Image Compression Using JPEG Guidelines

Project Report

Submitted by

Arshad Y. Imani
Sree Rangarajan .S
Rajesh Rao .R

Under the Guidance of

Mr. D. Ramesh, M.E.,

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

BACHELOR OF ENGINEERING IN

COMPUTER SCIENCE AND ENGINEERING

OF THE BHARATHIAR UNIVERSITY, COIMBATORE

1995-96

Department of Computer Science and Engineering

Kumaraguru College of Technology

Coimbatore-641 006

Kumaraguru College of Technology

Coimbatore - 641 006.

Department of Computer Science and Engineering

Certificate

This is to Certify that the Report entitled

"IMAGE COMPRESSION USING JPEG GUIDELINES"

has been submitted by

Mr. ARSHAD, SREE RANGARAJAN.S, RAJESH RAO.R

in partial fulfilment of the requirements for the award of Degree of Bachelor of Engineering in the Computer Science and Engineering Branch of the Bharathiar University, Coimbatore - 641 046 during the academic year 1995-'96.

[illegible] 12/4/96

(Guide) (Head of Department)

Certified that the Candidate was Examined by us in the Project Work Viva-Voce Examination held on 12-4-1996 and the University Register Number is ________________

(Internal Examiner) (External Examiner)

ACKNOWLEDGEMENT

The authors wish to place on record their heartfelt gratitude to the Principal **Dr. S. Subramanian**, for his kind permission to avail the facilities that enabled the successful completion of this project.

They profoundly thank **Prof. P. Shanmugam**, M.Sc(Engg)., M.S(Hawaii)., M.I.E.E.E., M.I.C.S.I., M.I.S.T.E., Head of the Department of Computer Science and Engineering for all his suggestions and constant encouragement.

The authors acknowledge the able guidance, valuable suggestions, indepth involvement and timely advice offered by their dynamic guide **Mr. D. Ramesh**, M.E.,

They wish to place on record their appreciation to the amiable Faculty members and the non-teaching staff.

Finally, they remain indebted to their parents for their unfailing love, inspiration and compassion at all stressful moments.

CONTENTS

SYNOPSIS

Image data compression is concerned with minimizing the number of bits required to represent an image. Typical television images have spatial resolution of approximately 512 x 512 pixels per frame. At 8 bits per pixel per color channel and 30 frames per second, this translates into a rate of nearly 180 x 10^6 bits per second. Depending on the application, digital image raw data rates vary from 10^5 bits per frame to 10^8 bits per frame. The large channel and memory requirements for digital transmission and storage make it desirable to consider image compression techniques.

JPEG, the Joint Photographers Expert Group, an ISO/CCITT - backed international standards committee has defined an image compression specification for still images. The BMP(BitMap File Format) has been extended to encompass JPEG compressed images.

JPEG achieves image compression by methodically throwing away visually insignificant image information. This information includes high frequency components of an image which are less important to the image content than the low frequency components. These high frequency components cannot be retrieved on decompression and hence this type of compression is termed to be lossy.

During encoding, an image is broken up into 8 x 8 pixel blocks that are processed individually, from left to right and top to bottom. Each block of image data is subject to a forward discrete cosine transform which converts pixel values into their corresponding frequency components. The frequency component coefficients are resequenced(zigzaged) in order of increasing frequency.

The resultant frequency coefficients are then quantized which causes components with near zero amplitudes to become zero. The entropy encoding process performs 2 types of image compression on the block of frequency coefficients. Firstly, it run length encodes the number of zero coefficient values using statistically generated tables. The encoded bit stream is written to the output stream. During decoding the reverse process is carried out.

The above process has been implemeted using 'C' with Visual Basic as the front end tool.

INTRODUCTION

1.1 IMAGE COMPRESSION

Consider an image with 512 x 512 pixel spatial resolution and 8 bits (256 levels) intensity resolution. This represents 0.25 Mbyte of image data. This amount of data over time represents a data rate of almost 63 million bits per second. Image compression seeks to reduce this data rate.

The heart of image compression techniques center on two entities :
1) The development of an image representation that removes a significant amount of the inherent redundancy in the image data i.e., a transformation of the image data such that the tranformed image consists, ideally of uncorrelated data.
2) The achievement of a reconstruction scheme that undoes the compression or encoding scheme. This reconstruction scheme together with the chosen computer technique is adopted to minimize subjective distortion in the resulting image.

1.2 NEED FOR IMAGE COMPRESSION

Image compression enables efficient storage and transmission of images. A large amount of memory is occupied when a number of images are stored. With the capacities of storage media substantial and the cost of storage media on the rise, it becomes essential to store images in as minimum amount of space as possible. Now, image compression comes into play i.e., the original image occupies almost half the space once it has been compressed.

Also, in some image processing applications, it would be necessary to transmit several images over a communication channel. A typical example is the transmission of satellite pictures of space to substations on earth. The communication channel, in this case, is a very long one with applicable delay. The image representation, now requires more number of bits since channel capacity is limited. Image data compression algorithms limit the file size thus enabling more number of images to be transmitted within allotted time.

1.3 PROBLEM AND ITS OBJECTIVE

The problem is concerned with the compression of still images. The BMP files have been considered for compression.
The objective of this project is to develop an image compression algorithm according to the specification of Joint Photographers Expert Group (JPEG) and thereby achieve a good compression ratio.

1.4 SOLUTION DETAILS

Compression of an image reduces the number of bits required for its storage. The image is divided into 8 x 8 pixel blocks. A discrete fourier cosine tranform is carried out on the pixel block which prepares the matrix for the quantization step. Zigzag sequencing is then performed on the image pixel array. Quantization which discards the visually insignificant details of the image is the next step. Finally, entropy encoding is carried out to encode the values obtained. Decompression is the reverse process of compression.

The algorithm has been implemented in 'C' with Visual Basic as the front end tool.

IMAGE COMPRESSION

2.1 INTRODUCTION

Image compression basically consists of taking a stream of symbols and transforming them into codes. If the compression is effective, the resulting stream of codes will be smaller than the original symbols.

There are many algorithms developed for image data compression applications. The algorithms are based on principles of redundancy, reduction or decorrelation of the image specified. Based on these principles, there are two different techniques for image compression. They are the spatial coding and the transform coding techniques.

2.2 TECHNIQUES FOR IMAGE COMPRESSION

2.2.1 Spatial coding method

Here an array of uncorrelated random variables are generated from the given image using an invertible transform. The uncorrelated random variables can be represented using minimum number of bits. During decompression, the inverse transformation is carried out.

The structure of transformation is defined by a process which is used to model the image. An image model is an orderly representation of an image. The model is described by a set of parameters. These parameters can be represented using fewer number of bits giving rise to compression.

2.2.2 Transform coding method

In this method, a linear invertible two dimensional transform is applied to the given image. The transformed coefficients are analyzed and a subset of them are retained, quantized and then stored or transmitted. The coefficients can be represented by less number of bits since most of the coefficients are discarded. In the decompression process, the transform coefficients are reconstructed by applying the inverse transform. This results in an image which is quite close to the original image.

Karhunen - Loeve transform is the best invertible transform. This transform when applied results in a very high compression ratio. But it is compute intensive. Most of the transform coding algorithms use sub - optimal transforms such as discrete cosine transform, Walsh transform etc. These transforms when applied do not result in compute intensive operations but the compression ratio obtained is lower.

2.3 JPEG AND ITS SPECIFICATIONS

JPEG, the Joint Photographers Expert Group, an ISO/CCITT - backed international standards committee has specified some image compression specifications for still images. The standard stipulates the following

* Compression algorithms should be portable i.e., they should allow for software implementations on a wide variety of systems.
* Algorithms should operate at or near the image compression rates.
* Compressed images must be good to excellent in quality.
* Compression ratios should be user variable.
* Compression should be applicable to all sorts of still images.
* Interoperability should exist between implementations from different vendors.

2.4 MODES OF OPERATION

The JPEG specification defines four different modes of operation of JPEG software. They are

1) Sequential coding method : Here the images are coded in a top to bottom, left to right fashion.

2) Progressive encoding method : Here the images are compressed such that on decoding, they paint an image that is successively refined for each decoded scan. This is similar to interleaved GIF images.

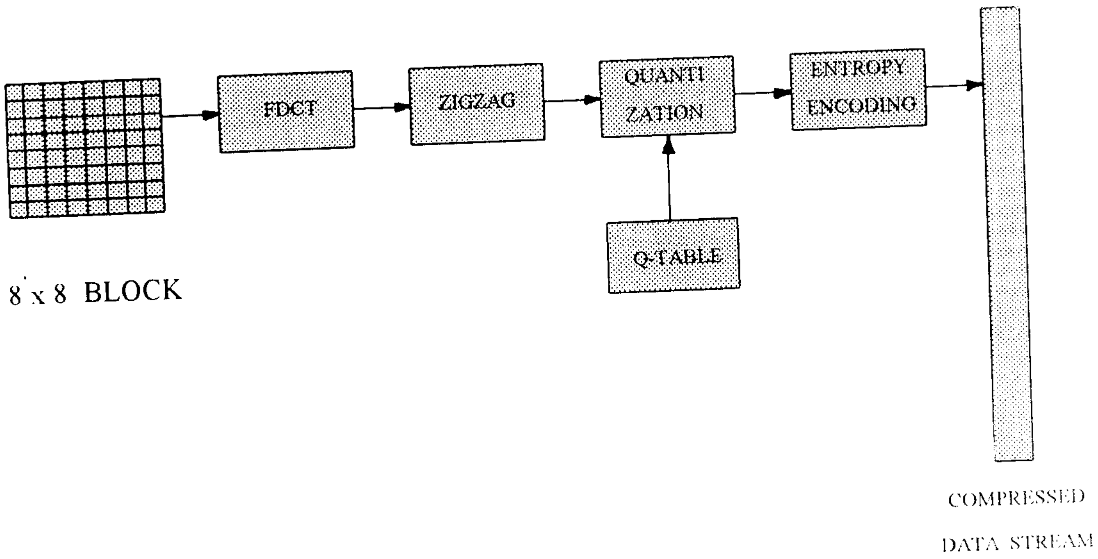
3) Lossless encoding method : Here the image is guaranteed to be a bit-for-bit copy of the original image. Lossless encoding cannot achieve the compression ratios of normal lossy compression methods and so it is rarely implemented.

4) Hierarchical encoding method : Multiple copies of the image are

encoded together with each copy bearing a different resolution. This allows an application to decode an image of the specific resolution it requires without having to handle images of too high or too low resolution.

A baseline sequential mode of operation has been implemented. Compression is achieved by discarding visually insignificant image information. High frequency components contain such information and so they are less important to the image content when compared to the low frequency components. These high frequency components cannot be retrieved once discarded. Hence the name lossy compression. When a compressed image is displayed, much of the high frequency information is not present. The image is not a bit-for-bit copy of the original image. There is usually a trade - off between image quality and compression ratio. Lossy compression produces a high compression ratio and an image quality that is good.

2.5 JPEG ENCODING PIPELINE



During encoding an image is broken up into 8 x 8 pixel blocks that are processed individually form left to right and top to bottom. Each pixel information is read using the BMP(BitMap File Format). This format makes it possible to store the scanned images of photographs. The 8 x 8 block is now subjected to a Forward Discrete Cosine Transform.

2.5.1 Discrete Cosine Transform

DCT's convert 2D pixel amplitude and spatial information into frequency content for subsequent manipulation i.e., a DCT is used to calculate the frequency components of a given signal sampled at a given sampling rate. The DCT's are calculated by applying a series of

9-241

weighting coefficients to the pixel data.

FDCT

$$F(u,v) = \tfrac{1}{4} C_u C_v \sum_{u=0}^{7} \sum_{v=0}^{7} f(i,j) \{ \cos[(2i+1)\pi u / 16] \,.\, \cos[(2j+1)\pi v / 16] \}$$

where $u,v = 0,1,\ldots,7$

$C_l = 1/\sqrt{2}\,,\ l = 0 \qquad l = u,v.$

$\quad = 1 \qquad ,\ l \neq 0$

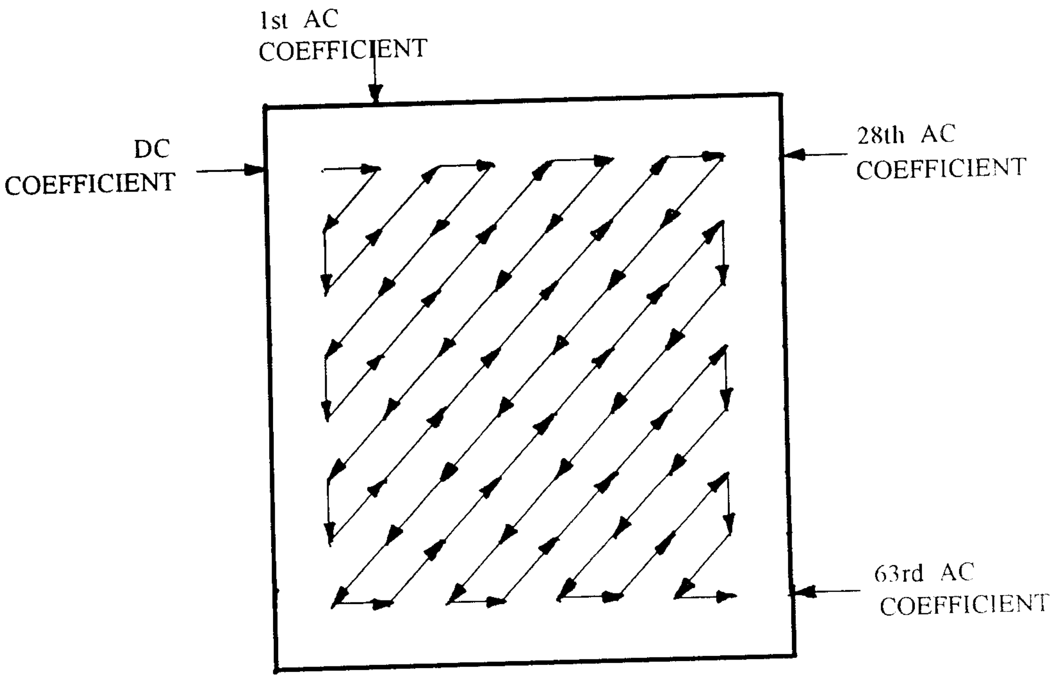
The above equation describes the 2D FDCT process for 8 x 8 pixel blocks. The transcendental functions cannot be computed with perfect accuracy. If the FDCT could be computed with perfect accuracy, the exact pixel data fed into the FDCT could be recovered when the transformed data is returned from IDCT.

The FDCT when applied to an 8 x 8 block of pixel data results in a series of 64 frequency coefficients. The first is a 'DC coefficient' and it is the average of all pixel values within the block. The other 63 'AC coefficients' contain the spectrum of frequency components that make up the block of image data. Continuous tone images have a slowly changing nature and this causes many high - frequency coefficients produced by the FDCT to be zero or close to zero. This is exploited in the quantization process. Speed of the DCT is a factor which determines the image compression performance.

DCT is a lossless step in the compression algorithm. The DCT step prepares the matrix for the quantization step where the compression occurs.

2.5.2The ZigZag Sequence

Once the block of image pixels have been processed with the FDCT, the result is a frequency spectrum of pixels. The coefficients that make up the frequency spectrum obtained from FDCT are not arranged in ascending order. They are usually clustered with the DC coefficient at the upper left surrounded by the lower frequency components. The higher frequency components are grouped towards the lower right. This is indicated in the following diagram



8 x 8 Block Of Integers

The application of zigzag sequencing after FDCT converts the 64 frequency coefficients into ascending order, as required for further processing. When the frequency coefficients are in ascending order, the DC and low frequency coefficients are grouped together followed by the high frequency coefficients.

The zigzag sequencing of the elements in the matrix improves the efficiency in calculation when compared to the other sequencing methods.

2.5.3 Quantization

The DCT output matrix occupies more space than the original matrix of pixels. The input to the DCT consists of 8 bits/pixel whereas the values obtained after DCT ranges from a low of -1024 to a high of 1023 i.e around 11 bits/pixel. Quantization is the step where most image compression takes place and is therefore the principle source of loss in image compression. Once a DCT image has been compressed, it is possible to reduce the precision of the coefficients more by moving away from the DC coefficient at the origin. The farther the element from the position (0,0), the less the element contributes to the graphical image.

In accordance with JPEG guidelines, quantization is implemented using a quantization matrix. For every element position in the DCT matrix, a corresponding element in the quantization matrix gives a quantum value. The quantum value indicates the step size for each element in the compressed rendition of the picture, with values ranging from one to 255.

The elements that matter the most are encoded using a smaller step size, size 1 offering the most precision. Values away from the origin are high. The formula for quantization is given by

$$\text{Quantized Value } (i,j) = \frac{\text{DCT } (i,j)}{\text{Quantum Value } (i,j)}$$

From the formula, it can be noted that quantization values above 25 or 50 assure that virtually all the high frequency components will be rounded down to zero. Only if the high frequency components go upto unusually large values will then be encoded as non-zero components.

There are atleast two experimental approaches of selecting a quantization matrix. One measures the mathematical error found between an input image and output image after it has been decompressed. The other method tries to judge the decompression on the eye. This approach does not usually correspond to the differences in error levels.

Quantization is implemented using the Human Visual Sensitivity(HVS) approach. The quatization matrix or luminance matrix for a quality factor 2 is represented by the matrix given below.

$$\begin{bmatrix} 3 & 5 & 7 & 9 & 11 & 13 & 15 & 17 \\ 5 & 7 & 9 & 11 & 13 & 15 & 17 & 19 \\ 7 & 9 & 11 & 13 & 15 & 17 & 19 & 21 \\ 9 & 11 & 13 & 15 & 17 & 19 & 21 & 23 \\ 11 & 13 & 15 & 17 & 19 & 21 & 23 & 25 \\ 13 & 15 & 17 & 19 & 21 & 23 & 25 & 27 \\ 15 & 17 & 19 & 21 & 23 & 25 & 27 & 29 \\ 17 & 19 & 21 & 23 & 25 & 27 & 29 & 31 \end{bmatrix}$$

Since the quantization matrix can be defined at runtime when compression takes place, JPEG allows the use of any quantization matrix. This enables the user to decide the quality of the picture during compression. By choosing extraordinarily high step sizes for most DCT coefficients, it is possible to obtain excellent compression ratios but only a poor image quality. Low step sizes result in less compression ratios and excellent image quality. This allows for a great deal of flexibility as it is possible to choose the image quality based on both the imaging requirements and the storage capacity.

The quantum step sizes are determined from the quality factor which usually ranges from 1-25. Values larger than 25 would work, but picture quality has degraded enough at a quality level of 25 to make going any further an exercise in futility.

The threshold value of an element is set before it contributes any meaningful information to the picture. Any contribution under the value of the threshold is simply thrown out. This is the point in the algorithm when lossy effect is said to take place.

2.5.4 Entropy Encoding

This is the final step in the encoding process. Arithmetic or Huffman encoding may be used. Arithmetic encoding performs better for some images but is very complex to implement. Huffman encoding provides additional lossless compression for the already highly processed image data. Huffman compression is based on the statistical characteristics of the data to be compressed. Symbols which occur frequently are assigned shorter Huffman codes and those that occur infrequently are assigned longer codes. Compression occurs as long as there is a large difference between the occurence counts of the most common and least common symbols. This type of coding is bit oriented. The Huffman codes assigned to the various symbols are bit packed together into the tightest possible configuration of bytes.

uring the enoding process, the frequency coefficients for a block of nage data are bit encoded as a variable length Huffman code. Two dditional forms of data compression can occur in the entropy coding ep.

) Delta coding of the DC coefficients of adjacent blocks of image ata.

) Run length encoding of zero-valued frequency coefficients.

hese compression mechanisms contribute to the overall compression chieved for compressed images.

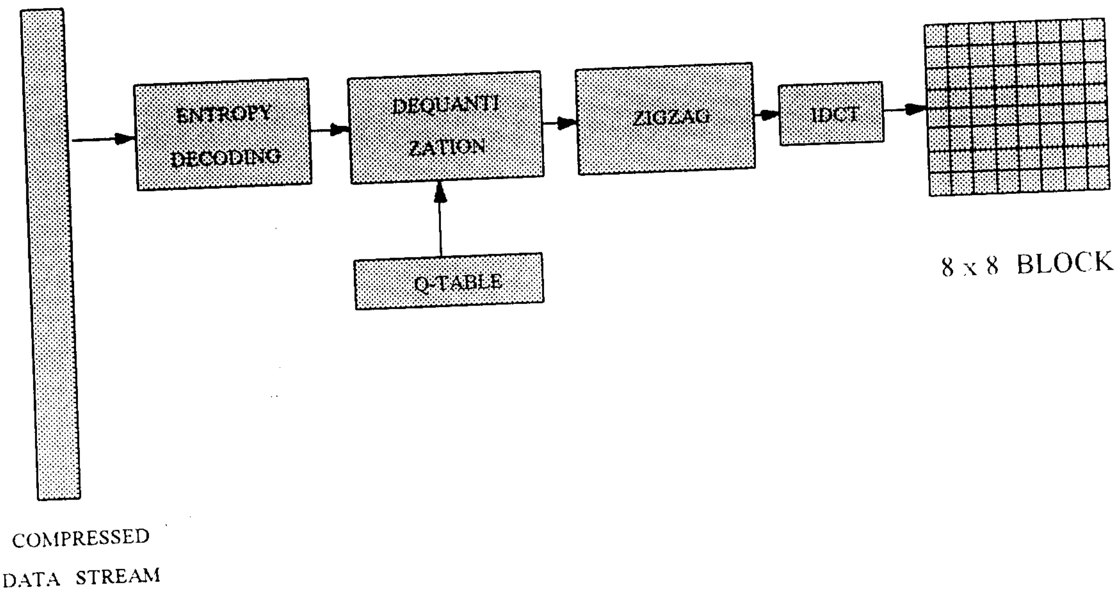
here is usually a high level of correlation between the DC coefficients f adjacent blocks of image data and the values of the DC coefficients re generally large. Hence significant compression can be achieved by ncoding the difference values in the DC coefficients between adjacent locks rather than the actual values.

un length encoding of zero values produces significant compression. uantization causes many high frequency components in a block of mage to become zero. Significant compression can be achieved by ounting the number of zero coefficients in the block preceding a non-ero coefficient and encoding that number along with the non-zero oefficient. Two codes are present for this purpose. They are

) Code 0xF0 called zero run length (ZRL) used to signal when there are 16 zero coefficients in a row.

2) Code 0x00 called End of block (EOB) which signifies that all of the remaining coefficients in a block are zero and so no further processing of the block is necessary.

2.6 JPEG DECODING PIPELINE



Entropy decoding

Entropy decoding is just the reverse process of entropy encoding. Run length encoding is undone and the encoded zero values i.e., the lost information is brought back into the 8 x 8 pixel block.

Dequantization

During quantization, the image data is divided by the values in the quantization table, but during decoding, image data must be dequantized. Dequantization is performed by multiplying the decoded image data by the value in the quantization table, thereby restoring it to a value close to the prequantization value. The formula used is given as follows

$$\text{DCT }(i,j) = \text{Quantized Value }(i,j) \times \text{Quantum Value }(i,j)$$

Again, it can be noticed that with large quantum values it is possible to run into the risk of generating large errors. But errors occuring in the high frequency components during dequantization do not have a serious effect on image quality.

Inverse Discrete Cosine Transform

The formula for IDCT is given by

$$f(i,j) = \tfrac{1}{4} \sum_{i=0}^{7} \sum_{j=0}^{7} C_u\, C_v\, F(u,v)\ \{\ [\cos (2i+1)\, \pi u / 16]\ .\ [\cos (2j+1)\, \pi v / 16\ ]\ \}$$

where $i,j = 0,1,\ldots,7$

$C_l = 1 / \sqrt{2}\ , l = 0 \qquad l = u,v$

$= 1 \qquad , l \neq 0$

The DCT cannot be computed with perfect accuracy. If the DCT's could be computed with perfect accuracy, the exact pixel data fed can be recovered.

IMPLEMENTATION

3.1 MINIMUM REQUIREMENTS

The algorithm requires a minimum of an 80486 processor with 4 Mbyte RAM. An image viewer to view the images after decompression is also required.

3.2 LANGUAGE USED

The language used is ‘C’. It has been used owing to its easiness in programming and its portability. The pseudocode is easy to read. This language is a language without too many surprises. The few constructs which it uses as basic language elements can easily be translated into other languages. Efficiency is another feather in C’s cap.

3.3 BMP FILE FORMAT

The BMP file extension indicates a BitMap file format popularly used in WINDOWS 3.1 onwards. Each file begins with a file header consisting of 14 bytes followed by header information of 64 bytes. It is then followed by color scheme information which varies from file to file, depending on the type of BMP file used.

Bitmap file header :

BYTE	CONTENTS
1-2	Type
3-6	Size
7-8	x-hotspot
9-10	y-hotspot
11-14	Offset

Bitmap header :

15-18	Size
19-22	Width
23-26	Height
27-28	Number of bit planes
29-30	Number of bits per plane
31-34	Compression scheme
35-38	Size of image data
39-42	X-resolution
43-46	Y-resolution
47-50	Number of colors used
51-54	Number of important colors
55-56	Resolution units
57-58	Padding
59-60	Origin
61-62	Half-toning

63-66	Half-toning parameter1
67-70	Half-toning parameter2
71-74	Color encoding
75-78	Identifier

3.4 DATA STRUCTURES

Arrays and pointers have been used. Static storage allocation has been used in processing of pixel blocks and run time storage allocation has been used in giving input and output to the files concerned. Two structures have been used to store the header information. They can be defined as follows

```
struct
{
   uint16  type;
   uint32  size;
    int16   xhs;
    int16   yhs;
    uint32  offset;
 } bmpfileheader;
```

```
struct
{
    uint32  size;
    int32   width;
    int32   height;
    uint16  numbitplanes;
    uint16  numbitsperplane;
    uint32  compressionscheme;
    uint32  sizeofimagedata;
    uint32  xresolution;
    uint32  yresolution;
    uint32  numcolorused;
    uint16  resolutionunits;
    uint16  padding;
    uint16  origin;
    uint16  halftoning;
    uint32  halftoningparam1;
    uint32  halftoningparam2;
    uint32  colorencoding;
    uint32  identifier;
} bmpheader;
```

A structure has been used for sequencing the pixel block elements in zigzag fashion. It is given by

```
struct zigzag
{
   int row;
   int col;
 } Zigzag[N * N] =
 {
    {0,0},
    {0,1},{1,0},
    {2,0},{1,1},{0,2},
      .
      .
      .
     {7,5},{6,6},{5,7},
     {6,7},{7,6},
     {7,7}
   };
```

3.5 IMAGES DEALT

The images dealt in this project work are uncompressed BMP images of any resolution. Images are divided into 8 x 8 pixel blocks which are processed at a time.

3.6 PROGRAM IMPLEMENTATION

There are two main modules used. They are

1) Compression

2) Decompression

Compression

The following functions are used in the compression algorithm

a) Initialization : The cosine matrix c[N][N], cosine transform matrix ct[N][N] and the quatization matrix quantum[N][N] are initialized based on the quality factor determined by the user.

b) Forward discrete cosine tranform : The cosine and cosine transform matrices are multiplied with the 8 x 8 pixel block. This function causes values in the pixel block to vary from -1024 to 1023. This is a lossless step in the image compression algorithm and it prepares the matrix for the quantization process.

c) Quantization : This is the lossy step in the image compression algorithm where the values after FDCT are reduced to a range of about (-128,127). Most of the visually insignificant image information is lost here.

d) Encoding : The number of zeroes in the matrix after quantzation are counted and encoded. Two codes called zero run length and end of block have been used for this purpose.

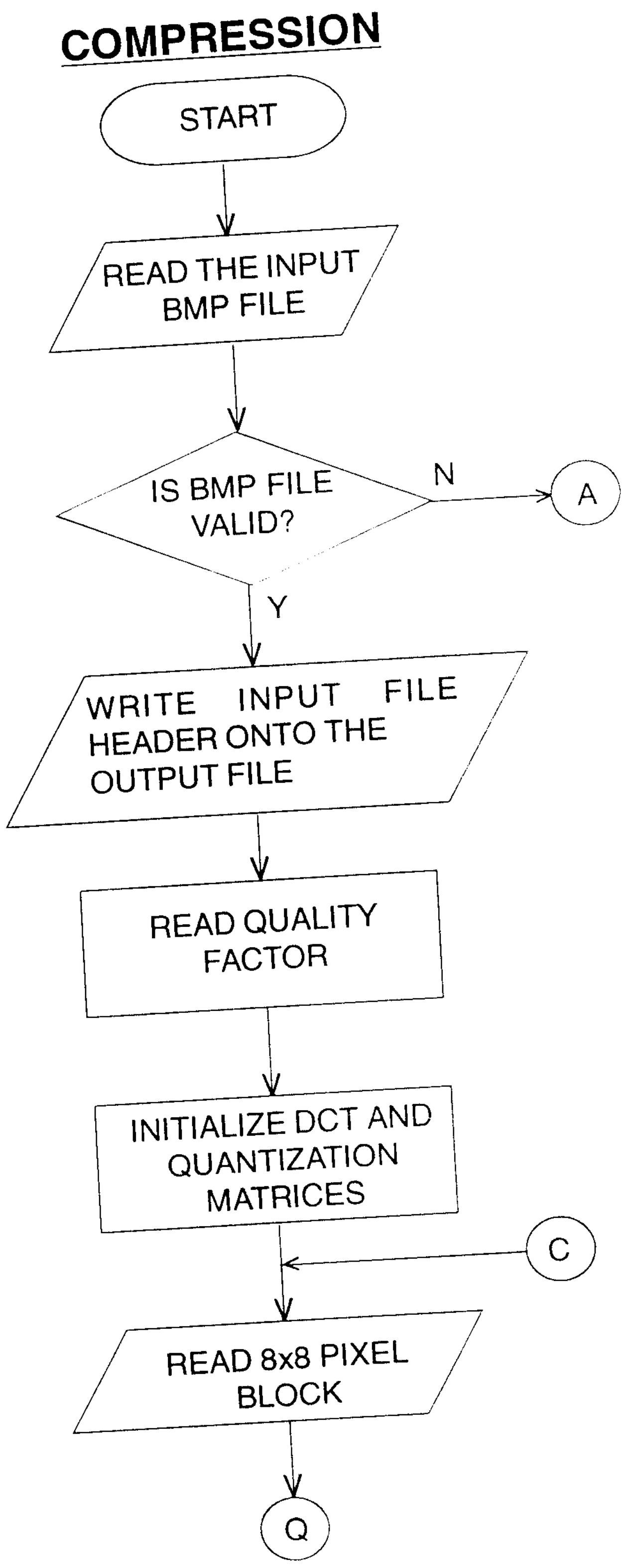
After the above steps have been performed, the values are written into a file with extension BAR.

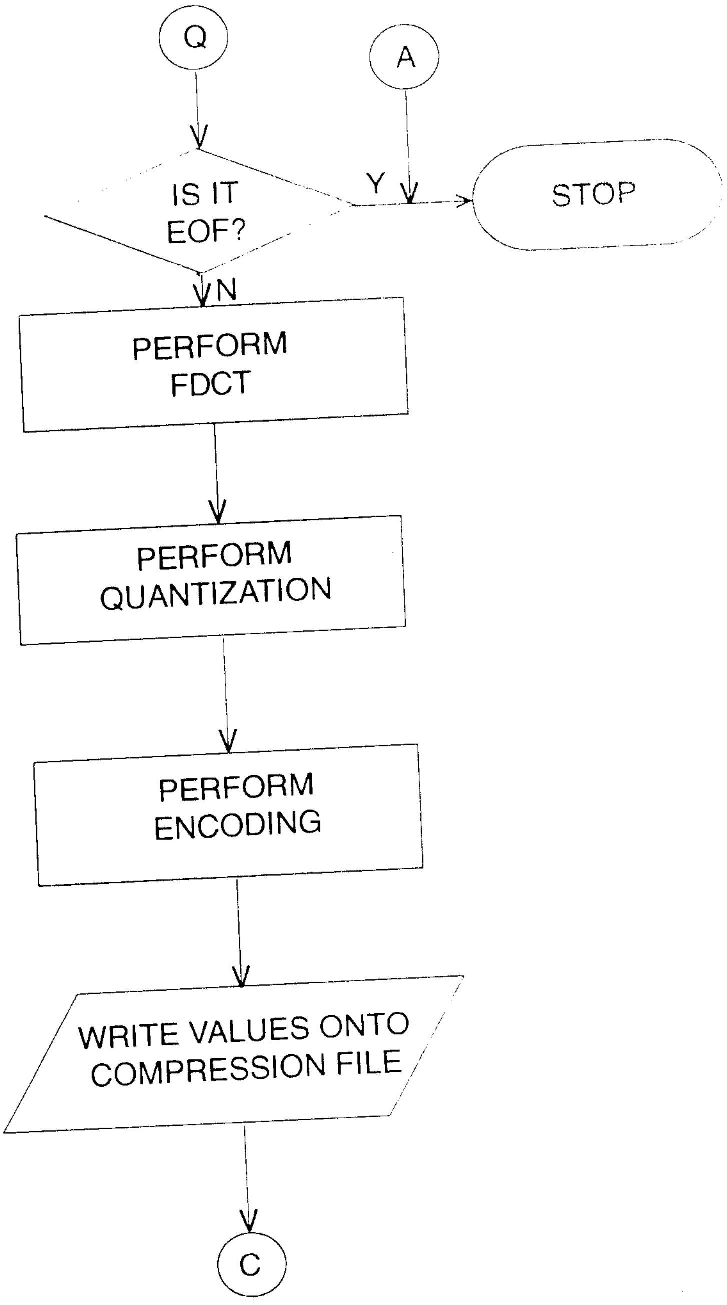
Decompression

The main functions used are
a) Entropy decoding : In this function run length decoding is performed where the values made zero in the compression process are written back into the program.
b) Dequantization : The quantum value is multiplied with the quantized value to obtain the DCT value. This process is just the reverse of quantization.
c) Inverse Discrete Cosine tansform : The values obtained after dequantization are subjected to IDCT inorder to obtain the original pixel value. This process undoes the effect of FDCT performed during compression.

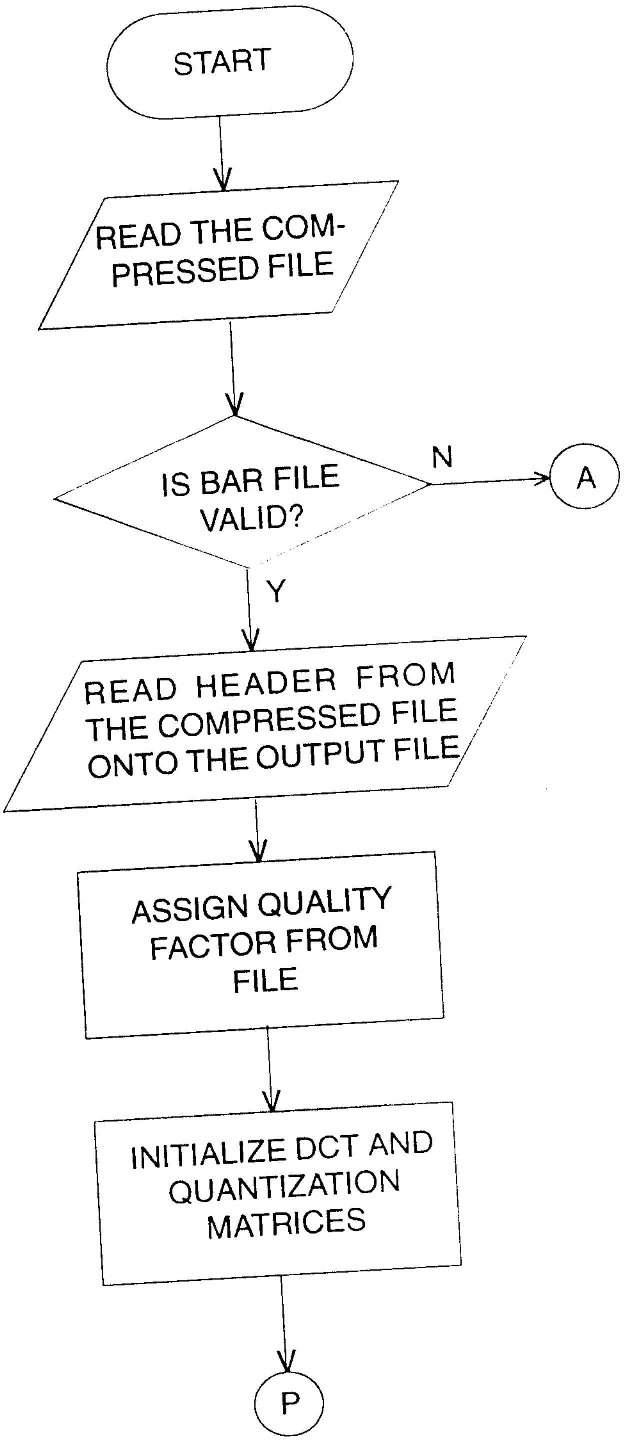
After decompression has been carried out, the resultant image values are written into a file with extension BMA.

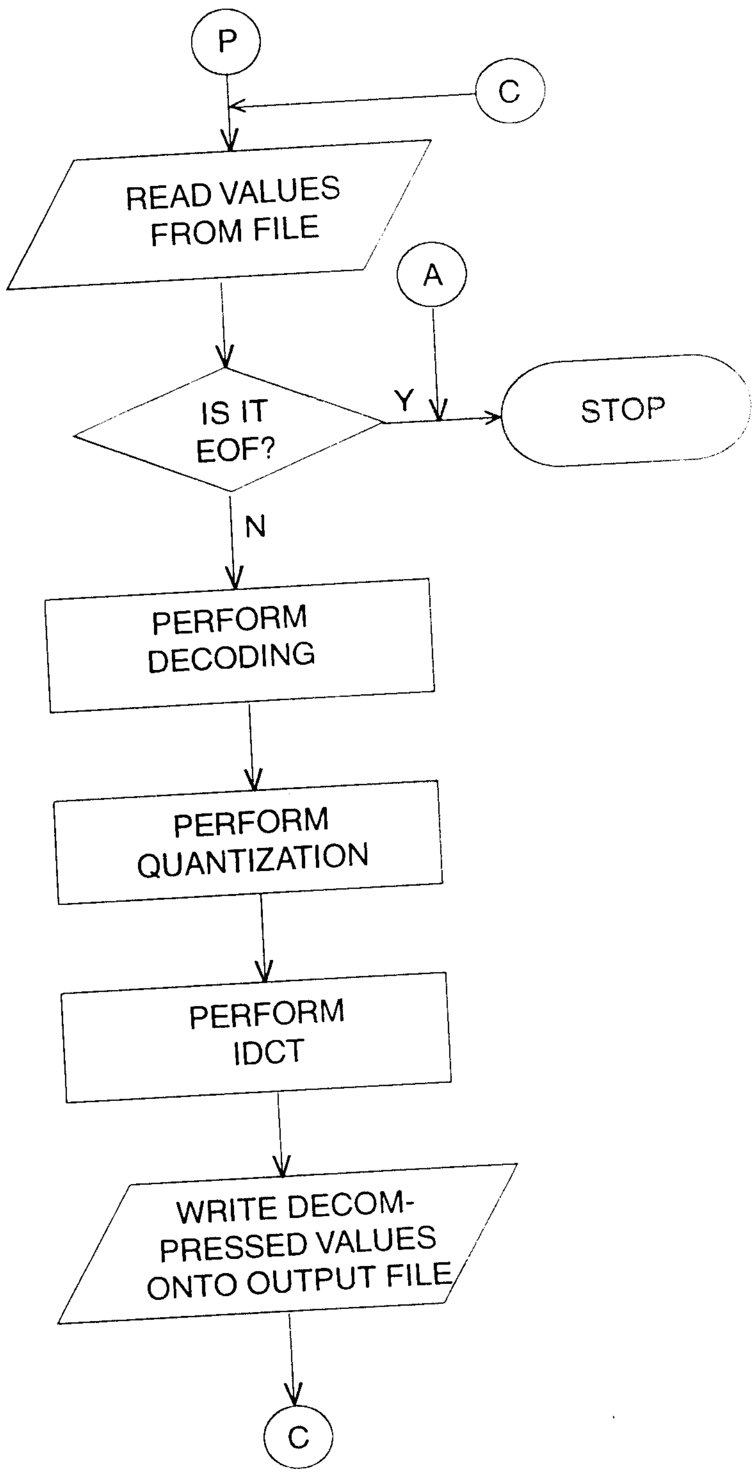
7 FLOWCHART


COMPRESSION
START
READ THE INPUT BMP FILE
IS BMP FILE VALID?
N
A
Y
WRITE INPUT FILE HEADER ONTO THE OUTPUT FILE
READ QUALITY FACTOR
INITIALIZE DCT AND QUANTIZATION MATRICES
C
READ 8x8 PIXEL BLOCK
Q


Q
A
IS IT EOF?
Y
N
STOP
PERFORM FDCT
PERFORM QUANTIZATION
PERFORM ENCODING
WRITE VALUES ONTO COMPRESSION FILE
C

DECOMPRESSION


START
READ THE COM-PRESSED FILE
IS BAR FILE VALID?
N
A
Y
READ HEADER FROM THE COMPRESSED FILE ONTO THE OUTPUT FILE
ASSIGN QUALITY FACTOR FROM FILE
INITIALIZE DCT AND QUANTIZATION MATRICES
P


P
C
READ VALUES FROM FILE
A
IS IT EOF?
Y
STOP
N
PERFORM DECODING
PERFORM QUANTIZATION
PERFORM IDCT
WRITE DECOM-PRESSED VALUES ONTO OUTPUT FILE
C

Results & Conclusion

RESULTS AND CONCLUSION

The performance of the above algorithm which has been implemented can be given in terms of two performance measures namely the
1)compression ratio
2)image quality

Excellent image compression ratios have been achieved by implementing this algorithm. The image quality obtained after the decompression process has been one factor to reckon this algorithm with. For a quality factor of 1 - 25 good image quality has been obtained.

The main advantage of this algorithm lies in the fact that
1) Compression ratio is user defined,
2) Good image quality,
3) Easy implementation of the algorithm.

FUTURE SCOPE

This algorithm has been implemented for BMP file formats, it can be further extended to encompass various other file formats.

APPLICATIONS

Image Compression finds a lot of applications such as digital video cameras, color video printers, FAX machines, desktop publishing, copiers, scanners, storage/transmission, multimedia and printers, medical image data compression, remotely piloted vehicles and broadcast television.

Medical Image Data Compression

Despite the high cost of computer tomography systems, they are being used in many places throughout the world. Physicians would like to store the X-ray images and maintain an image database for each patient. Physicians may be concerned that the data compression might introduce distortion that would remove important information or add spurious information, either of which could lead to incorrect diagnosis. Consequently, low error techniques should be considered.

Video conferencing

Unlike broadcast television, there is rarely a large amount of movement in the pictures used for video conferencing. This allows conditional replenishment coding to be used effectively.

Remotely piloted vehicles

Remotely piloted vehicles are small aircrafts that are used by the military in reconnaissance operations in a military environment. Transmission here must be protected from jamming by the enemy. Hence compression again comes into play.

REFERENCES:

1) JPEG like Image Compression - Dr. Dobb's Journal, July 1995.

2) Algorithms for manipulating compressed images - IEEE Computer Graphics and Applications.

3) File Format Handbook by Allen Taylor.

4) Digital Image Processing and Computer Vision by Robert. J. Schalkoff.

5) Fundamentals of digital image processing by Anil. K. Jain.

6) Image Processing in 'C' by Dwayne Philips.

```
void Init(int qty);
void Read_input_file(FILE *input,unsigned char **dat);
void FDCT(unsigned char *input[N],int output[N][N]);
void Quantization(BIT_FILE *Out_File,int out_dat[N][N]);
void OutputCode(BIT_FILE *Out_File,int code);
void valid_bmp(FILE *);
void write_bmp(BIT_FILE *,FILE *);

unsigned char **pix_value;
double C[N][N];
double Ct[N][N];
int ORL;
int Quantum[N][N];
int ROWS,COLS;
long tmp;

struct zigzag {
                int row;
                int col;
             }
 ZigZag[N*N] =
 /* Optimal coding sequence to obtain consequent zeroes */
 {
     {0,0},
     {0,1}, {1,0},
     {2,0}, {1,1}, {0,2},
     {0,3}, {1,2}, {2,1}, {3,0},
     {4,0}, {3,1}, {2,2}, {1,3}, {0,4},
     {0,5}, {1,4}, {2,3}, {3,2}, {4,1}, {5,0},
     {6,0}, {5,1}, {4,2}, {3,3}, {2,4}, {1,5}, {0,6},
     {0,7}, {1,6}, {2,5}, {3,4}, {4,3}, {5,2}, {6,1}, {7,0},
     {7,1}, {6,2}, {5,3}, {4,4}, {3,5}, {2,6}, {1,7},
     {2,7}, {3,6}, {4,5}, {5,4}, {6,3}, {7,2},
     {7,3}, {6,4}, {5,5}, {4,6}, {3,7},
     {4,7}, {5,6}, {6,5}, {7,4},
     {7,5}, {6,6}, {5,7},
     {6,7}, {7,6},
     {7,7}
  };

  /* opening a compression bit file */
  BIT_FILE *OpenOutputBitFile(char *name)
  {
     BIT_FILE *bit_file;
     bit_file = (BIT_FILE *)calloc(1,sizeof(BIT_FILE));
     if(bit_file == NULL)
       return(bit_file);
     bit_file -> file                = fopen(name,"wb");
     bit_file -> cur_byte            = 0;
     bit_file -> mask                = 0x80;
     return(bit_file);
  }

  /* closing the compression bit file */
  void CloseOutputBitFile(BIT_FILE *bit_file)
  {
       if(bit_file->mask != 0x80)
```

```
        printf("Error In Closing");
        exit(0);
    }
    fclose(bit_file->file);
    free((char *)bit_file);
}
/* writing onto compression file */
void OutputBits(BIT_FILE *bit_file,unsigned long code,int count)
{
    unsigned long mask;

    mask = 1L << (count-1);
    while(mask != 0)
    {
        if(mask & code)
        bit_file->cur_byte |= bit_file->mask;
        bit_file->mask >>= 1;
        if(bit_file->mask == 0)
        {
            if(putc(bit_file->cur_byte,bit_file->file) !=
            bit_file->cur_byte)
            {
                printf("Error In OutputBits\n");
                exit(0);
            }
            bit_file->cur_byte = 0;
            bit_file->mask = 0x80;
        }
        mask >>= 1;
    }
}

void main()
{
    FILE *f_in,*ff1,*ff2;
    BIT_FILE *f_out;
    union REGS inregs,outregs;
    char file[14],file1[14],ff[14];
    int row,col,outp[N][N],qty,j;
    int i=0,wait=1;
    unsigned char *inp[N];
    unsigned char c;
    char choice;

    clrscr();

    printf("Enter Filename for Compression :  ");
    gets(file);
    strcpy(file1,file);
    if((f_in = fopen(file,"rb")) == NULL)
    {
        printf("\nERROR : File not present...");
        exit(0);
    }
    rewind(f_in);

    while(wait)
```

```
        (file[i+2] == 'm' || file[i+2] == 'M') &&
        (file[i+3] == 'p' || file[i+3] == 'P'))
      {
        file1[i]='.';file1[i+1]='b';
        file1[i+2]='a'; file1[i+3]='r';

        valid_bmp(f_in);
        f_out = OpenOutputBitFile(file1);
        write_bmp(f_out,f_in);
        fseek(f_in,bmpfileheader.offset,SEEK_SET);
        wait=0;
      }
      if(strlen(file) < i)
      {
         printf("\n ERROR : This file format is not supported ..."
         exit(0);
      }
      else
         i++;
    }

    printf("\n\nEnter the quality factor (1-25) : ");
    scanf("%d",&qty);

    printf("\n\t\t Processing file, Please Wait ");

    Init(qty);
    OutputBits(f_out,(unsigned long)qty,8);
    tmp = ftell(f_out->file);

    for (row=0; row<ROWS; row+=N)
    {
       Read_input_file(f_in,pix_value);

       for (col=0; col<COLS; col+=N)
       {
          for (i=0; i<N; i++)
            inp[i] = pix_value[i] + col;

          FDCT(inp,outp);
          Quantization(f_out,outp);
       }
    }

    OutputCode(f_out,1);
    CloseOutputBitFile(f_out);
    fclose(f_in);
    printf("\n\nCompressed file name : %s", file1);

    /* Delete file */
    fflush(stdin);
    printf("\n\nDelete  %s  (Yes/No) ? ", file);
    scanf("%c", &choice);
    if((choice == 'y') || (choice == 'Y'))
    {
       inregs.x.dx = (int) &file;
       inregs.h.ah = 0x41;
       intdos(&inregs, &outregs);
```

```
/* Intializing the Quantization and cosine matrices */
void Init(int qty)
{
    int i,j;
    double pi=atan(1.Ø)*4.Ø;
    for (i=Ø; i<N; i++)
      for (j=Ø;j<N;j++)
        Quantum[i][j] = 1 + ((1+i+j) * qty);

    ORL = Ø;

    for (j=Ø; j<N; j++)
    {
       C[Ø][j]   = 1.Ø / sqrt((double)N);
       Ct[j][Ø]  = C[Ø][j];
    }
    for (i=1; i<N; i++)
    {
      for (j=Ø; j<N; j++)
      {
         C[i][j]   = sqrt(2.Ø/N) * cos(pi*(2*j+1)*i / (2.Ø*N));
         Ct[j][i]  = C[i][j];
      }
    }
 }

 /* Forward cosine transform */
 /* This is basically to prepare the input
     matrix for Quantization */
 void FDCT(unsigned char *input[N],int output[N][N])
 {
     double temp[N][N];
     double temp1;
     int i,j,k;

     for (i=Ø; i<N; i++)
       for (j=Ø; j<N; j++)
       {
           temp[i][j]=Ø.Ø;
           for (k=Ø; k<N; k++)
             temp[i][j] += ((int)input[i][k]-128) * Ct[k][j];
       }

     for (i=Ø; i<N; i++)
     {
       for (j=Ø; j<N; j++)
       {
           temp1 = Ø.Ø;
           for (k=Ø; k<N; k++)
             temp1 += C[i][k] * temp[k][j];

           output[i][j] = ROUND(temp1);
       }
     }
  }

  /* Performing Quantization on FDCT values */
     [illegible] is performed to make most of the
    int iz,row,col;
    double result;

    for (iz=0; iz<(N*N); iz++)
    {
        row = ZigZag[iz].row;
        col = ZigZag[iz].col;
        result = out_dat[row][col] / Quantum[row][col];
        OutputCode(Out_File,ROUND(result));

    }
}
/* Reading the input from the input file */
void Read_input_file(FILE *input,unsigned char **dat)
{
    int row,cols,c;
    for (row=0; row<N; row++)
    {
        for (cols=0; cols<COLS; cols++)
        {
            c = fgetc(input);
            if(c == EOF)
            {
                printf("Error Reading Input File...");
                exit(0);
            }
            dat[row][cols]=(unsigned char)c;
        }
    }
}

/* writing bit values onto the compression file */
void OutputCode(BIT_FILE *Out_File,int code)
{
    int top_of_range,bit_count,abs_code;

    if(code == 0)
    {
        ORL++;
        return;
    }

    if(ORL != 0)
    {
        while(ORL > 0)
        {
            OutputBits(Out_File,0L,2);
            if(ORL <= 16)
            {
                OutputBits(Out_File,(unsigned long)(ORL-1),4);
                ORL=0;
            }
            else
            {
                OutputBits(Out_File,15L,4);
                ORL -= 16;
            }
```

```
else
    abs_code=code;

 top_of_range=1;
 bit_count=1;

 while(abs_code > top_of_range)
 {
     bit_count++;
     top_of_range=((top_of_range+1)*2)-1;
  }
  if(bit_count < 3)
    OutputBits(Out_File,(unsigned long)(bit_count+1),3);
  else
    OutputBits(Out_File,(unsigned long)(bit_count+5),4);

  if(code > 0)
    OutputBits(Out_File,(unsigned long)code,bit_count);
  else
    OutputBits(Out_File,
                  (unsigned long)(code+top_of_range),bit_count);


}

/* checking validity of BMP file */
void valid_bmp(FILE *fp)
{
    int i;

    printf("\n\t\t Please wait reading file...");

    fread(&bmpfileheader.type    ,
            sizeof(bmpfileheader.type)     ,1,fp);

    fread(&bmpfileheader.size    ,
            sizeof(bmpfileheader.size)     ,1,fp);
    fread(&bmpfileheader.xhs     ,
            sizeof(bmpfileheader.xhs)      ,1,fp);
    fread(&bmpfileheader.yhs     ,
            sizeof(bmpfileheader.yhs)      ,1,fp);
    fread(&bmpfileheader.offset,
            sizeof(bmpfileheader.offset)  ,1,fp);

    fread(&bmpheader.size                 ,
            sizeof(bmpheader.size),1,fp);
    fread(&bmpheader.width                ,
            sizeof(bmpheader.width),1,fp);
    if(bmpheader.width > 1024)
    {
        printf("\n\t\tFATAL ERROR : Not a BMP format ");
        exit(0);
    }
    fread(&bmpheader.height               ,
            sizeof(bmpheader.height),1,fp);
    fread(&bmpheader.numbitplanes         ,
            sizeof(bmpheader.numbitplanes),1,fp);
    fread(&bmpheader.numbitsperplane      ,
               f(bmpheader.numbitsperplane),1,fp);
        sizeof(bmpheader.sizeofimagedata),1,fp);
fread(&bmpheader.xresolution              ,
        sizeof(bmpheader.xresolution),1,fp);
fread(&bmpheader.yresolution              ,
        sizeof(bmpheader.yresolution),1,fp);
fread(&bmpheader.numcolorsused            ,
        sizeof(bmpheader.numcolorsused),1,fp);
fread(&bmpheader.numimportantcolors       ,
        sizeof(bmpheader.numimportantcolors),1,fp);
fread(&bmpheader.resolutionunits          ,
        sizeof(bmpheader.resolutionunits),1,fp);
fread(&bmpheader.padding                  ,
        sizeof(bmpheader.padding),1,fp);
fread(&bmpheader.origin                   ,
        sizeof(bmpheader.origin),1,fp);
fread(&bmpheader.halftonning              ,
        sizeof(bmpheader.halftonning),1,fp);
fread(&bmpheader.halftonningparam1        ,
        sizeof(bmpheader.halftonningparam1),1,fp);
fread(&bmpheader.halftonningparam2        ,
        sizeof(bmpheader.halftonningparam2),1,fp);
fread(&bmpheader.colorencoding            ,
        sizeof(bmpheader.colorencoding),1,fp);
fread(&bmpheader.identifier               ,
        sizeof(bmpheader.identifier),1,fp);


  pix_value = (unsigned char **)malloc(N * sizeof(char *));
  for (i=0; i<N; i++)
     *(pix_value+i) =
     (char *)calloc((COLS=bmpheader.width),sizeof(char));
  ROWS = bmpheader.height;

}

/* writing bit values onto file */
void write_bmp(BIT_FILE *fp,FILE *fp1)
{
   long filepos,c;
   OutputBits(fp,bmpfileheader.type     ,
              sizeof(bmpfileheader.type)     *8);


   OutputBits(fp,bmpfileheader.size     ,
              sizeof(bmpfileheader.size)     *8);
   OutputBits(fp,bmpfileheader.xhs      ,
              sizeof(bmpfileheader.xhs)      *8);
   OutputBits(fp,bmpfileheader.yhs      ,
              sizeof(bmpfileheader.yhs)      *8);
   OutputBits(fp,bmpfileheader.offset,
              sizeof(bmpfileheader.offset)  *8);


   OutputBits(fp,bmpheader.size                  ,
              sizeof(bmpheader.size)*8);
   OutputBits(fp,bmpheader.width                 ,
              sizeof(bmpheader.width)*8);
   OutputBits(fp,bmpheader.height                ,
              sizeof(bmpheader.height)*8);
   OutputBits(fp,bmpheader.numbitplanes          ,
              sizeof(bmpheader.numbitplanes)*8);
              sizeof(bmpheader.numbitplanes
                     sizeof(bmpheader.compressionscheme)*8);
                     sizeof(bmpheader.sizeofimagedata        ;
OutputBits(fp,bmpheader.sizeofimagedata)*8);
                     sizeof(bmpheader.xresolution            ;
OutputBits(fp,bmpheader.xresolution)*8);
                     sizeof(bmpheader.yresolution            ;
OutputBits(fp,bmpheader.yresolution)*8);
                     sizeof(bmpheader.numcolorsused          ;
OutputBits(fp,bmpheader.numcolorsused)*8);
                     sizeof(bmpheader.numimportantcolors     ;
OutputBits(fp,bmpheader.numimportantcolors)*8);
                     sizeof(bmpheader.resolutionunits        ;
OutputBits(fp,bmpheader.resolutionunits)*8);
                     sizeof(bmpheader.padding                ;
OutputBits(fp,bmpheader.padding)*8);
                     sizeof(bmpheader.origin                 ;
OutputBits(fp,bmpheader.origin)*8);
                     sizeof(bmpheader.halftonning            ;
OutputBits(fp,bmpheader.halftonning)*8);
                     sizeof(bmpheader.halftonningparam1      ;
OutputBits(fp,bmpheader.halftonningparam1)*8);
                     sizeof(bmpheader.halftonningparam2      ;
OutputBits(fp,bmpheader.halftonningparam2)*8);
                     sizeof(bmpheader.colorencoding          ;
OutputBits(fp,bmpheader.colorencoding)*8);
                     sizeof(bmpheader.identifier             ;
OutputBits(fp,bmpheader.identifier)*8);
                     sizeof(bmpheader.identifier

  filepos=ftell(fp1);
  while(filepos < bmpfileheader.offset)
  {
      c = fgetc(fp1);
      OutputBits(fp,(unsigned char)c,8);
      filepos++;
   }
}
```

```
/* IMAGE DECOMPRESSION */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <alloc.h>
#include <conio.h>
#include <dos.h>
#include <io.h>

#define N          8
#define ROUND(a) (((a)<0) ? (int)((a)-0.5) : (int)((a)+0.5))

#define uint16 unsigned int
#define uint32 unsigned long int
#define int16  int
#define int32  long int

typedef struct bit_file
{
    FILE *file;
    unsigned char mask;
    int cur_byte;
}BIT_FILE;

struct
{
    uint16 type;
    uint32 size;
    int16  xhs;
    int16  yhs;
    uint32 offset;
}bmpfileheader;

struct
{
    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 halftonning;
    uint32 halftonningparam1;
    uint32 halftonningparam2;
    uint32 colorencoding;
    uint32 identifier;
}bmpheader;

                   [illegible]
```

```
}

/* reading from compression file */
unsigned long InputBits(BIT_FILE *bit_file,int bit_count)
{
   unsigned long mask;
   unsigned long return_value;

   mask = 1L <<(bit_count-1);
   return_value = 0;
   while(mask != 0)
   {
     if(bit_file->mask == 0x80)
     {
        bit_file->cur_byte = getc(bit_file->file);
        if(bit_file->cur_byte == EOF)
        {
          printf("Error In InputBits\n");
          exit(0);
        }
     }
     if(bit_file->cur_byte & bit_file->mask)
     return_value |= mask;
     mask >>= 1;
     bit_file->mask >>= 1;
     if(bit_file->mask == 0)
       bit_file -> mask = 0x80;
   }
   return(return_value);
}

void main()
{
   FILE *f_out,*ff1;
   BIT_FILE *f_in;
   union REGS inregs,outregs;
   char file[14],file1[14],ff[14];
   int row,col,inp[N][N],qty,j;
   int i=0,wait=1;
   unsigned char *outp[N];
   char c;

   clrscr();

   printf("Enter Filename for Decompression :   ");
   gets(file);
   strcpy(file1,file);
   f_in = OpenInputBitFile(file);

   while(wait)
   {
      if((file[i]   == '.') &&
         (file[i+1] == 'b' || file[i+1] == 'B') &&
         (file[i+2] == 'a' || file[i+2] == 'A') &&
         (file[i+3] == 'r' || file[i+3] == 'R'))
      {
                 file1[i+1]='b';
          wait=0;
          valid_bmp(f_in);
          write_bmp(f_out,f_in);
       }
       if(strlen(file) <= i)
       {
          printf("\n\nERROR :  ");
          printf("This file format is not supported .. ");
          exit(0);
       }
       else
          i++;
   }
   tmp=ftell(f_in->file);
   qty = (int)InputBits(f_in,8);
   init(qty);
   printf("\n   Prcessing for quality factor :  %d",qty);
   printf("\n\n\t\tPlease Wait ");

   for (row=0; row<ROWS; row+=N)
   {
        for (col=0; col<COLS; col+=N)
        {
            for (i=0; i<N; i++)
              outp[i] = pix_value[i]+col;

            Dequantization(f_in,inp);
            IDCT(inp,outp);
        }
        write_outfile(f_out,pix_value);
   }

   CloseInputBitFile(f_in);
   fclose(f_out);
   fflush(stdin);
   printf("\n\n Decompressed file name : %s\n", file1);

   /* Deleting compressed file */
         inregs.x.dx = (int) &file;
         inregs.h.ah = 0x41;
         intdos(&inregs,&outregs);
}

void init(int qty)
{
   int i,j;
   double pi=atan(1.0)*4.0;
   for (i=0; i<N; i++)
       for (j=0; j<N; j++)
          Quantum[i][j] = 1 + ((1+i+j) * qty);
   IRL = 0;

   for (j=0; j<N; j++)
   {
       c[0][j]  = 1.0 / sqrt((double)N);
       ct[j][0] = c[0][j];
   }
   for (i=1; i<N; i++)
```

```
         c[i][j]  = sqrt(2.0/N)*cos(pi*(2*j+1)*i/(2.0*N))
         ct[j][i] = c[i][j];
      }
   }
}

void Dequantization(BIT_FILE *in_file,int dat[N][N])
{
   int iz,row,col;
   for (iz=0; iz<(N*N); iz++)
   {
      row = ZigZag[iz].row;
      col = ZigZag[iz].col;
      dat[row][col] = InputCode(in_file) * Quantum[row][col];
   }
}

void IDCT(int input[N][N],unsigned char *output[N])
{
   double temp[N][N];
   double temp1;
   int i,j,k;

   for (i=0; i<N;i++)
   {
      for (j=0; j<N; j++)
      {
         temp[i][j] = 0.0;
         for (k=0; k<N; k++)
            temp[i][j] += input[i][k] * c[k][j];
      }
   }

   for (i=0; i<N; i++)
   {
      for (j=0; j<N; j++)
      {
         temp1 = 0.0;
         for (k=0; k<N; k++)
            temp1 += ct[i][k] * temp[k][j];
         temp1 += 128.0;
         if(temp1 < 0)
            output[i][j] = 0;
         else
            if(temp1 > 255)
               output[i][j] = 255;
            else
               output[i][j] = (unsigned char)ROUND(temp1);

      }
   }
}

void write_outfile(FILE *output,unsigned char **strip)
{
   int row,col;

   for (row=0; row<N; row++)
      for (col=0; col<COLS; col++)
```

```
/* read bit values from the compressed file */
int InputCode(BIT_FILE *in_file)
{
    int bit_count,result;
    if(IRL > 0)
    {
        IRL--;
        return(0);
    }

    bit_count = (int)InputBits(in_file,2);
    if(bit_count == 0)
    {
        IRL = (int)InputBits(in_file,4);
        return(0);
    }

    if(bit_count == 1)
        bit_count = (int)InputBits(in_file,1)+1;
    else
        bit_count = (int)InputBits(in_file,2)+(bit_count<<2)-5;

    result = (int)InputBits(in_file,bit_count);

    if(result & (1 << (bit_count-1) ) )
        return(result);

    return(result - (1 << bit_count) + 1 );
}

/* check for validity of compressed file */
void valid_bmp(BIT_FILE *fpi)
{
   int i;

   bmpfileheader.type   = (int)InputBits(fpi,
                          sizeof(bmpfileheader.type)*8);

   if(bmpfileheader.type != 19778)
   {
      printf("\n\n FATAL ERROR : Not a BMP format...");
      exit(0);
   }

   bmpfileheader.size    = (int)InputBits(fpi,
                           sizeof(bmpfileheader.size)   *8);
   bmpfileheader.xhs     = (int)InputBits(fpi,
                           sizeof(bmpfileheader.xhs)    *8);
   bmpfileheader.yhs     = (int)InputBits(fpi,
                           sizeof(bmpfileheader.yhs)    *8);
   bmpfileheader.offset  = (int)InputBits(fpi,
                           sizeof(bmpfileheader.offset) *8);

   bmpheader.size          = (int)InputBits(fpi,
                             sizeof(bmpheader.size)*8);
   bmpheader.width         = (int)InputBits(fpi,
                             sizeof(bmpheader.width)*8);
   bmpheader.height        = (int)InputBits(fpi,
  bmpheader.numbitsperplane=(int)InputBits(fpi,
                              sizeof(bmpheader.numbitsperplane)*8
  bmpheader.compressionscheme=(int)InputBits(fpi,
                                sizeof(bmpheader.compressionschem
  *8);

  if(bmpheader.compressionscheme != 0)
  {
     printf("\n\nERROR : File cannot be decompressed ...");
     exit(0);
  }

  bmpheader.sizeofimagedata=(int)InputBits(fpi,
                              sizeof(bmpheader.sizeofimagedata)
  *8);
  bmpheader.xresolution=(int)InputBits(fpi,
                          sizeof(bmpheader.xresolution)
  *8);
  bmpheader.yresolution=(int)InputBits(fpi,
                          sizeof(bmpheader.yresolution)
  *8);
  bmpheader.numcolorsused=(int)InputBits(fpi,
                            sizeof(bmpheader.numcolorsused)
  *8);
  bmpheader.numimportantcolors=(int)InputBits(fpi,
                            sizeof(bmpheader.numimportantcolors)
  *8);
  bmpheader.resolutionunits=(int)InputBits(fpi,
                            sizeof(bmpheader.resolutionunits)
  *8);
  bmpheader.padding=(int)InputBits(fpi,
                            sizeof(bmpheader.padding)
  *8);
  bmpheader.origin=(int)InputBits(fpi,
                            sizeof(bmpheader.origin)
  *8);
  bmpheader.halftonning=(int)InputBits(fpi,
                            sizeof(bmpheader.halftonning)
  *8);
  bmpheader.halftonningparam1=(int)InputBits(fpi,
                            sizeof(bmpheader.halftonningparam1)
  *8);
  bmpheader.halftonningparam2=(int)InputBits(fpi,
                            sizeof(bmpheader.halftonningparam2)
  *8);
  bmpheader.colorencoding=(int)InputBits(fpi,
                            sizeof(bmpheader.colorencoding)
  *8);
  bmpheader.identifier=(int)InputBits(fpi,
                            sizeof(bmpheader.identifier)
  *8);

  pix_value = (unsigned char **)malloc(N * sizeof(char *));
  for (i=0; i<N; i++)
     *(pix_value+i)=
     (char *)calloc((COLS=bmpheader.width),sizeof(char));
  ROWS = bmpheader.height;
}
```

```
{
    long filepos;
    char c;
    fwrite(&bmpfileheader.type    ,
           sizeof(bmpfileheader.type)    ,1,fpo);

    fwrite(&bmpfileheader.size    ,
           sizeof(bmpfileheader.size)    ,1,fpo);
    fwrite(&bmpfileheader.xhs     ,
           sizeof(bmpfileheader.xhs)     ,1,fpo);
    fwrite(&bmpfileheader.yhs     ,
           sizeof(bmpfileheader.yhs)     ,1,fpo);
    fwrite(&bmpfileheader.offset,
           sizeof(bmpfileheader.offset)  ,1,fpo);

    fwrite(&bmpheader.size                  ,
           sizeof(bmpheader.size)
    ,1,fpo);
    fwrite(&bmpheader.width                 ,
           sizeof(bmpheader.width)
    ,1,fpo);
    fwrite(&bmpheader.height                ,
           sizeof(bmpheader.height)
    ,1,fpo);
    fwrite(&bmpheader.numbitplanes          ,
           sizeof(bmpheader.numbitplanes)
    ,1,fpo);
    fwrite(&bmpheader.numbitsperplane       ,
           sizeof(bmpheader.numbitsperplane)
    ,1,fpo);
    fwrite(&bmpheader.compressionscheme     ,
           sizeof(bmpheader.compressionscheme)
    ,1,fpo);
    fwrite(&bmpheader.sizeofimagedata       ,
           sizeof(bmpheader.sizeofimagedata)
    ,1,fpo);
    fwrite(&bmpheader.xresolution           ,
           sizeof(bmpheader.xresolution)
    ,1,fpo);
    fwrite(&bmpheader.yresolution           ,
           sizeof(bmpheader.yresolution)
    ,1,fpo);
    fwrite(&bmpheader.numcolorsused         ,
           sizeof(bmpheader.numcolorsused)
    ,1,fpo);
    fwrite(&bmpheader.numimportantcolors    ,
           sizeof(bmpheader.numimportantcolors)
    ,1,fpo);
    fwrite(&bmpheader.resolutionunits       ,
           sizeof(bmpheader.resolutionunits)
    ,1,fpo);
    fwrite(&bmpheader.padding               ,
           sizeof(bmpheader.padding)
    ,1,fpo);
    fwrite(&bmpheader.origin                ,
           sizeof(bmpheader.origin)
```

```
                sizeof(bmpheader.halftonningparam1)
    ,1,fpo);
    fwrite(&bmpheader.halftonningparam2         ,
                sizeof(bmpheader.halftonningparam2)
    ,1,fpo);
    fwrite(&bmpheader.colorencoding             ,
                sizeof(bmpheader.colorencoding)
    ,1,fpo);
    fwrite(&bmpheader.identifier                ,
                sizeof(bmpheader.identifier)
    ,1,fpo);

    filepos=ftell(fpi->file);
    while(filepos < bmpfileheader.offset)
    {
        c = (int)InputBits(fpi,8);
        fputc((unsigned char)c,fpo);
        filepos++;
    }
}
```

```
/* Program to write BMP image file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "bitio.c"

#define uint16 unsigned int
#define uint32 unsigned long int
#define int16  int
#define int32  long int

struct
{
   uint16 type;
   uint32 size;
   int16  xhs;
   int16  yhs;
   uint32 offset;
}bmpfileheader;

struct
{
   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 halftonning;
   uint32 halftonningparam1;
   uint32 halftonningparam2;
   uint32 colorencoding;
   uint32 identifier;
}bmpheader;

uint32 count=0;

void write_bmp(FILE *, FILE *);
void valid_bmp(FILE *);

void main()
{
   FILE *fi,*fo;
   char in[14],c;

   clrscr();
   printf("\n Enter the input filename : ");
   gets(in);
```

```
    rewind(fi);
    valid_bmp(fi);
    write_bmp(fo,fi);
    fclose(fi);
    fclose(fo);
}

void valid_bmp(FILE *fp)
{
    fread(&bmpfileheader.type   ,sizeof(bmpfileheader.type)   ,1,

    fread(&bmpfileheader.size   ,sizeof(bmpfileheader.size)   ,1,
    fread(&bmpfileheader.xhs    ,sizeof(bmpfileheader.xhs)    ,1,
    fread(&bmpfileheader.yhs    ,sizeof(bmpfileheader.yhs)    ,1,
    fread(&bmpfileheader.offset,sizeof(bmpfileheader.offset) ,1,

    if(bmpfileheader.offset != 118)
    {
       printf("\n\nFATAL ERROR : Wrong file format,not a BMP fil
       exit(0);
    }
    fread(&bmpheader.size                     ,
          sizeof(bmpheader.size),1,fp);
    fread(&bmpheader.width                    ,
          sizeof(bmpheader.width),1,fp);
    fread(&bmpheader.height                   ,
          sizeof(bmpheader.height),1,fp);
    fread(&bmpheader.numbitplanes             ,
          sizeof(bmpheader.numbitplanes),1,fp);
    fread(&bmpheader.numbitsperplane          ,
          sizeof(bmpheader.numbitsperplane),1,fp);
    fread(&bmpheader.compressionscheme        ,
          sizeof(bmpheader.compressionscheme),1,fp);
    fread(&bmpheader.sizeofimagedata          ,
          sizeof(bmpheader.sizeofimagedata),1,fp);
    fread(&bmpheader.xresolution              ,
          sizeof(bmpheader.xresolution),1,fp);
    fread(&bmpheader.yresolution              ,
          sizeof(bmpheader.yresolution),1,fp);
    fread(&bmpheader.numcolorsused            ,
          sizeof(bmpheader.numcolorsused),1,fp);
    fread(&bmpheader.numimportantcolors       ,
          sizeof(bmpheader.numimportantcolors),1,fp);
    fread(&bmpheader.resolutionunits          ,
          sizeof(bmpheader.resolutionunits),1,fp);
    fread(&bmpheader.padding                  ,
          sizeof(bmpheader.padding),1,fp);
    fread(&bmpheader.origin                   ,
          sizeof(bmpheader.origin),1,fp);
    fread(&bmpheader.halftonning              ,
          sizeof(bmpheader.halftonning),1,fp);
    fread(&bmpheader.halftonningparam1        ,
          sizeof(bmpheader.halftonningparam1),1,fp);
    fread(&bmpheader.halftonningparam2        ,
          sizeof(bmpheader.halftonningparam2),1,fp);
    fread(&bmpheader.colorencoding            ,
          sizeof(bmpheader.colorencoding),1,fp);
    fread(&bmpheader.identifier               ,
void write_bmp(FILE *fp,FILE *fp1)
{
   char c;
   fwrite(&bmpfileheader.type   ,sizeof(bmpfileheader.type),1,fp

   fwrite(&bmpfileheader.size   ,sizeof(bmpfileheader.size)     ,1
   fwrite(&bmpfileheader.xhs    ,sizeof(bmpfileheader.xhs)      ,1
   fwrite(&bmpfileheader.yhs    ,sizeof(bmpfileheader.yhs)      ,1
   fwrite(&bmpfileheader.offset,sizeof(bmpfileheader.offset)  ,1

   fwrite(&bmpheader.size                       ,
            sizeof(bmpheader.size),1,fp);
   fwrite(&bmpheader.width                      ,
            sizeof(bmpheader.width),1,fp);
   fwrite(&bmpheader.height                     ,
            sizeof(bmpheader.height),1,fp);
   fwrite(&bmpheader.numbitplanes               ,
            sizeof(bmpheader.numbitplanes),1,fp);
   fwrite(&bmpheader.numbitsperplane            ,
            sizeof(bmpheader.numbitsperplane),1,fp);
   fwrite(&bmpheader.compressionscheme          ,
            sizeof(bmpheader.compressionscheme),1,fp);
   fwrite(&bmpheader.sizeofimagedata            ,
            sizeof(bmpheader.sizeofimagedata),1,fp);
   fwrite(&bmpheader.xresolution                ,
            sizeof(bmpheader.xresolution),1,fp);
   fwrite(&bmpheader.yresolution                ,
            sizeof(bmpheader.yresolution),1,fp);
   fwrite(&bmpheader.numcolorsused              ,
            sizeof(bmpheader.numcolorsused),1,fp);
   fwrite(&bmpheader.numimportantcolors         ,
            sizeof(bmpheader.numimportantcolors),1,fp);
   fwrite(&bmpheader.resolutionunits            ,
            sizeof(bmpheader.resolutionunits),1,fp);
   fwrite(&bmpheader.padding                    ,
            sizeof(bmpheader.padding),1,fp);
   fwrite(&bmpheader.origin                     ,
            sizeof(bmpheader.origin),1,fp);
   fwrite(&bmpheader.halftonning                ,
            sizeof(bmpheader.halftonning),1,fp);
   fwrite(&bmpheader.halftonningparam1          ,
            sizeof(bmpheader.halftonningparam1),1,fp);
   fwrite(&bmpheader.halftonningparam2          ,
            sizeof(bmpheader.halftonningparam2),1,fp);
   fwrite(&bmpheader.colorencoding              ,
            sizeof(bmpheader.colorencoding),1,fp);
   fwrite(&bmpheader.identifier                 ,
            sizeof(bmpheader.identifier),1,fp);

   while((c = fgetc(fp1)) != EOF)
   {
       fputc(c,fp);
       count++;
   }
   printf("%Ld",count);
 }
```

```
/* Program to write IMG image file */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "bitio.c"

#define uint16 unsigned int
#define uint32 unsigned long int
#define int16  int
#define int32  long int
#define ch8    char
#define uch8   unsigned char

struct
{
   int16  fv1;
   int16  fv2;
   int16  fv3;
   int16  patternlength;
   int16  pixelwidth;
   int16  pixelheight;
   uint16 scancols;
   uint16 scanrows;
}imgheader;

void write_img(FILE *, FILE *);
void valid_img(FILE *);

void main()
{
   FILE *fi,*fo;
   char in[14];
   clrscr();
   printf("\n Enter the input filename : ");
   gets(in);
   fi = fopen(in,"rb");
   fo = fopen("testi.img","wb");
   rewind(fi);
   valid_img(fi);
   write_img(fo,fi);
   fclose(fi);
   fclose(fo);
}

void valid_img(FILE *fp)
{
   fread(&imgheader.fv1           ,
          sizeof(imgheader.fv1)            ,1,fp);
   fread(&imgheader.fv2           ,
          sizeof(imgheader.fv2)            ,1,fp);
   fread(&imgheader.fv3           ,
          sizeof(imgheader.fv3)            ,1,fp);
   fread(&imgheader.patternlength ,
          sizeof(imgheader.patternlength),1,fp);
   fread(&imgheader.pixelwidth    ,
          sizeof(imgheader.pixelwidth)     ,1,fp);
   fread(&imgheader.pixelheight   ,
          sizeof(imgheader.pixelheight)    ,1,fp);
```

```
    fread(&imgheader.scancols         ,
            sizeof(imgheader.scancols)        ,1,fp);
}

void write_img(FILE *fp, FILE *fp1)
{
    int c;
    fwrite(&imgheader.fv1                   ,
             sizeof(imgheader.fv1),1,fp);
    fwrite(&imgheader.fv2                   ,
             sizeof(imgheader.fv2),1,fp);
    fwrite(&imgheader.fv3                   ,
             sizeof(imgheader.fv3),1,fp);
    fwrite(&imgheader.patternlength ,
             sizeof(imgheader.patternlength),1,fp);
    fwrite(&imgheader.pixelwidth        ,
             sizeof(imgheader.pixelwidth),1,fp);
    fwrite(&imgheader.pixelheight      ,
             sizeof(imgheader.pixelheight),1,fp);
    fwrite(&imgheader.scanrows          ,
             sizeof(imgheader.scanrows),1,fp);
    fwrite(&imgheader.scancols          ,
             sizeof(imgheader.scancols),1,fp);

    while((c = fgetc(fp1)) != EOF)
        fputc(c,fp);
}
```