
               sum=sum+in[x][y]*cos(((2.0*x+1)*u*PI)/16.0)*
                  cos(((2.0*y+1)*v*PI)/16.0);
            }
         if (u==0) Cu=1/sqrt(2); else Cu=1;
         if (v==0) Cv=1/sqrt(2); else Cv=1;
```

```
            out[u][v]=(int)1/4.0*Cu*Cv*sum/norm[u][v];
            printf("%8d ",out[u][v]);
         }
         printf("\n");
      }

      printf("\n");
      return(0);
}
```

It can be seen from Sample run 8.2 that most of the normalized and quantized components are zero. This helps in the next stages of the compression, which involve either LZW or RLE. Thus, a scheme which stores similar values results in a larger compression than non-arranged values. It can also be seen from Sample run 8.2 that most of the non-zero values are in the top left-hand corner.

To achieve the compression, the DC components are stored as the difference in the DC value from one block to the next. This is because DC components, from block to block tend to be similar. The AC components are then stored logically in a zigzag order, as illustrated in Figure 8.3. This puts the lowest-frequency components first. Typically the quantized high-frequency components will be zero, and the final compression stage will compress these to a high degree.

Sample run 8.2 would be stored as:

```
251, 0, -5, -1, -3, -2, 0, -1, 0, 0, 0, 0, -1, 0, 0, 0, 0,…,0
```

which has a run of 51 zeros (as well as an earlier run of 4 zeros).

Sample run 8.2

```
 251    0   -2   -1    0    0    0    0
  -5   -3    0    0    0    0    0    0
  -1   -1    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
   0    0    0    0    0    0    0    0
```

8.2.4 Final compression

The final part of JPEG compression uses either a modified Huffman coding or arithmetic coding. Huffman coding is by far the most popular technique, but tends to lead to a larger compressed file.

Data values are coded with a modified Huffman code called the variable-length code (VLC). This encodes values as the difference between consecutive values. A positive value is stored with its binary equivalent and a negative value as the one's complement (all the bits inverted) equivalent, such as:

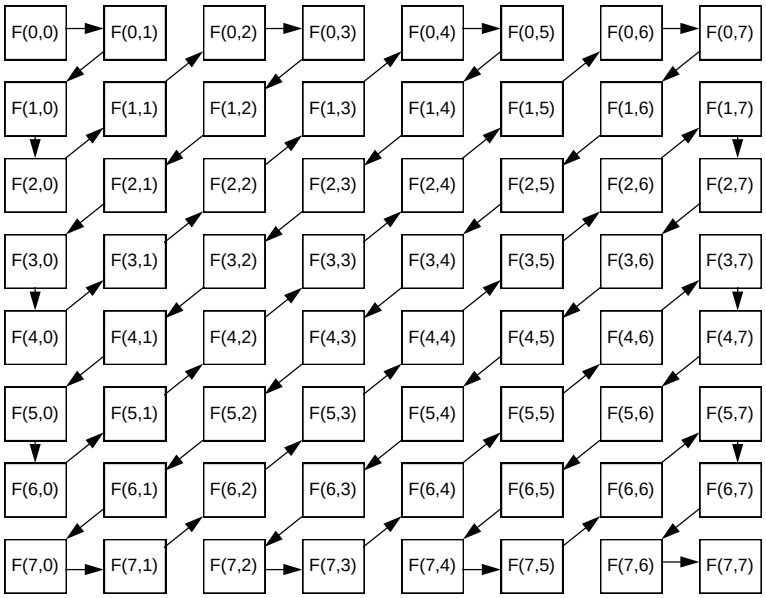

Figure 8.3 Zigzag storage

5	101	–5	010
10	1010	–10	0101
1	1	–1	0
23	10111	–23	01000

This difference value is preceded by a 4-bit binary value which defines the number of bits in the data values. Figure 8.4 gives an example. In this case, the data is 12, 10, 11, 11 and 11. The initial value for the difference encoding is taken as zero, thus the difference values will be 12, –2, 1, 0, 0. The first four bits will be 0100 (4) as the value 12 requires four bits. Next the value of 12 is stored (1010). The next difference is –2, which is 01 in 1's complement. This requires two bits, thus the next four bits will be 0010 (to define two bits), followed by the –2 value (01). This then continues. Note that a zero value is stored as a single 4-bit value of 0000, so no other bits follow it.

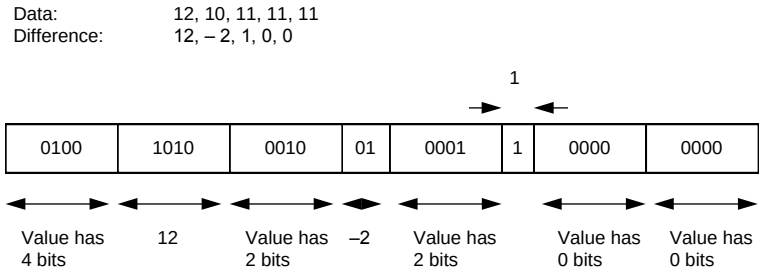

Figure 8.4 VLC storage

AC components (such as $F(0,1)$, $F(1,0)$, and so on) are stored as an 8-bit Huffman code fol-

lowed by a variable-length integer. The Huffman code is made up of high 4 bits which give the number of zero values preceding this value, and the low 4 bits which give the length of the variable-length integer. Figure 8.5 shows an example of AC coding with the data 0, 0, 0, 0, 2, 0, 0, 6. In this case, the first four bits of the 8-bit Huffman code (before it is converted to a Huffman code) is 0101, because there are four consecutive zeros. The value after these zeros is 2, which requires only two bits. Thus, the second part of the Huffman code (before coding) will be 0010 to specify two bits. Next, the data contains two zeros so the Huffman code (before coding) will be 0010. The data after the two zeros is a 6 which requires four bits, thus the second part of the Huffman code (before coding) is 0100. After this, the data value for 6 is represented in binary (1010). The AC components contain many runs of zeros so the code produced will tend to be extremely compressed.

The binary value of 0000 0000 (00h) can never occur in the AC coding scheme. This code is used as a special code to identify that all of the values until the end of a block are zero. This is a common occurrence and thus saves coding bits.

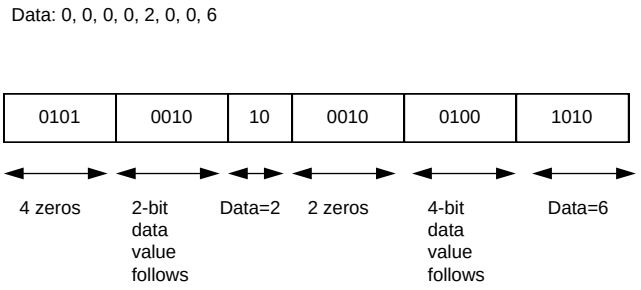

Figure 8.5 Storage of AC components

8.3 JPEG decoding

JPEG decoding involves reversing the process:

- Uncompression.
- Unquantizing (using the stored table or a standard table of factors).
- Reverse DCT.
- Block regeneration.

The DCT is reversed with the transform:

$$f(x,y) = \frac{1}{4}\left[\sum_{u=0}^{7}\sum_{v=0}^{7} C(u)C(v)F(u,v)\cos\frac{(2x+1)u\pi}{16}\cos\frac{(2y+1)v\pi}{16}\right]$$

Program 8.3 uses this inverse transform and contains the normalized and quantized coefficients from the previous example. To recap, the input data was:

```
144   139   149   155   153   155   155   155
151   151   151   159   156   156   156   158
151   156   160   162   159   151   151   151
```

```
158   163   161   160   160   160   160   161
158   160   161   162   160   155   155   156
161   161   161   161   160   157   157   157
162   162   161   160   161   157   157   157
162   162   161   160   163   157   158   154
```

The applied normalization matrix was:

```
5     3     4     4     4     3     5     4
4     4     5     5     5     6     7     12
8     7     7     7     7     15    11    11
9     12    13    15    18    18    17    15
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
20    20    20    20    20    20    20    20
```

Program 8.3

```
#include <stdio.h>
#include <math.h>
#define  PI 3.1415926535897

int   main(void)
{
int   x,y,u,v;
int   in[8][8]=   {{251,0,-2,-1,0,0,0,0},
            {-5,-3,0,0,0,0,0,0},
            {-1,-1,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0},
            {0,0,0,0,0,0,0,0}};

float norm[8][8]= {{5,3,4,4,4,3,5,4},
   {4,4,5,5,5,6,7,12},
   {8,7,7,7,7,15,11,11},
   {9,12,13,15,18,18,17,15},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20},
   {20,20,20,20,20,20,20,20}};

float out[8][8];
float sum,Cu,Cv;

   for (x=0;x<8;x++)
   {
      for (y=0;y<8;y++)
      {
         sum=0;
         for (u=0;u<8;u++)
            for (v=0;v<8;v++)
            {
               if (u==0) Cu=1/sqrt(2); else Cu=1;
               if (v==0) Cv=1/sqrt(2); else Cv=1;

               sum=sum+Cu*Cv*norm[u][v]*in[u][v]*cos(((2.0*x+1)*u*PI)/16.0)*
                  cos(((2.0*y+1)*v*PI)/16.0);
            }

         out[x][y]=1/4.0*sum;
         printf("%8.0f ",out[x][y]);
```

```
      }
      printf("\n");
   }
   printf("\n");
   return(0);
}
```

Sample run 8.3 shows that the values are similar to the input data values.

Sample run 8.3

```
 146   148   151   153   154   154   155   156
 148   150   153   154   155   155   155   156
 153   154   156   157   157   156   156   156
 157   158   159   159   158   157   156   156
 159   160   161   161   159   157   156   156
 160   161   162   162   160   158   157   156
 159   160   162   161   160   158   157   157
 158   160   161   161   160   158   157   157
```

The errors in the decoding are thus:

```
-2    -9    -2    2     -1    1     0     -1
3     1     -2    5     1     1     1     2
-2    2     4     5     2     -5    -5    -5
1     5     2     1     2     3     4     5
-1    0     0     1     1     -2    -1    0
1     0     -1    -1    0     -1    0     1
3     2     -1    -1    1     -1    0     0
4     2     0     -1    3     -1    1     -3
```

It can be seen that these errors are all less than 10, thus the decoding as not produced any significant errors.

8.4 JPEG file format

JPEG is a standard compression technique. A JPEG file normally complies with JFIF (JPEG file interchange format) which is a defined standard file format for storing a gray scale or YC_bC_r color image. Data within the JFIF contains segments separated by a 2-byte marker. This marker has a binary value of 1111 1111 (FFh) followed by a defined marker field. If a 1111 1111 (FFh) bit field occurs anywhere within the file (and it isn't a marker), the stuffed 0000 0000 (00h) byte is inserted after it so that it cannot be read as a false marker. The un-compression program must then discard the stuffed 00h byte.

Table 8.1 outlines some of the markers. For example, the code FFC0h the file is a base-line DCT frame with Huffman coding. Program 8.4 uses these codes to display the markers in a sample JPG file, and Sample run 8.1 show a sample run with a JPG file.

It can be seen that the markers in the test run are:

- Start of image (FFD8h). The segments can be organized in any order but the start-of-image marker is normally the first 2 bytes of the file.
- Application-specific type 0 (FFE0h). The JFIF header is placed after this marker.
- Define quantization table (FFDBh). Lists the quantization table(s).

- Baseline DCT, Huffman coding (FFC0h). Defines the type of coding used.
- Define Huffman table (FFC4h). Defines Huffman table(s).

8.4.1 JFIF header information

The header information of the JFIF file is contained after the application-specific type 0 marker (FFE0h). Figure 8.6 shows its format and Program 8.5 reads some of the header information from a JFIF file. It can be seen that in Sample run 8.5 the length of the segment is 4096 bytes and the file is a JFIF file, as it has the JFIF string at the correct location. The version in this case is 1.01 and the units are given in pixels per inch. Finally, it shows that the horizontal and vertical pixel density are both 11 265. For comparison, another sample run for a different file is shown in Sample run 8.6.

Table 8.1 Typical standard compressed graphics formats

Marker	*Description*	*Marker*	*Description*
C0h	Baseline DCT frame, Huffman coded	C1h	Baseline sequential DCT frame, Huffman coded
C2h	Extended sequential DCT frame, Huffman coded	C3h	Progressive DCT frame, Huffman coded
C4h	Define Huffman table	C5h	Differential sequential DCT frame, Huffman coded
C6h	Differential progressive DCT frame, arithmetic coded	C7h	Differential lossless frame, Huffman coded
C8h	Reserved	C9h	Extended sequential DCT frame, arithmetic coded
CAh	Progressive DCT frame, arithmetic coded	CBh	Lossless frame, arithmetic coded
CDh	Differential extended sequential DCT frame, arithmetic coded	CEh	Differential progressive DCT frame, arithmetic coded
CFh	Differential lossless frame, arithmetic	D8h	Start of image
D9h	End of image	E0h	Application-specific type 0

Program 8.4

```
#include    <stdio.h>

#define     NO_MARKS 19

int   main(void)
{
```

```
FILE  *in;
int   i,ch;
int   markers[NO_MARKS]={0xC0,0xC1,0xC2,0xC3,0xC4,
      0xC5,0xC6,0xC7,0xC8,0xC9,0xCA,0xCB,
      0xCD,0xCE,0xCF,0xD8,0xD9,0xDB,0xE0};
char  fname[BUFSIZ];
char  *msgs[NO_MARKS]={"Baseline DCT, Huff","Extended DCT, Huff",
      "Progress DCT, Huff","Lossless frame, Huff",
      "Define Huffman table", "Diff encoded DCT frame, Huff coded",
      "Diff progressive DCT frame, Huff", "Diff lossless frame, Huff",
      "Reserved", "Extended sequential DCT frame, arith coded",
      "Progressive DCT frame, arith coded",
      "Lossless frame, arith coded",
      "Diff extended sequential DCT frame, arith coding",
      "Diff progressive DCT frame, arith coding",
      "Diff lossless frame, arith coding",
      "Start of image", "End of image",
      "Define Quantization Tables", "Application specific type 0"};

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

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

   do
   {
      ch=getc(in);
      if (ch==0xff)
      {
         ch=getc(in);
         printf("%x",ch);
         for (i=0;i<NO_MARKS;i++)
            if (ch==markers[i])  printf("Found:%s\n",msgs[i]);
      }

   } while (!feof(in));

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

Sample run 8.4

```
Enter JPG file>> marble.jpg
d8:Found:Start of image
e0:Found:Application specific type 0
db:Found:Define Quantization Tables
db:Found:Define Quantization Tables
c0:Found:Baseline DCT, Huff
c4:Found:Define Huffman table
c4:Found:Define Huffman table
```

The reason that the segment is 4096 bytes is that a thumbnail version of the image is stored within the segment, after the basic header information defined in Figure 8.6.

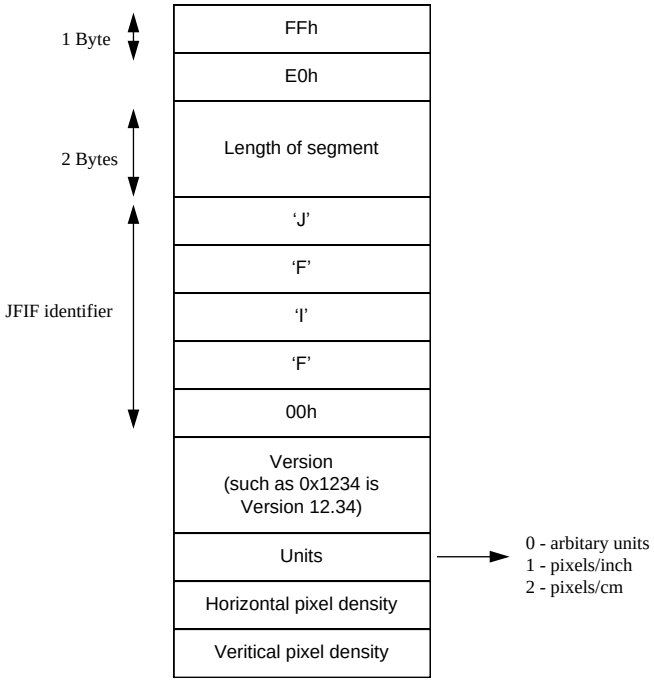

Figure 8.6 JFIF header information

Program 8.5

```
#include    <stdio.h>

int   main(void)
{
FILE  *in;
int   i,ch,version,length,units,pixelden_X,pixelden_Y;
char  fname[BUFSIZ],str[BUFSIZ];
char  *Units[3]={"Artibrary","Pixels per inch","Pixels per cm"};

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

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

   do
   {
      ch=getc(in);

      if (ch==0xff)
      {
         ch=getc(in);
         if (ch==0xe0)
         {
            fread(&length,2,1,in); printf("Length: %d\n",length);
            fread(str,5,1,in); printf("Marker: %s\n",str);
            fread(&version,2,1,in); printf("Version: %0x\n",version);
```

```
            fread(&units,1,1,in); printf("Units: %s\n",Units[units]);
            fread(&pixelden_X,2,1,in); printf("X den: %d\n",pixelden_X);
            fread(&pixelden_Y,2,1,in); printf("Y den: %d\n",pixelden_Y);
         }
      }

   } while (!feof(in));

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

Sample run 8.5

```
Enter JPG file>> marble.jpg
Length: 4096
Marker: JFIF
Version: 101
Units: Pixels per inch
X den: 11265
Y den: 11265
```

Sample run 8.6

```
Enter JPG file>> test.jpg
Length: 4096
Marker: JFIF
Version: 101
Units: Artibrary
X den: 256
Y den: 256
```

8.4.2 Quantization table

The quantization table is defined after the quantization table marker (FFDBh). Its format, after the marker, is:

- 2 bytes for the length of the segment.
- 1 byte, of which the high 4 bits define the precision (a 0 defines a table with 8-bit entries, a 1 defines 16-bit entries), and the low 4 bits give the table's ID (such as 0 for the first, 1 for the second, and so on).
- 64 entries for the table (either 8-bit or 16-bit entries). These entries are stored in a zigzag manner (see Figure 8.3).

Program 8.6 can be used to list the quantization table and Sample run 8.7 gives a sample run from a JPG file. It can be seen that in this case the factor varies from 3 to 28.

Program 8.6

```
#include    <stdio.h>
int   main(void)
{
FILE  *in;
int      i,ch,length;
char  fname[BUFSIZ], table, entry;

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

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

   do
   {
      ch=getc(in);

      if (ch==0xff)
      {
         ch=getc(in);
         if (ch==0xdb)
         {
            fread(&length,2,1,in); printf("Length: %d\n",length);
            fread(&table,1,1,in); printf("Marker: %x\n",table);

            for (i=0;i<64;i++)
            {
               fread(&entry,1,1,in);
               printf("%d ",entry);
            }
         }
      }

   } while (!feof(in));
   fclose(in);
   return(0);
}
```

Sample run 8.7

```
Enter JPG file>> test.jpg
Length: 17152
Marker: 0
5 3 4 4 4 3 5 4 4 4 5 5 5 6 7 12 8 7 7 7 7 15 11 11 9 12 17 15 18 18 17 15 17
17 19 22 28 23 19 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20 20
20 20 20 20 20
```

8.4.3 Huffman tables

Huffman tables are defined after the define Huffman table marker (FFC4h). One table defines the DC components, $F(0,0)$, and the other defines the AC components. Its format, after the marker, is:

- 2 bytes for the length of the segment.
- 1 byte, of which the high 4 bits defines the table class (0 for DC codes and 1 for AC codes), and the low 4 bits gives the table's ID (such as 0 for the first, 1 for the second, and so on).
- 16 bytes for the code lengths.
- A variable number of bytes which contain the Huffman codes (the code length defines the number of bits used for each code).

For example if the code lengths are:

3, 3, 3, 4, 4, 4

the packed Huffman code of:

000001010000100101000

would be separated as:

000 001 010 0001 0010 1000

8.5 JPEG modes

JPEG has three main modes of compression:

- Progressive mode – this mode allows the viewing of a rough outline of an image while decoding the rest of the file. It is useful when an image is being received over a relatively slow transfer channel (such as over a modem or from the Internet). There are two main methods used: spectral-selection mode and successive-approximation mode. The successive-approximation mode first sends the high-order bits of each of the encoded values and then the lower-order lower bits. Spectral-selection mode first sends the low-frequency components of each of the 8×8 blocks then sends the high-frequency terms.
- Hierarchical mode – in this mode the image is stored in increasing resolution. For example a 1280×960 image might be stored as 160×120, 320×240, 640×480 and 1280×960. The viewing program can then show the image in increasing resolution as it reads (or receives) the file. Most systems do not implement this facility.
- Lossless mode – this mode allows data to be stored and recovered in exactly in its original state. It does not use DCT conversion or subsampling.