Digital Image Processing Laboratory: Achromatic Baseline JPEG encoding Lab

May 11, 2011

1 Introduction

JPEG is an acronym for the "Joint Picture Expert Group" (www.jpeg.org). This committee was formed jointly by ISO and ITU to form standards for continuous tone image compression. The standard has been very successful, and has resulted in a commonly used baseline format known as JPEG based on lossy block transform coding of grayscale and color images. JPEG has become very popular because it is a non-proprietary standard, and because it is simple and efficient to implement while yielding good performance.

The JPEG standard supports a number of standard modes as described below:

- **Sequential mode** - Block-by-block lossy encoding in raster scan order based on block transform coding using DCT's.
- **Progressive mode** - Coded image is transferred starting with coarse resolution information and progressing to finer resolution detail.
- **Lossless mode** - Lossless image coding based on predictive coding using a neighborhood of 3 samples.
- **Hierarchical mode** - lower-resolution image is encoded first, upsampled and interpolated to predict the full-resolution image and the prediction error is encoded with one of above 3 operation modes.

In this lab, you will learn about block transform coding by implementing the baseline JPEG standard. The baseline JPEG coder is the simplest version of DCT-based sequential coder. For simplicity, we will only consider coding of 8-bit grayscale images.

Figure 1 illustrates the main procedures of the DCT-based JPEG encoder. The source image is partitioned into 8×8 blocks. Then, each block is transformed through a forward discrete cosine transform (FDCT) and quantized. After the final step of entropy coding, we can get the compressed JPEG data. Decompression the requires reversing these steps with entropy decoding followed by computation of the inverse discrete cosine transform (IDCT).

Questions or comments concerning this laboratory should be directed to Prof. Charles A. Bouman, School of Electrical and Computer Engineering, Purdue University, West Lafayette IN 47907; (765) 494-0340; bouman@ecn.purdue.edu

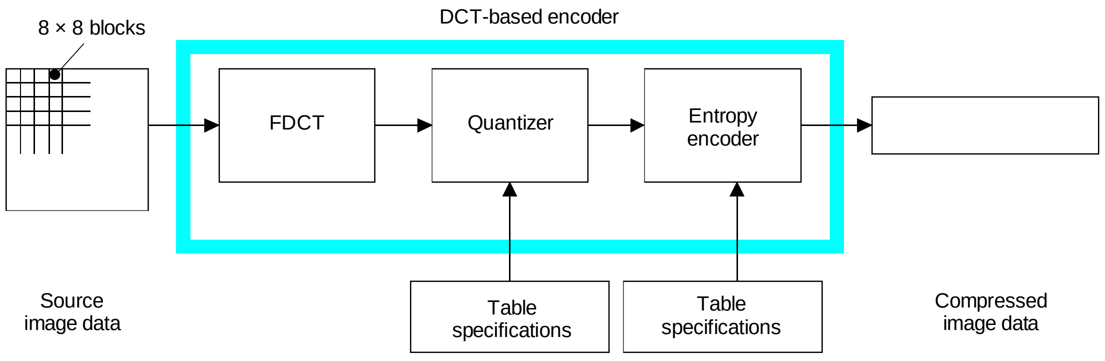


Figure 1: DCT-based JPEG encoder simplified diagram

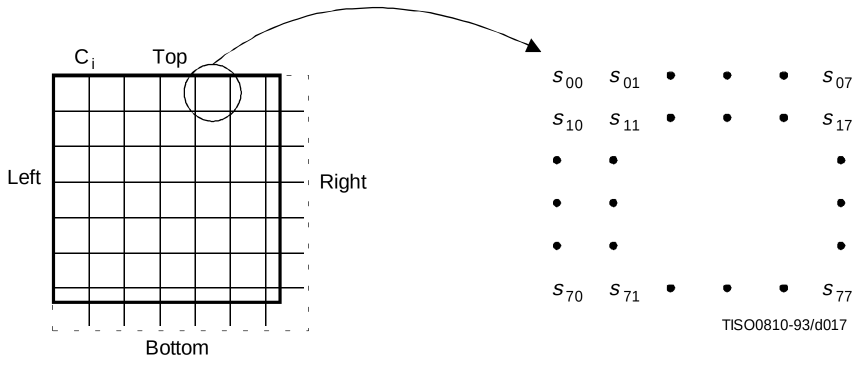


Figure 2: Partition and orientation of 8x8 blocks

2 DCT Block Transforms and Quantization

The source image is first broken into 8×8 blocks. The pixel values s_{yx} in each of these blocks are then transformed by the FDCT into an 8×8 block of 64 DCT coefficients. Figure 2 illustrates this process with the DCT coefficients denoted by the variables S_{vu} where v and u are integer frequency variables between 0 and 7. The coefficient S_{00} is known as the DC coefficient because it represents the average value of the block, whereas the remaining values are known as the AC coefficients.

The FDCT and IDCT are defined as follows:

$$\begin{aligned}
&FDCT: \\
&S_{vu} = \frac{1}{4} C_u C_v \sum_{x=0}^{7} \sum_{y=0}^{7} s_{yx} cos \frac{(2x+1)\,u\pi}{16} cos \frac{(2y+1)\,v\pi}{16} \\
&IDCT: \\
&s_{yx} = \frac{1}{4} \sum_{u=0}^{7} \sum_{v=0}^{7} C_u C_v S_{vu} cos \frac{(2x+1)\,u\pi}{16} cos \frac{(2y+1)\,v\pi}{16}
\end{aligned}$$

where C_u and C_v are defined by

$$C_u = \begin{cases} \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ 1 & \text{for } u \neq 0 \end{cases}$$

Note that the *dct2* function in Matlab conforms to this definition.

Prior to the FDCT operation, each 8×8 block is level shifted by subtracting 128. This requires a signed representation for each pixel, so 8-bit 2's complement form is used. More specifically, the image sample values are initially in the range of $[0, \ldots, 255]$; so after subtraction of 128 they are converted to values in the range $[-128, \ldots, 127]$ in 2's complement form.

A numerical analysis of the 8×8 FDCT coefficients shows that the non-fractional part of the DCT coefficients can grow by at most a factor of 8 or equivalently 3 bits. This means that after transformation by the FDCT the coefficients require 11=8+3 bit signed integers in 2's compliment form. So, the sample values in the range $[-128, \ldots, 127]$ may grow to the range $[-1024, \ldots, 1023]$.

After the 8×8 FDCT, each of the 64 resulting DCT coefficients is quantized by a uniform quantizer. The quantizer step size is defined in a 8×8 table of integers known as the quantization table. Loss of image information is caused by this quantization process with different step sizes resulting in different amounts of information loss. In general, larger quantization step sizes will result in reduced quality. Importantly, the preprocessing by the FDCT allows the amount of loss to be varied as a function of the spatial frequency of the image.

The uniform quantizer operates according to the following equation

$$Q_{vu} = round\left(\frac{S_{vu}}{\gamma \Delta_{vu}}\right) \tag{1}$$

where Q_{vu}, S_{vu}, Δ_{vu}, and γ are defined as follows.

Q_{vu} - Quantized DCT coefficient formed by rounding.

S_{vu} - Input sample (v, u) in 8×8 source image block.

Δ_{vu} - Step size at location (v, u) in the 8×8 quantization table.

γ - A scalar that controls overall quality level. A larger value of γ reduces quality, whereas a smaller value of γ increases quality. The maximum value of $\gamma\Delta_{vu}$ is upper bounded to 16 bits in the ITU T-81 recommendation.

Proper choice of the quantization matrix is critical to achieving good performance with the JPEG standard. Not surprisingly, there has been a great deal of research performed to determine the best selection of this matrix. It is well known that visual sensitivity falls off with higher spatial frequency; so generally, the higher frequency entries of the quantization table are proportionately larger. This tends to put more of the distortion into the higher spatial frequencies where they are less noticeable.

The JPEG standard specifies a typical quantization table; however the quantization table may also be specified in the file parameter field of the JPEG file header. Each coefficient of the quantization table is represented by an 11 bit signed 2's complement number. A typical quantization table for either a grayscale image or the luminance component of a color image is shown below.

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

Figure 3: Typical quantization table used for the luminance component of a JPEG image.

2.1 Exercises

In the following, you will use Matlab to apply block DCT transforms followed by quantization to an image in order to determine their effect.

1. Download the grayscale image *img03y.tif* from the laboratory home page. The image size has been chosen to have both height and width which is a multiple of 8. This will simplify block transform processing in 8×8 blocks.

2. Load *img03y.tif* into Matlab and convert the resulting matrix to type *double*.

3. Level shift the image by subtracting the value 128 from each element of the array.

4. Download the *JPEG utilities* from the laboratory home page. This zip file contains a Matlab script called *Qtables.m*.

5. The Matlab script *Qtables.m* contains definitions for the matrices *Quant* and *Zig*. You can load these matrices into your Matlab workspace by running the *Qtables.m* script. The variable *Quant* is an 8×8 matrix containing the typical quantization coefficients for JPEG.

6. Perform 8×8 block FDCT and Quantization using Matlab's built-in functions *dct2* and *blockproc*. The function *dct2* performs 2-D FDCT's of any specified size, and the function *blockproc* allows you to apply operations to each block of an image. You may apply this function using the commands:

   ```
   fn = @(x) round(dct2(x.data,[8,8])./(Quant*gamma));
   dct_blk = blockproc(img,[8,8],fn);
   ```

   The first line defines the function handle, *fn*, describing the operation to be performed on each block. The result of the block processing and quantization is stored in the matrix *dct_blk*.

7. Use Matlab's *fwrite* command to save the variables in *dct_blk* as 16-bit signed integers using 2's complement representation. Save the data in a file named *img03y.dq*. The file should start with two 16-bit signed values specifying the number of rows (height) and number of columns (width) in the image. This file will be used as an input file in a future section.

   This file may be written by using *fwrite* with the PRECISION field set to 'integer*2' . The file data should be stored in raster order going from left-to-right first, and then moving from top-to-bottom. (Note: Matlab naturally orders data along columns first rather then rows. Taking the matrix transpose before writing out will store the array in conventional raster order.)

8. Write a Matlab script that reads in the file *img03y.dq* and reverses the operations of block transformation and quantization to get the restored image. You should use the *blockproc* and *idct2* commands in this script.

9. Obtain the difference image by subtracting the restored image from the source image. The difference image may contain small or negative values. So you should scale the difference by 10 and level shift the values by adding 128 to make them positive.

10. For your report, print or export the original image, restored image, and also the shifted difference image. Repeat this procedure for $\gamma = 0.25$ and $\gamma = 4$.

Section 2 Report:
Do the following:

1. Hand in a hard copy of your Matlab script for block transforming, quantizing, and storing the file *img03y.dq*.

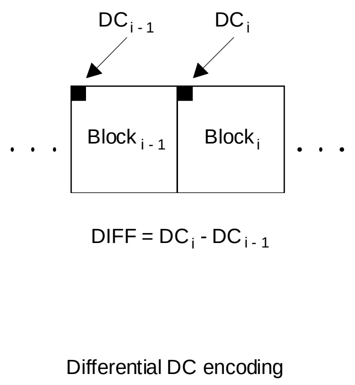
2. Hand in a hard copy of your Matlab script for restoring the image from the file *img03y.dq*.

3. Hand in a hard copy of the original, restored, and difference images for $\gamma = 0.25$, 1, and 4.

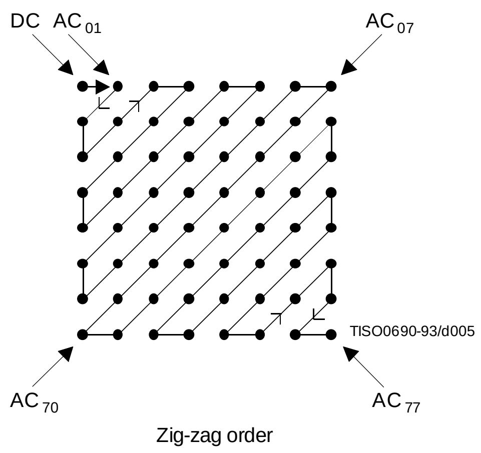
4. Comment on the effect of γ on your results.

2.2 Differential Encoding and the Zig-Zag Scan Pattern

Among the quantized 64 DCT coefficients, the DC coefficient is treated separately from the other 63 AC coefficients. (In the following sequel, when DCT coefficients are mentioned, they implicitly mean the quantized values.) This is because it corresponds to the average gray level of each 8×8 block. Generally, the DC coefficient has the highest energy of the DCT coefficients; so a fine quantization step size will lead to a large number of bits to code.



(a)



(b)

Figure 4: Encoding of DC and AC coefficients in JPEG. (a) The DC coefficient is differentially encoded from block to block using a raster ordering of the blocks. (b) The AC coefficients are ordered within each block using a Zig-zag scan pattern. The pattern is used so that small coefficients that are likely to be zero are grouped together.

Alternatively, a large quantization step size will result in a substantial DC shift, which can cause blocking artifacts in the restored image.

Another important property of the DC coefficient is that it tends to be highly correlated among adjacent image blocks. This is because the average gray level of adjacent image blocks is likely to be similar.

To improve image quality and reduce bit rate, the DC coefficient is differentially encoded. This means that, using a raster ordering of the blocks, only the difference between the current and previous DC coefficients is coded. The differential DC coefficient for the k^{th} block is computed by

$$DIFF(k) = Q_{00}^{(k)} - Q_{00}^{(k-1)} \tag{2}$$

where $Q_{00}^{(k)}$ is the current DC coefficient and $Q_{00}^{(k-1)}$ is the DC coefficient from the previous block in raster order. For the first block of the image, $Q_{00}^{(k-1)}$ is set to zero.

The remaining 63 AC coefficients have a high probability of being zero after quantization because the higher frequency coefficients usually have lower energy and the higher frequency quantization table entries are usually larger. Therefore, it is very important to order the coefficients so that coefficients of zeros are likely to be grouped together, forming long "run lengths" of zeros. To accomplish this goal, the AC coefficients are order using a so-called zig-zag scan pattern as illustrated in Figure 4.

2.3 Exercises

In the following, you will use Matlab to observe the properties of the DC and AC DCT coefficients.

1. Using the results from section 2.1 for $\gamma = 1.0$, extract the DC coefficient from each JPEG block. Organize the coefficients in a 2-D array corresponding to their positions in the original image. Then display this 2-D array as an image. To display negative value properly, add 128 to the DC coefficient values.

2. For each JPEG block, put the AC coefficients in zig-zag order. Do this by forming a Matlab array of size $N \times 63$ where N is the number of blocks in the image. You will find the variable *Zig* defined in *Qtables.m* useful for this step.

3. Compute the mean absolute value of each AC coefficient in zig-zag order (average across blocks). Plot the mean value as a function of the coefficient index.

Section 2 Report:
Do the following:

1. Hand in a hard copy of the image formed by the DC coefficients. What does it look like?

2. Explain why the DC coefficients of adjacent blocks are correlated.

3. Hand in your plot of the mean value of the magnitude of the AC coefficients for $\gamma = 1.0$. Explain the form of this plot.

3 Entropy Encoding of Coefficients

In order to reduce the number of bits required to represent the quantized image, both the differential DC and AC coefficients must be entropy encoded. To do this, JPEG uses two basic encoding schemes, the *Variable-Length Code* (VLC) and the *Variable-Length Integer* (VLI). The VLC encodes the number of bits used for each coefficient, and the VLI encodes the signed integer efficiently.

Each JPEG block is encoded as an integer sequence of bytes, starting with the differential DC coefficient $DIFF(k)$, and then continuing in the zig-zag ordered AC coefficients. The baseline JPEG encoding uses a Huffman encoding to reduce the average number of bits required to represent the coefficients. Huffman coding is a method of variable length entropy coding that associates a specific value with a unique code. The Huffman code is known as a prefix code because no code is the prefix or beginning of a longer coder word. This property makes the Huffman code uniquely decodable.

The differential DC coefficient $DIFF(k)$ is encoded using the following sequence of steps.

1. Let m be the number of bits required to represent $DIFF(k)$ without the sign. So for example, the value -3 has an unsigned binary representation of 11; so in this case, $m = 2$ bits.

Range of $DIFF(k)$ values	Bit Size (m)	Huffman Code
0	0	00
$-1, 1$	1	010
$-3, -2, 2, 3$	2	011
$-7 \ldots -4, 4 \ldots 7$	3	100
$-15 \ldots -8, 8 \ldots 15$	4	101
$-31 \ldots -16, 16 \ldots 31$	5	110
$-63 \ldots -32, 32 \ldots 63$	6	1110
$-127 \ldots -64, 64 \ldots 127$	7	11110
$-255 \ldots -128, 128 \ldots 255$	8	111110
$-511 \ldots -256, 256 \ldots 511$	9	1111110
$-1023 \ldots -512, 512 \ldots 1023$	10	11111110
$-2047 \ldots -1024, 1024 \ldots 2047$	11	111111110

Table 1: Huffman codes for representing each possible value of m for the differential DC coefficient $DIFF(k)$. This mapping results in the VLC encoding.

2. Use Table 1 to encode the value of m. This is referred to as the VLC.

3. If $DIFF(k) \geq 0$, then take the m least significant bits of $DIFF(k)$. If $DIFF(k) < 0$, then form the 2's complement representation of $DIFF(k) - 1$ and take the m least significant bits. This step is referred to as VLI encoding. It is done to efficiently represent the coefficient since we know that $|DIFF(k)| \leq 2^m - 1$.

   For example when $m = 2$, $DIFF(k)$ must be in the set $\{-3, -2, 2, 3\}$. Since the values $\{-1, 0, 1\}$ are not possible, we should remap the integer value to the range $\{0, 1, 2, 3\}$ so as not to waste bits.

   Consider the case when $DIFF(k) = -3$. In this case, $m = 2$. Since $DIFF(k) < 0$, form the 8-bit 2's complement representation of $DIFF(k) - 1$ which is 11111100. Taking the m least significant bits results in 00.

   Alternatively if $DIFF(k) = 3$, then the 8-bit 2's complement representation of $DIFF(k)$ is 00000011; so the 2 least significant bits are 11.

   In practice, 16-bit 2's complement arithmetic should be used to accommodate the 12 bit signed values of $DIFF(k)$.

The result of encoding is then represented as a string of bits first containing the VLC part, and then the VLI part.

The AC coefficients are encoded in a manner similar to the encoding of $DIFF(k)$, but the encoding method for the AC coefficients must be modified to efficiently represent the long strings of zeros that result from quantization of the AC coefficients. Since each run of zeros must terminate in a particular nonzero value, the run length may be encoded as a pair (i, m) where i is the number of preceding zeros and m is the number of bits in the terminal value of the run. So, for example, the sequence $0, 0, 0, -3$ may be represented by $(i, m) = (3, 2)$. This leads to some special cases. The value $(15, 0)$ corresponds to a run of 16 zeros and is denoted by the term ZRL or zero run length. The ZRL symbol is used when a run with more then 16 zeros occurs. Another special case is when the run of zeros extends

Bit Size	AC coefficient value range
0	0
1	$-1, 1$
2	$-3, -2, 2, 3$
3	$-7 \ldots -4, 4 \ldots 7$
4	$-15 \ldots -8, 8 \ldots 15$
5	$-31 \ldots -16, 16 \ldots 31$
6	$-63 \ldots -32, 32 \ldots 63$
7	$-127 \ldots -64, 64 \ldots 127$
8	$-255 \ldots -128, 128 \ldots 255$
9	$-511 \ldots -256, 256 \ldots 511$
10	$-1023 \ldots -512, 512 \ldots 1023$

Table 2: AC coefficient magnitude category for bit size

3	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	9	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Table 3: Example of a quantized DCT block

to the end of the JPEG block. In this case, the $(0, 0)$ pair is used to indicate an end of block or EOB condition. The EOB symbol can be interpreted as an "escape" symbol indicating that no more non-zero values occur in the zig-zag scanned sequence.

Each pair of values (i, m) is encoded using a Huffman table. Appendix B contains a complete table of the typical Huffman codes (sometimes referred to as the default Huffman table) used for the AC coefficients. The JPEG standard also allows the user to specify a different Huffman code if desired; however, the JPEG header routines that we will provide you do not support this flexibility.

For each run-length, the AC coefficients are encoded as follows:

1. Determine a pair (i, m) to represent the run. If there is no run of zeros, simply use $(0, m)$ to represent the nonzero AC coefficient.
2. Map each terminal AC coefficient to an m bit sequence denoted by VLI_{ac}.
3. Use the table of Appendix B to map the pair (i, m) to a Huffman code denoted by VLC_{ac}.
4. Form the final code by concatenating the VLC and VLI representations, $[VLC_{ac}, VLI_{ac}]$.

For example, let's assume that we are to encode a quantized DCT block as shown in Table 3. Also assume that it is the first block in the image. Then, the encoded bit streams have the form:

`‘011’`	$\leftarrow$	$m = 2$ VLC code for DC *DIFF*
`‘11’`	$\leftarrow$	*value* $= 3$ VLI code for DC *DIFF*
`‘11111111001’`	$\leftarrow$	VLC_{ac} of *ZRL*
`‘11111111001’`	$\leftarrow$	VLC_{ac} of *ZRL*
`‘11111111001’`	$\leftarrow$	VLC_{ac} of *ZRL*
`‘111111110100’`	$\leftarrow$	VLC_{ac} of pair (2,4) for AC
`‘1001’`	$\leftarrow$	VLI_{ac} for AC *value* $= 9$
`‘1010’`	$\leftarrow$	VLC_{ac} of *EOB*

When concatenated this codes for the following bit stream.

$$\underbrace{\text{‘011’ ‘11’ ‘111}}_{7F} - \underbrace{\text{11111001’}}_{F9} - \underbrace{\text{‘11111111}}_{FF} - \underbrace{\text{001’ ‘11111}}_{3F} - \underbrace{\text{111001’ ‘11}}_{E7} - \underbrace{\text{11111101}}_{FD} - \underbrace{\text{00’ ‘1001’ ‘10}}_{26} - \underbrace{\text{10’}xxxxxx}_{??}$$

The "-" signs denotes the byte boundaries, and the single quotes denote groups of bits associated with a encoded code word. As you may notice, the bit stream does not always end on a byte boundary. If the given block is not the last block in the image, the encoded bits for next block will be concatenated to the tail of the bit stream for the current block. If the block is the last JPEG block, we add additional zero bits to the bit stream to ensure that there are an integer number of bytes. This process is called *zero padding.*

The second line denotes the hexadecimal value associated with each byte. Notice that the value *0xFF* occurs in the sequence. However, this value is reserved for the JPEG header. So to prevent misinterpretation as a reserved header, any occurrence of the value *0xFF* must be followed by an inserted sequence of *0x00*. This is called *byte stuffing*. Taking the details into consideration, we will get the following output byte sequence for the encoded JPEG block. The actual value for the last byte will be determined after encoding of the next block.

7F F9 FF 00 3F E7 FD 26 ??

3.1 Experiment

In this section, you will implement a C program conforming to the *VLC* and *VLI* coding scheme for the quantized DC and AC coefficients in 8x8 DCT blocks. You should first download the *JPEG utilities* from the laboratory home page. This includes a variety of C header files and subroutines that you will need. It also include the main program for your coder. When writing the code *you should not use any global variables* other then those defined in the *Htables.h* header file. Global variables usually result from a poor programming style, and make debugging and maintaining code much more difficult.

1. First download the *JPEG utilities* from the laboratory home page. This zip file includes the files *Htables.h* and *JPEGutil.c*. The file *Htables.h* contains the Huffman code tables for VLC encoding of both the DC and AC coefficients. The file *JPEGutil.c* contains subroutines for creating JPEG header and tail file structures.

2. The file *Htables.h* defines two C variables. They are complex data structures containing information about the DC and AC Huffman code tables.

   The first data structure, *dcHuffman*, has two components. Let m be the number of bits in the unsigned portion of $DIFF(k)$, then these two components are defined by:

   dcHuffman.size[m] (type int) - Number of bits in the Huffman codeword for a value of $DIFF(k)$ with a bitsize of m.

   dcHuffman.code[m][k] (type char) - An ASCII character containing the k^{th} bit of the Huffman codeword for a bitsize of m. Each ASCII character is either "0" or "1" depending on the bit's value.

   The second data structure, *acHuffman*, has two components. Let (i, m) be the integer pair corresponding to the run length and the bitsize as described above. Then these two components are defined by

   acHuffman.size[i][m] (type int) - Number of bits in the Huffman codeword for run length (i, m).

   acHuffman.code[i][m][k] (type char) - An ASCII character containing the k^{th} bit of the Huffman codeword for run length (i, m). Each ASCII character is either "0" or "1" depending on the bit's value.

3. Write a C subroutine *BitSize* with the following structure

```
int bitsize;
int value;
bitsize = BitSize(value);
```

   where

   - *bitsize* - An integer containing the value m that specifies the position of the most significant bit in the unsigned value.
   - *value* - The integer input.

4. Write a C subroutine *VLI_encode* with the following structure

```
int bitsize;
int value;
char *block_code;
void  VLI_encode(bitsize, value, block_code);
```

   where

   - *bitsize* - The value returned by *BitSize(value)* representing the number of bits in *value*.
   - *value* - The input value to be VLI encoded.

- *block_code* - This pointer is used both to pass input to the routine, and pass back output from the routine. *The input* is any valid ASCII character string. *The output* also an character string with the VLI code appended to the end of the string. As with any C character string, the terminal delimiter must be the NULL symbol (0x00).

  Typically, the input to this subroutine is an incomplete binary encoding of a JPEG block. The encoding of the next coefficient is then appended to the input string as "0" and "1" characters. The new VLI code is added with most significant bit first.

  Notice that the original character string *block_code* must have enough memory allocated to support the maximum possible length output binary sequence. We recommend that you allocate a minimum of $2^{13} = 8192$ bytes for this array.

5. Write a C subroutine *ZigZag* with the following structure

```
int zigzag[64];
int **img;
int i0,j0;
void = ZigZag(img, i0, j0, zigzag);
```

where

- *zigzag* - A 1-D array of 64 integers containing the DCT coefficients for the JPEG block starting at position $(i0, j0)$ in zig zag order. The memory for this array should be allocated before calling the *ZigZag* routine. You will use the variable *Zig* defined in the *Htables.h* to implement this subroutine.
- *img* - The full set of DCT coefficients read in from the Matlab output file.
- *i0* - The row number of the JPEG block.
- *j0* - The column number of the JPEG block.

6. Write a C subroutine *DC_encode* with the following structure

```
int dc_value;
int prev_value;
char *block_code;

void   DC_encode(dc_value, prev_value, block_code);
```

where

- *dc_value* - DC coefficient value in current JPEG block.
- *prev_value* - DC coefficient value in previous JPEG block in raster order.

- *block_code* - This pointer is used both to pass input to the routine, and pass back output from the routine. *The input* is any valid ASCII character string. *The output* also an character string with the binary code for the DC coefficient appended to the end of the string. As with any C character string, the terminal delimiter must be the NULL symbol (0x00).

  Typically, the input to this subroutine is an character string that will be used to store the binary encoding of a JPEG block. This input character string may be empty, or it may contain binary characters from the previous JPEG block that have not yet been written out to the JPEG file.

  The encoding of the DC coefficient is appended to the input string as "0" and "1" characters. The DC characters string is generated corresponding to the output binary sequence produced by encoding $DIFF(k)$. The most significant bit should be first, with the VLC code followed by the VLI code. Use the data structure *dcHuffman* and the subroutine *VLI_encode(bit size, value)* to implement this subroutine.

  Notice that the original character string *block_code* must have enough memory allocated to support the maximum possible length output binary sequence. We recommend that you allocate a minimum of $2^{13} = 8192$ bytes for this array.

7. Write a C subroutine *AC_encode* with the following structure

   ```
   int *zigzag;
   char *block_code;
   void  AC_encode(zigzag, block_code);
   ```

   where

   - *zigzag* - The 1-D array of 64 DCT coefficients in zig zag ordering.
   - *block_code* - This pointer is used both to pass input to the routine, and pass back output from the routine. *The input* is any valid ASCII character string. *The output* also an character string with the binary code for the AC coefficients appended to the end of the string. As with any C character string, the terminal delimiter must be the NULL symbol (0x00).

     Typically, the input to this subroutine is an incomplete binary encoding of a JPEG block. The encoding of the AC coefficients is then appended to the input string as "0" and "1" characters.

     The AC characters string is generated corresponding to the output binary sequence produced by encoding the AC coefficients in zigzag ordering. The most significant bit should be first, with the VLC code followed by the VLI code for each run length. Use the data structure *acHuffman* and the subroutine *VLI_encode(bit size, value)*. As with any C character string, the terminal delimiter must be the NULL symbol (0x00).

Below is a pseudo-code example of how the *AC_encode* subroutine should be structured.

```
AC_encode(zigzag, block_code) {
    /* Init variables */
    int idx = 1 ;
    int zerocnt = 0 ;
    int bitsize ;

    while( idx < 64 ) {
        if( zigzag[idx] == 0 ) zerocnt ++ ;
        else {
            /* ZRL coding */
            for( ; zerocnt > 15; zerocnt -= 16)
                block_code ← strcat( block_code, acHuffman.code[15][0] );
            bitsize = BitSize( zigzag[idx] ) ;
            block_code ← strcat( block_code, acHuffman.code[zerocnt][bitsize] );
            VLI_encode( bitsize, zigzag[idx], block_code ) ;
            zerocnt = 0 ;
        }
        idx ++ ;
    }
    /* EOB coding */
    if(zerocnt) block_code ← strcat( block_code, acHuffman.code[0][0] );
```

Notice that the subroutine uses the ANSI C subroutine call *strcat*. The *strcat* subroutine is used to concatenate standard C character strings.

8. Write a C subroutine *Block_encode* with the following structure

```
char *block_code;
int *zigzag;
int prev_dc ;
void Block_encode(prev_dc, zigzag, block_code);
```

where

- *prev_dc* - DC coefficient value in previous JPEG block in raster order.
- *zigzag* - The 1-D array of 64 DCT coefficients in zig zag ordering.
- *block_code* - This pointer is used both to pass input to the routine, and pass back output from the routine. *The input* is any valid ASCII character string. *The output* also an character string with the binary code for the entire JPEG block. As with any C character string, the terminal delimiter must be the NULL symbol (0x00).

  This routine simply calls *DC_encode* followed by *AC_encode* to encode the entire block of quantized DCT coefficients.

9. Write a C subroutine *Convert_encode* with the following structure

```
unsigned char *byte_code;
char *block_code;
int length;
length = Convert_encode(block_code,byte_code);
```

where

- *block_code* - This pointer is used both to pass input to the routine, and pass back output from the routine. *The input* is a character string containing the binary encoding of one or more, complete or partial JPEG blocks. This character string is produced by the *Block_encode* subroutine. *The output* is a character string containing binary characters that have not yet been encoded into the *byte_code* array. In general, the returned string will be of length < 8 and will contain the trailing bits that would not completely fill a full byte.

  The array *block_code* should be allocated outside the *Block_encode* subroutine, and should contain a minimum of $2^{13} = 8192$ elements so that the array is guaranteed not to overflow.
- *byte_code* This is the converted output byte sequence produced by mapping the characters of *block_code* to the bits of unsigned characters. This output **must** include byte stuffing in the final byte sequence. The memory for this array should be allocated once in the main program as an array of 1024 unsigned characters since this is larger than the maximum possible length of the byte sequence.
- *length* - The number of bytes in the array *byte_code*.

10. Write a C subroutine *Zero_pad* with the following structure

```
char *block_code;
unsigned char byte_value;
byte_value = Zero_pad(block_code);
```

where

- *block_code* - A character string containing the remaining bytes after the last JPEG block has been encoded. This string must have length greater than 0 and less than 8. This character string is produced by the *Convert_encode* subroutine.
- *byte_value* - This is the converted output byte produced by padding additional zeros to *block_code*.

This routine is used only one time for the last JPEG block.

11. The C main program *JPEG_encode* is included in the JPEG utilities you have downloaded. This main routine has been written for you to handle input from command line and reading the Matlab file DCT coefficients file. The main program has been written so that it has the following command line structure.

```
JPEG_encode <Quant scale factor> <matlab file name> <output file name>
```

where *Quant scale factor* is defined as γ in eq (1).

This main program calls the C subroutine *JPEG_encode* to perform the JPEG encoding. This subroutine has the following structure.

```
int **input_img ;
int height;
int width;
FILE *outfp;
void jpeg_encode(input_img,row,column,outfp);
```

where

- *input_img* - Image of DCT coefficients read in from matlab output file.
- *height* - Number of rows in image.
- *width* - Number of columns in image.
- *outfp* - File pointer to output JPEG image.

This subroutine will need to call the subroutine *put_header* at the beginning and *put_tail* at the end. These two routines write out complex binary header and trailer information that is necessary for a standard JPEG decoder to interpret your data. For details on their function refer to Appendix A. Right before *put_tail*, don't forget to call *Zero_pad* routine.

The call structure of these two routines are as follows:

```
int width;
int height;
int quant[8][8];
FILE *fileout;

void put_header(width,height,quant,fileout);
```

where

- *width* - Number of columns in source image.
- *height* Number of rows in source image.
- *quant* - The 8x8 quantization matrix defined in *Htables.h*. This variable **should** be the same as the variable used in section 2.1. That is, you need to scale the quant matrix properly as in eq (1).
- *fileout* - Output file pointer.

```
FILE *fileout;

void put_tail(fileout);
```

where

- *fileout* - Output file pointer.

12. Generate an output JPEG image for $\gamma = 0.25$, 1, and 4. For each of the three output JPEG images, read them into *xv* and print out the images.

13. Congratulations! - Your are done. Go to sleep.

Section 3 Report:
Do the following:

1. Hand in C code for the subroutines *BitSize*, *VLI_encode*, *ZigZag*, *DC_encode*, *AC_encode*, *Block_encode*, *Convert_encode*, and *Zero_pad*.
2. Hand in C code for your main program *JPEG_encode*.
3. Email the encoded image using $\gamma = 1$ as an attachment to the course TA.
4. Hand in the three printouts from xv.

References

[1] ISO/IEC 10918-1,1993(E)

[2] G.K.Wallace. The JPEG still picture compression standard. *Communications of the ACM*, Vol. 34, No.4:30-44,April 1991.

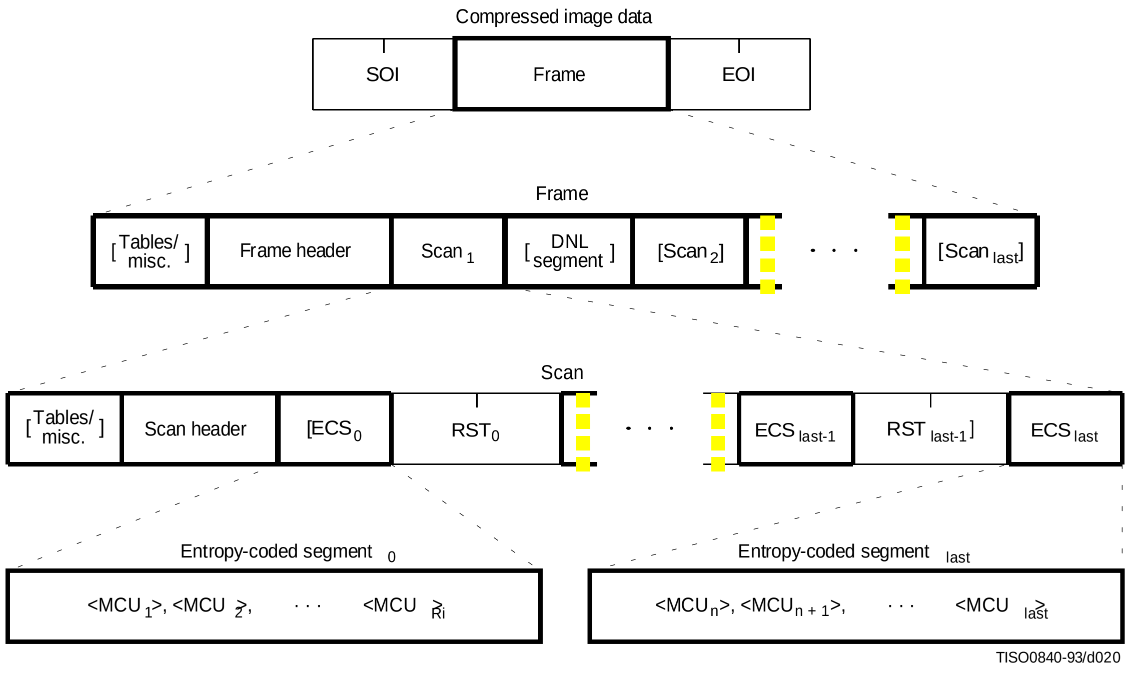


Figure 5: Syntax for sequential DCT-based operation

A Basic JPEG Header Formats

This appendix describes the structure of a simple JPEG header. The header includes both the quantization table and the Huffman coding tables for the DC and AC coefficients. However, the Huffman tables are themselves in coded form, and we do not explain the details of this coding process.

A.1 Start of Image

Figure 5 shows the general structure of a JPEG file. Each JPEG file starts with a special 16-bit sequence called start of image (SOI). This sequence indicates the beginning of an image, and the start of a frame.

SOI: (16 bit) Start of image is given by SOI(0xFFD8).

A.2 Quantization Table

The first element of a frame is the quantization table. The quantization table starts with the *define quantization table* (DQT) symbol. This is then followed by a sequence of symbols that gives specific information about the type of quantization and its values.

DQT (16 bit) Define Quantization table symbol is 0xFFDB.

Lq: (16 bit) Quantization table length.

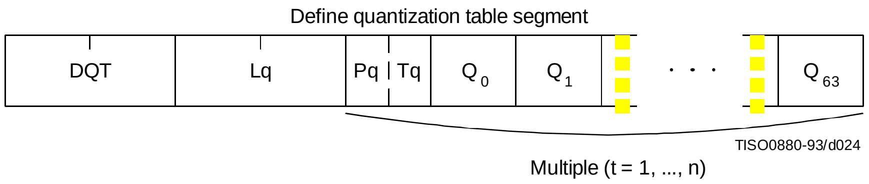


Figure 6: Quantization table syntax

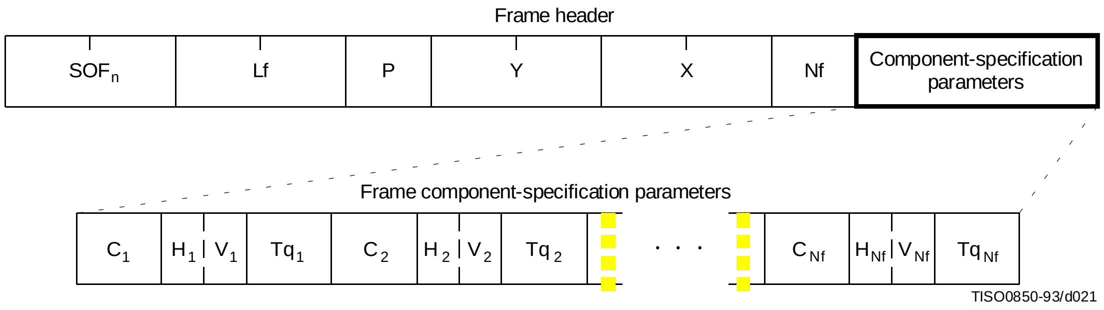

Frame header
SOFn
Lf
P
Y
X
Nf
Component-specification parameters
Frame component-specification parameters
C1
H1
V1
Tq1
C2
H2
V2
Tq2
CNf
HNf
VNf
TqNf
TISO0850-93/d021

Figure 7: Frame Header syntax

Pq: (4 bit) Table element Qk's precision. $'0' = 8$ bit, $'1' = 16$ bit.

Tq: (4 bit) Quantization table destination identifier.

Qk: (8 bit) Quantization table elements in zig-zag scan order.

We use a typical quantization table in Figure 3 for the luminance component. For baseline DCT JPEG, the quantizer table is denoted as follows:

```
FF DB 00 43 00
10 0B 0C 0E 0C 0A 10 0E 0D 0E 12 11 10 13 18 28
1A 18 16 16 18 31 23 25 1D 28 3A 33 3D 3C 39 33
38 37 40 48 5C 4E 40 44 57 45 37 38 50 6D 51 57
5F 62 67 68 67 3E 4D 71 79 70 64 78 5C 65 67 63
```

A.3 Frame Header

The frame header starts after the quantization table. Generally, the frame contains some tables, headers and scan segments. In the baseline encoder, we will use only one scan segment, one luminance quantization table, and two Huffman tables, one each for the DC and AC coefficients.

The frame header specifies the source image characteristics, and encoded component specific parameters. Baseline DCT JPEG is designated by a start of frame zero (SOF0) marker.

SOF0 (16 bit) The start of frame zero maker is 0xFFC0.

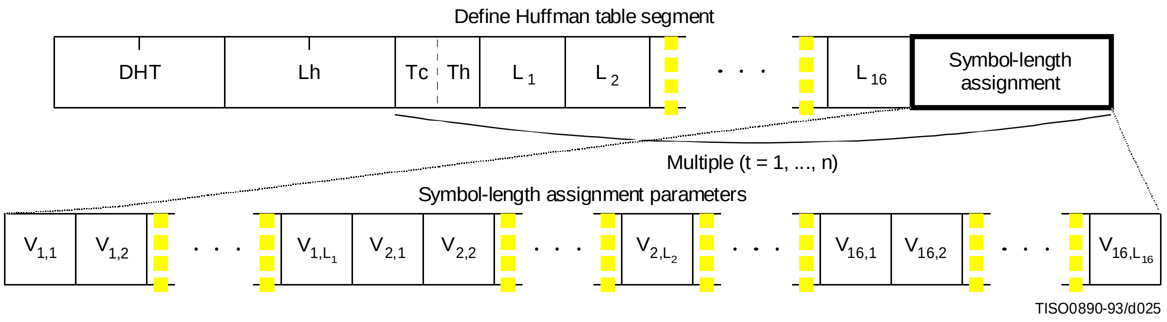


Figure 8: Huffman table syntax

Lf: (16 bit) Frame header length in bytes.

P: (8 bit) Bits/Sample precision.

Y: (16 bit) Number of lines in the source image.

X: (16 bit) Number of samples in one line.

Nf: (8 bit) Number of image component in the frame.

C_1**:** (8 bit) Component identifier label.

H_1**:** (4 bit) Horizontal sampling factor.

V_1**:** (4 bit) Vertical sampling factor.

Tq_1**:** (8 bit) Quantization table destination selector.

For example, with a 512x768 luminance only source image, the frame header content looks like following in hexadecimal numbers: FF C0 0B 08 03 00 02 00 01 11 00

A.4 Huffman code Tables

The Huffman tables are located after the frame header. Their syntax is shown in Figure 8. Like the quantization table definition, it starts with a define Huffman table (DHT) symbol. The structure of the Huffman table is given by:

DHT (16 bit) The define Huffman table symbol is 0xFFC4.

Lh: (16 bit) Huffman table definition length

Tc: (4 bit) Table class, 0 = DC table, 1 = AC table

Th: (4 bit) Huffman table destination identifier

L_i**:** (8 bit) Number of Huffman codes of length i. $(1 \leq i \leq 16)$

$V_{i,j}$**:** (8bit) Value associated with each Huffman code.

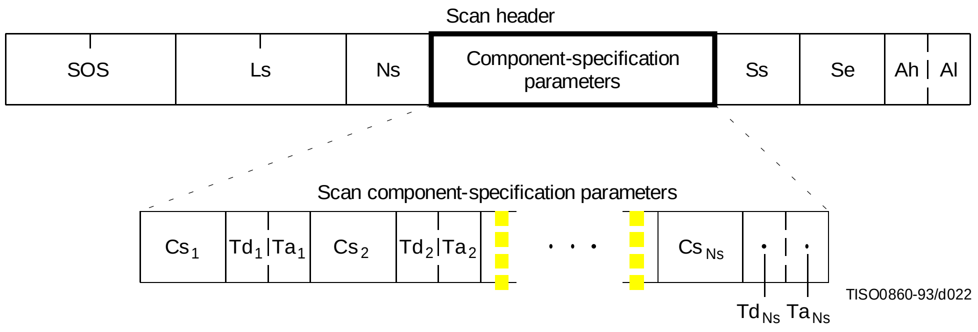


Figure 9: Scan Header Syntax

Table 1 content is converted into a DHT segment. A DC Huffman table example for a typical Huffman code is shown below

```
FF C4 00 1F 00
00 01 05 01 01 01 01 01 01 00 00 00 00 00 00 00
00 01 02 03 04 05 06 07 08 09 0A 0B
```

The first row shows the DHT to Th field. The second row represents $L_1 \ldots L_{16}$. Since L_3 value is 05, it is interpreted as that there are 5 codes of length 3 bit in the DC Huffman code table, and they are {01 02 03 04 05} in the third row. They corresponds to Table 1. For AC Huffman table definition example, refer to [1] Annex K.3.3.2. The only difference is that they need 162 $V_{i,j}$ entries.

A.5 Scan Segment

This segment is the last parameter segment before the actual encoded data appear. It specifies which component was coded and which Huffman tables were used. The following parameters are to be defined.

SOS (16 bit) Start scan segment code given by 0xFFDA.

Ls: (16 bit) Scan header length.

Ns: (8 bit) Number of image components in this scan segment.

Cs_j**:** (8 bit) scan component selector.

Td_j**:** (4 bit) DC Huffman table destination selector.

Ta_j**:** (4 bit) AC Huffman table destination selector.

Ss**:** (8 bit) Start of spectral selection. Specify the first DCT coefficient in zig-zag order,to be coded.

Se: (8 bit) End of spectral selection. Specify the last DCT coefficient in zig-zag order, to be coded.

Ah: (4 bit) Set to zero in Sequential DCT.

Al: (4 bit) Set to zero in Sequential DCT.

The following example shows the bit stream from the SOS to the EOI symbols for an achromatic baseline DCT JPEG. The symbols 'zz ww yy . . . ' represent the Huffman codes for the coefficients of the first 8×8 DCT block.

```
FF DA 00 08       % Start of Scan marker
01 01 00 00 3f 00
zz ww yy . . .    % encoded data stream
. . .
. . .
FF D9             % EOI marker
```

A.6 End of Image

The JPEG file is ended by the 16-bit end of image (EOI) symbol given by 0xFFD9.

B Typical AC Huffman Table

Run/Size	Code length	Code word
0/0 (EOB)	4	1010
0/1	2	00
0/2	2	01
0/3	3	100
0/4	4	1011
0/5	5	11010
0/6	7	1111000
0/7	8	11111000
0/8	10	1111110110
0/9	16	1111111110000010
0/A	16	1111111110000011
1/1	4	1100
1/2	5	11011
1/3	7	1111001
1/4	9	111110110
1/5	11	11111110110
1/6	16	1111111110000100
1/7	16	1111111110000101
1/8	16	1111111110000110
1/9	16	1111111110000111
1/A	16	1111111110001000
2/1	5	11100
2/2	8	11111001
2/3	10	1111110111
2/4	12	111111110100
2/5	16	1111111110001001
2/6	16	1111111110001010
2/7	16	1111111110001011
2/8	16	1111111110001100
2/9	16	1111111110001101
2/A	16	1111111110001110
3/1	6	111010
3/2	9	111110111
3/3	12	111111110101
3/4	16	1111111110001111
3/5	16	1111111110010000
3/6	16	1111111110010001
3/7	16	1111111110010010
3/8	16	1111111110010011
3/9	16	1111111110010100
3/A	16	1111111110010101

Table 4: Typical Huffman table for AC coefficients(sheet 1 of 4)

Run/Size	Code length	Code word
4/1	6	111011
4/2	10	1111111000
4/3	16	1111111110010110
4/4	16	1111111110010111
4/5	16	1111111110011000
4/6	16	1111111110011001
4/7	16	1111111110011010
4/8	16	1111111110011011
4/9	16	1111111110011100
4/A	16	1111111110011101
5/1	7	1111010
5/2	11	11111110111
5/3	16	1111111110011110
5/4	16	1111111110011111
5/5	16	1111111110100000
5/6	16	1111111110100001
5/7	16	1111111110100010
5/8	16	1111111110100011
5/9	16	1111111110100100
5/A	16	1111111110100101
6/1	7	1111011
6/2	12	111111110110
6/3	16	1111111110100110
6/4	16	1111111110100111
6/5	16	1111111110101000
6/6	16	1111111110101001
6/7	16	1111111110101010
6/8	16	1111111110101011
6/9	16	1111111110101100
6/A	16	1111111110101101
7/1	8	11111010
7/2	12	111111110111
7/3	16	1111111110101110
7/4	16	1111111110101111
7/5	16	1111111110110000
7/6	16	1111111110110001
7/7	16	1111111110110010
7/8	16	1111111110110011
7/9	16	1111111110110100
7/A	16	1111111110110101

Table 5: Typical Huffman table for AC coefficients(sheet 2 of 4)

Run/Size	Code length	Code word
8/1	9	111111000
8/2	15	111111111000000
8/3	16	1111111110110110
8/4	16	1111111110110111
8/5	16	1111111110111000
8/6	16	1111111110111001
8/7	16	1111111110111010
8/8	16	1111111110111011
8/9	16	1111111110111100
8/A	16	1111111110111101
9/1	9	111111001
9/2	16	1111111110111110
9/3	16	1111111110111111
9/4	16	1111111111000000
9/5	16	1111111111000001
9/6	16	1111111111000010
9/7	16	1111111111000011
9/8	16	1111111111000100
9/9	16	1111111111000101
9/A	16	1111111111000110
A/1	9	111111010
A/2	16	1111111111000111
A/3	16	1111111111001000
A/4	16	1111111111001001
A/5	16	1111111111001010
A/6	16	1111111111001011
A/7	16	1111111111001100
A/8	16	1111111111001101
A/9	16	1111111111001110
A/A	16	1111111111001111
B/1	10	1111111001
B/2	16	1111111111010000
B/3	16	1111111111010001
B/4	16	1111111111010010
B/5	16	1111111111010011
B/6	16	1111111111010100
B/7	16	1111111111010101
B/8	16	1111111111010110
B/9	16	1111111111010111
B/A	16	1111111111011000

Table 6: Typical Huffman table for AC coefficients(sheet 3 of 4)

Run/Size	Code length	Code word
C/1	10	1111111010
C/2	16	1111111111011001
C/3	16	1111111111011010
C/4	16	1111111111011011
C/5	16	1111111111011100
C/6	16	1111111111011101
C/7	16	1111111111011110
C/8	16	1111111111011111
C/9	16	1111111111100000
C/A	16	1111111111100001
D/1	11	11111111000
D/2	16	1111111111100010
D/3	16	1111111111100011
D/4	16	1111111111100100
D/5	16	1111111111100101
D/6	16	1111111111100110
D/7	16	1111111111100111
D/8	16	1111111111101000
D/9	16	1111111111101001
D/A	16	1111111111101010
E/1	16	1111111111101011
E/2	16	1111111111101100
E/3	16	1111111111101101
E/4	16	1111111111101110
E/5	16	1111111111101111
E/6	16	1111111111110000
E/7	16	1111111111110001
E/8	16	1111111111110010
E/9	16	1111111111110011
E/A	16	1111111111110100
F/0 (ZRL)	11	11111111001
F/1	16	1111111111110101
F/2	16	1111111111110110
F/3	16	1111111111110111
F/4	16	1111111111111000
F/5	16	1111111111111001
F/6	16	1111111111111010
F/7	16	1111111111111011
F/8	16	1111111111111100
F/9	16	1111111111111101
F/A	16	1111111111111110

Table 7: Typical Huffman table for AC coefficients(sheet 4 of 4)