PVRG-JPEG CODEC 1.1

Andy C. Hung

November 17, 1993

This package is based on the UNIX operating system. We appreciate any comments or corrections - they can be sent to Andy C. Hung at achung@cs.stanford.edu. We cannot guarantee that any bugs will be corrected however.

Funded by the Defense Advanced Research Projects Agency.
I am especially grateful to Hewlett Packard and Storm Technology for their financial support during the earlier stages of codec development. Any errors in the code and documentation are my own. The following people are acknowledged for their advice and assistance. We apologize if we have missed anyone from this list. Thanks, one and all.

The Portable Video Research Group at Stanford: Teresa Meng, Peter Black, Navin Chaddha, Ben Gordon, Sheila Hemami, Wee-Chiew Tan, Eli Tsern.

Adriaan Ligtenberg of Storm Technology. Jeanne Wiseman, Andrew Fitzhugh, Gregory Yovanof, and Chuck Rosenberg of Hewlett Packard. Eric Hamilton and Jean-Georges Fritsch of C-Cube Microsystems.

Lawrence Rowe of the Berkeley Plateau Research Group.

Tom Lane of the Independent JPEG Group.

Thierry Turletti, Inria.

Katsumi Tahara, Sony; Ciaran Mc Goldrick; Karl Lillevold; Mats Lofkvist; Hiroshi Saitoh, Panasonic; Frank Gadegast; Chad Fogg, Cascade; Anders Klemets; Graham Logan; Jelle van Zeijl; George Warner, AT&T; Chris Adams, Hewlett Packard.

Dedicated to the standardization committees and their efforts in bringing the advancement of technology to everyone.

Contents

Chapter 1

Getting Started

This document describes the Portable Research Video Group's (PVRG) PVRG-JPEG software codec. The initial version of this codec was developed in 1990, based on the early JPEG specifications. The PVRG-JPEG software codec that is described here is based on the JPEG ISO/IEC CD 10198-1 draft. Since 1991, very little development on the PVRG-JPEG software codec has occurred. We should caution the user that bugs are probably still lurking.

1.1 Common Questions and Answers

Let's review a few common questions about JPEG.

What is JPEG? It is an acronym for Joint Photographic Experts Group,. In common use, it refers to a definition of a still-image compression algorithm established by the JPEG committee.

How does JPEG differ from Group 3 and 4 Fax? The JPEG algorithm compresses continuous tone images such as grayscale and color. Group 3 and 4 Fax methods compress black-and-white images, sometimes called bi-level images.

How does JPEG differ from H.261 or MPEG? JPEG's application is transmission of single independent images, often called "still-images". H.261, or p*64 as it is sometimes referred to, is CCITT's (Comité Consultatif International Télégraphique et Téléphonique) method for compressing image sequences, which are highly correlated, such as those found in teleconferencing. MPEG's application is high-rate image sequence compression.

What is the JPEG Baseline system? The baseline system is the minimal system necessary to claim "JPEG compatibility". The extended system is used for special applications requiring unique features, for example, progressive buildup of quality and lossless coding.

What does this particular package do? The JPEG package presented here is a fairly complete implementation of the proposed JPEG baseline system, based on specification JPEG committee draft 10198-1. By complete, we mean that this system has a very flexible encoder that can generate compressed files through different valid strategies; we believe that the decoder can decode all possible JPEG baseline compressed files.

Where can I obtain this codec? The PVRG-JPEG codec can be obtained from anonymous ftp from havefun.stanford.edu:pub/jpeg/JPEGv1.2.tar.Z.

What distinguishes this package from other available systems? We offer a fairly complete JPEG baseline package for public domain use, targeted toward simulation and research. This package should be able to function with all of the JPEG baseline system. An excellent public domain JPEG software system targeted toward library integration and particular files possessing JFIF (a common JPEG interchange format) compatibility is available from the Independent JPEG group. It can be obtained from ftp.uu.net in the directory graphics/jpeg.

What special features are in this implementation package? This implementation can also generate a superset of the baseline system. One use, for example, is to exploit different subsampling and interleaving which cannot be met by a baseline system.

What kind of system do I need? Pure storage memory requirement (excluding the size of code) for the the entire possible JPEG-header operating range is 1.5 Mbyte. For strict baseline operation, the memory requirement is 0.1 Mbyte.

1.2 Further Information

Some references for JPEG are mentioned in the bibliography.

Chapter 2

Understanding Image Compression

2.1 Introduction

The complete picture of the image starts from the color representation. Thus, we begin with analysis of color spaces and then start the examination of transform image coding, quantization, and entropy coding.

2.2 Color Space

Since Newton's time, it has been known that a wide spectrum of colors can be generated from a set of three primaries. An artist, for example, can paint most colors from layering pigments of the subtractive primaries: red, yellow, and blue. Television displays generate colors by mixing lights of the additive primaries: red, green, and blue. Although two primary systems can be used– most generally a red-orange and a green-blue– the image rendered is not lifelike, and two color systems never became successful in either motion picture or pre-standard television[3].

The color space obtained through combining the three colors can be determined by drawing a triangle on a special color chart with each of the base colors as an endpoint. The classic color chart used in early television specifications was established in 1931 by the Commission Internationale de L'Eclairage (CIE).

One of the special concepts introduced by the 1931 CIE chart was the isolation of the luminance, or brightness, from the chrominance, or hue. Using the CIE chart as a guideline, the National Television System Committee (NTSC) defined the transmission of signals in a luminance and chrominance format, rather than a format involving the three color components of the television phosphors. The new color space was labeled YIQ, representing the luminance, in-phase chrominance, and quadrature chrominance coordinates respectively.

In Europe, two television standards were later established, the phase-alternation-line (PAL) format and the Séquentiel couleurà mémoire (SECAM) format, both with an identical color space, YUV. The only change between the PAL/SECAM YUV color space and the NTSC YIQ color space is a 33 degree rotation in UV space[4].

The digital equivalent of YUV is YCbCr, where the Cr chrominance component corresponds to the analog V component, and the Cb chrominance component corresponds to the analog U component, though with different scaling factors. The codec will use the terms Y, U, and V for Y, Cb, and Cr for a shorthand description of the digital YCbCr color space.

The conversion between the standard Red Green Blue (RGB) format to YCbCr format is slightly different for digital signals than for analog signals. For the JPEG JFIF[13] format convention, the full range of 8 bits is used for Y, Cb, and Cr. The digital conversion insures that if the RGB inputs are between 0 and 255, (normalized so that equal values represents reference white), then the Y output has values between 0-255 and the Cb and Cr have values between 0 and 128. The values are stored as 8 bit unsigned characters, thus the Cb and Cr values are level shifted by adding 128 so that the values are always non-negative, and if Cb and Cr equal 256, they are clamped to 255. The conversion, written in a matrix form, is

$$\begin{array}{ccc} 0\ 299 & 0\ 587 & 0\ 114 \\ 0\ 1687 & 0\ 3313 & 0\ 5 \\ 0\ 5 & 0\ 4187 & 0\ 0813 \end{array} \qquad 2\ 1$$

The inverse conversion (after subtracting 128, if the values of Cb and Cr are level-shifted) is

$$\begin{array}{ccc} 1 & 0 & 1\ 402 \\ 1 & 0\ 34414 & 0\ 71414 \\ 1 & 1\ 772 & 0 \end{array} \qquad 2\ 2$$

While it is easily confirmed that the RGB to YCbCr conversion yields values that are automatically between 0-255 for the Y and between 128 for Cb and Cr, conversely not all YCbCr space maps to legitimate RGB space. Thus the values for RGB should be clamped after colorspace conversion to 0 to 255.

The other common YCbCr conversion (indeed the specified conversion for H.261 and MPEG) performs slightly different scaling, by CCIR Recommendation 601 specifications. The CCIR YCbCr nominal range is between 16 to 235 for the luminance and 16 to 240 for the chrominances. Thus this color space is just a rescale and shift on the formulas described above (i.e. Y' = 219/255 * Y + 16, Cb' = 224/255 * Cb + 128, Cr' = 224/255 * Cr + 128 (assuming both Cb and Cr do not have level shifts prior to scaling, but Cb' and Cr' do)). Although the CCIR 601 YCbCr colorspace is specified for H.261, some streams may be generated with JFIF color components.

The YCbCr format, similar to the YIQ format[5], concentrates most of the image information into the luminance and less in the chrominance. The result is that the YCbCr elements are less correlated and, therefore, can be coded separately without much loss in efficiency.

Another advantage comes from reducing the transmission rates of the Cb and Cr chrominance components. Commonly known from the testing of the NTSC YIQ format and the PAL/SECAM YUV format is that the chrominance need not be specified as frequently as the luminance. Hence, for the JPEG-JFIF algorithm, only every other Cb element and every other Cr element in both the horizontal and vertical directions are sampled. The missing elements can be reconstructed by various means of interpolation, including duplication.

The reduction in data by converting from RGB to YCbCr is 2 to 1 (denoted 2:1). For example, if the RGB format is specified by eight bits for each color, then each RGB picture element is described by 24 bits; and after conversion and decimation, each YCbCr pixel is described by an average of 12 bits: the luminance at eight bits, and both the chrominances for every other pixel (horizontally and vertically) at eight bits.

The printing industry, different from the broadcast industry, has another color space format consisting of the subtractive primaries, CMYK, Cyan, Magenta, Yellow and Black. This uses four colors because black, although technically constructible from CMY, is added to the picture to obtain the best visual quality.

The JPEG specification allows for up to 255 primaries thus covering demanding specifications requiring even more precise spectral representation of color and overlays.

The conversion in color space is usually the first step toward compressing the image. Now we shall consider the next step, the baseline compression model.

2.3 Baseline Model

To satisfy the demands of the commercial market, the baseline method must be able to provide good quality images with compression ratios of at least 16:1 and with a pixel element specification between 4 to 12 bits. In the competition to field the best algorithm, twelve methods were originally proposed, ten techniques were eventually tested, and of these ten, three were selected as the finalists[2].[1] Of the contenders, the Adaptive Discrete Cosine Transform (ADCT) clearly performed better than the others and was unanimously selected to become the algorithm of the JPEG system. In the following sections we will continue discussion of the JPEG baseline (ADCT) algorithm.

[1]From Hudson[2], the ten methods are as follows: Discrete Cosine Transform with Vector Quantization (DCTV), Adaptive Discrete Cosine Transform (ADCT), Adaptive Discrete Cosine Transform and Differential Entropy Coding (DCTD), 16 by 16 DCT with filtering (BCTF), Block List Transform (BLT), Progressive Coding Scheme (PCS), Progressive Recursive Binary Nesting (PRBN), Adaptive Binary Arithmetic Differential Pulse Code Modulation (ABAC), Generalized Block Truncation Coding (GBTC), and Component Vector Quantization (CVQ). The three finalists were the ADCT, ABAC and BSPC methods.

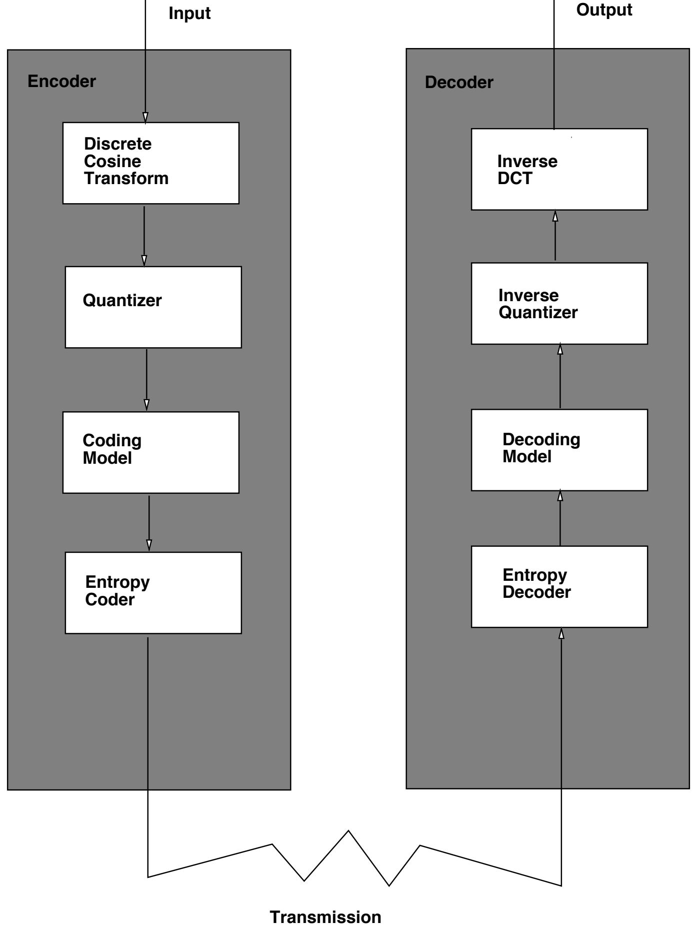


Figure 2.1: The flow of information from the encoder to the decoder for the JPEG baseline system.

The JPEG baseline model, shown in figure 2.1, consists of four stages: a transformation stage, a lossy quantization stage, and two lossless coding stages. The initial transform concentrates the information energy into the first few transform coefficients, the quantizer causes a controlled loss of information, and the two coding stages further compress the data. As discussed earlier, the individual color components in the YCbCr color space are less correlated than in the RGB color space. Therefore, we can apply this model to compress each YCbCr component individually.

The JPEG baseline model is considered a "lossy" compressor because the reconstructed image is not identical to the original. Lossless coders, which create images identical to the original, achieve very poor compression because the least significant bits of each color component become progressively more random and harder to code.

2.4 Transform Stage

For each separate color component, the video image can be broken into 8 by 8 blocks of picture elements which join end to end across the image. Each block typically has video energy scattered throughout the elements, the energy defined by the squared values of each element. If the video energy is of low spatial frequency– slowly varying, then a transform can be used to concentrate the energy into very few coefficients. The transform method chosen by JPEG is the two dimensional 8 by 8 DCT, a transform studied extensively for image compression.

Conceptually, a one dimensional DCT can be thought of as taking the Fourier Transform of an infinite sequence as shown in figure 2.2. This sequence consists of the data vector reflected across the axis and repeated indefinitely along the axes.

The two dimensional DCT can be obtained by performing a one dimensional DCT on the columns and a one dimensional DCT on the rows. An explicit formula[2] for the two dimensional 8 by 8 DCT can be written in

[2]The complexity of the DCT can be simplified tremendously. For example, in the one dimensional 8 element DCT, the multiplications can be reduced to 12, see [6]; and for the two dimensional 8 by 8 case, with scaled DCT's which integrate the multiplications into the

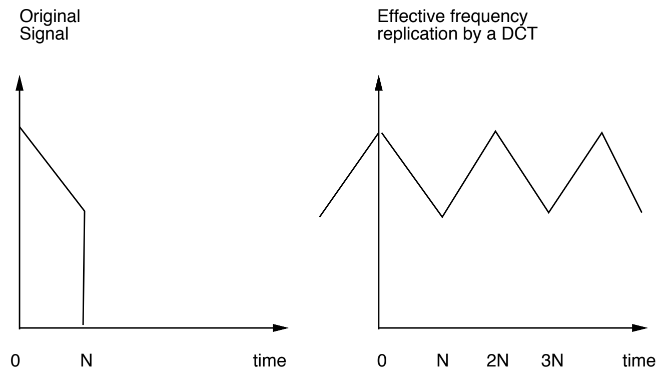


Figure 2.2: The DCT transform. The correspondence between the original windowed signal between 0 to N and the replication performed implicitly by the DCT to fill out times outside of the window.

terms of the pixel values, , and the frequency domain transform coefficients, .

$$1\ 4 \quad \sum_{0}^{7} \sum_{0}^{7} \quad \cos\ 2\quad 1\quad 16\ \cos\ 2\quad 1\quad 16 \qquad 2\ 3$$

where

$$\begin{matrix} 1\quad \overline{2} & 0 \\ 1 & \text{otherwise} \end{matrix} \qquad 2\ 4$$

The transformed output from the 2D DCT will be ordered so that the mean value, the DC coefficient, is in the upper left corner and the higher frequency coefficients progress by distance from the DC coefficient. The higher vertical frequencies represented by higher row numbers and higher horizontal frequencies represented by higher column numbers.

The inverse of the 2D DCT is written as

$$1\ 4 \quad \sum_{0}^{7} \sum_{0}^{7} \quad \cos\ 2\quad 1\quad 16\ \cos\ 2\quad 1\quad 16 \qquad 2\ 5$$

As an example of the DCT, lets take the transform of a pizza. Assume that the pizza has only cheese topping so that the top of the pizza is a uniform color. Also assume that the pizza is infinitely thin (2 dimensional). The 8 by 8 input represented by the symbol O for the outside of the pizza and X for the inside of the pizza is

```
OOOXXOOO
OXXXXXXO
XXXXXXXX
XXXXXXXX
XXXXXXXX
XXXXXXXX
OXXXXXXO
OOOXXOOO
```

If we assign 10 and 10, the pizza looks like

quantization stage, the number of multiplications can be reduced to 54, see [8].

-10	-10	-10	10	10	-10	-10	-10
-10	10	10	10	10	10	10	-10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
10	10	10	10	10	10	10	10
-10	10	10	10	10	10	10	-10
-10	-10	-10	10	10	-10	-10	-10

and by (2.3), the frequency domain representation, rounded to the nearest integer, is

40	0	-26	0	0	0	-11	0
0	0	0	0	0	0	0	0
-45	0	-24	0	8	0	-10	0
0	0	0	0	0	0	0	0
-20	0	0	0	20	0	0	0
0	0	0	0	0	0	0	0
-3	0	10	0	18	0	4	0
0	0	0	0	0	0	0	0

We can conclude that the DCT of a pizza doesn't resemble anything edible. But this example does illustrate the reduction in data caused by the transform: more zero coefficients and a concentration of energy around the upper left corner of the matrix.

Even though the energy distribution has changed, the total energy remains the same because the DCT is a unitary transformation (but because a balanced DCT involves the square root of the data length, the DCT is sometimes defined with unbalanced normalization).

Furthermore, since the DCT is unitary, the maximum value of each 8 by 8 DCT coefficient is limited to a factor of eight times the original values; and after rounding, an eight bit input value can be represented by an eleven bit transformed value. Coincidently, the IDCT of this block, when rounded to the nearest integer, yields the original pizza, but perfectly exact reconstruction is not always possible with an integer value DCT.

2.5 Quantization

The coefficients of the DCT are quantized to reduce their magnitude and to increase the number of zero value coefficients. The uniform midstep quantizer is used for the JPEG baseline method, where the stepsize is varied according to the coefficient location and which color component is encoded. This is shown in figure 2.3. The equations for the quantizer can be written as

2 2 6

where is the quantized coefficient, is the DCT frequency coefficient, and is the quantizer stepsize for the element in the block. The sign indicates a plus for a positive coefficient, , and a minus for a negative coefficient, .

The inverse quantizer is

2 7

Quantization is the lossy stage in the JPEG coding scheme. If we quantize too coarse, we may end up with images that look "blocky," but if we quantize too fine, we may spend useless bits coding (essentially) noise.

We can control the quantization by the Q-Factor, a number which changes the default quantization matrix by an effective multiplicative factor of the Q-Factor divided by 50. If the Q-Factor is 0, we bypass this step. A higher Q-Factor gives better compression, a lower Q-Factor gives a better quality image.

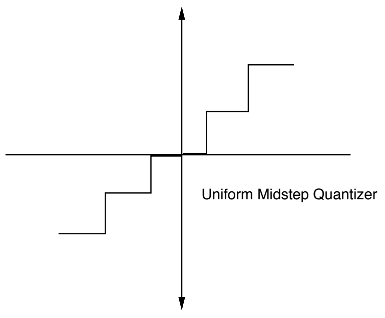


Figure 2.3: The quantizer transfer function.

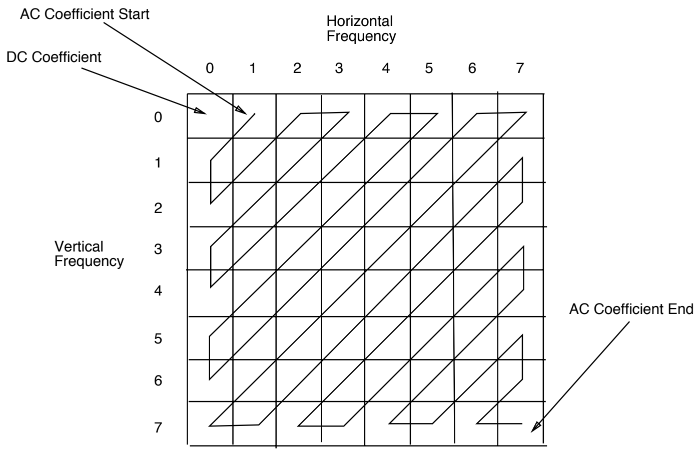


Figure 2.4: The zigzag pattern for reordering the two dimensional DCT coefficients into a one dimensional array.

2.6 Coding Model

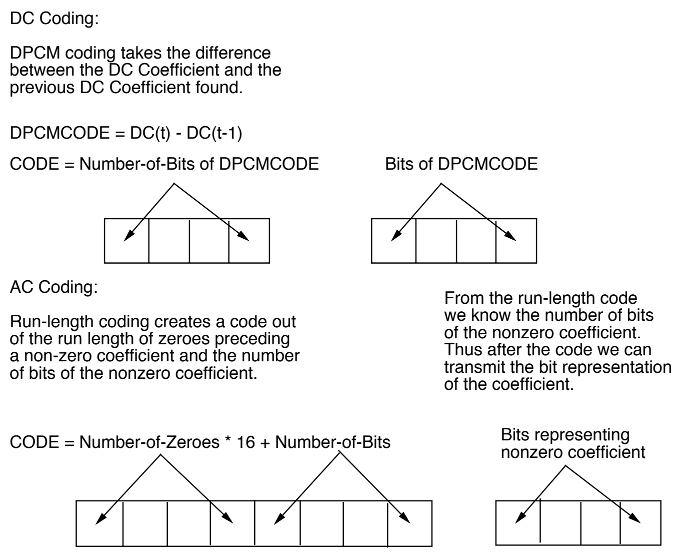
The coding model rearranges the quantized DCT coefficients into a zig-zag pattern, with the lowest frequencies first and the highest frequencies last. The zig-zag pattern is used to increase the run-length of zero coefficients found in the block. The assumption is that the lowest frequencies tend to have larger coefficients and the highest frequencies are, by the nature of most pictures, predominantly zero.

As illustrated in figure 2.4, the first coefficient (0,0) is called the DC coefficient and the rest of the coefficients are called AC coefficients. The AC coefficients are traversed by the zig-zag pattern from the (0,1) location to the (7,7) location.

The image DC coefficients often vary slightly between successive blocks. The coding of the DC coefficient exploits this property through Differential Pulse Code Modulation (DPCM). This technique codes the difference between the quantized DC coefficient of the current block and the quantized DC coefficient of the previous block. For example, consider the *kth* quantized block. The formula for the DPCM code is

$$0\,0 \qquad 0\,0_{1} \qquad\qquad 2\,8$$

The inverse DPCM calculates the current DC coefficient value by summing the current DPCM code with the previous DC coefficient value.


DC Coding:

DPCM coding takes the difference between the DC Coefficient and the previous DC Coefficient found.

DPCMCODE = DC(t) - DC(t-1)

CODE = Number-of-Bits of DPCMCODE

AC Coding:

Run-length coding creates a code out of the run length of zeroes preceding a non-zero coefficient and the number of bits of the nonzero coefficient.

From the run-length code we know the number of bits of the nonzero coefficient. Thus after the code we can transmit the bit representation of the coefficient.

CODE = Number-of-Zeroes * 16 + Number-of-Bits

Figure 2.5: The coding model for the DC coefficient and the AC coefficients.

0 0 0 0 $_1$ 2 9

After calculation of the DPCM code, the actual DC code is then given by the size of bits of the DPCM code followed by the significant bits of the DPCM code.

The quantized AC coefficients usually contain runs of consecutive zeroes. Therefore, a coding advantage can be obtained by using a run-length technique where the upper nybble of the run-length code byte is the number of consecutive zeroes before the next coefficient and the lower nybble of the code byte is the number of significant bits of the next coefficient.

Following the code byte is the significant bits of the next coefficient, the length of which can be determined by the lower nybble of the code byte[3]. Both AC coefficient and DC coefficient coding models are illustrated in figure 2.5.

2.7 Lossless Coding

Some applications require perfect reconstruction of the original signal, such as medical and multiple-generation reproductions. This is provided by JPEG's lossless mode.

The method used by JPEG to provide lossless coding is nearest neighbor prediction followed by Huffman coding. The nearest neighbors are the previous pel, , or the pel on the previous line right above it , and the pel on the previous line and to the left, . This location diagram is shown below

```
C B
A x
```

There are seven different predictors that can be used.

[3]The decoding of a run-length code is a bit complicated.
The inverse run-length coder translates the input coded stream into an output array of AC coefficients. It takes current code and appends a number of zeroes to the output array corresponding to the upper nybble of the run-length code. The next coefficient placed into the output array is has a bit significance determined by the lower nybble of the run-length code, and a value determined by that number of trailing bits.

2

2

2

For the first line and any line directly after the resynchronization interval, the prediction method is always , and the first pel of every such line is predicted by 1/2 the usable pel-range. Otherwise, the first pel of every line is predicted by the pel above it, .

The predictions are then entropy coded as discussed below.

2.7.1 Point Transform

In order to represent a wider range of values, a point transform may be used. Essentially a point transform is a shift of the input and output pel by a predetermined number of bits. This allows for a large bit-per-pel image to be effectively coded as a smaller bit-per-pel image. The point transform may not be used for baseline or extended baseline coding.

2.8 Entropy Coding

The block codes from the DPCM and run-length models can be further compressed using entropy coding. For the baseline JPEG method, the Huffman coder[9] is used to compress the data closer to symbol entropy. One reason for using the Huffman coder is that it is easy to implement in hardware. To compress data symbols, the Huffman coder creates shorter codes for frequently occurring symbols and longer codes for occasionally occurring symbols.

For example, let's encode an excerpt from Michael Jackson's song *Bad*[4].

```
Because I'm bad, I'm bad-- come on
Bad, bad-- really, really bad
You know I'm bad, I'm bad--
you know it
Bad, bad-- really, really bad
You know I'm bad, I'm bad-- come on, you know
Bad, bad-- really, really bad
```

The first step in creating Huffman codes is to create a table assigning a frequency count to each phrase. In the above lyrics, ignoring capitalization, this is shown in table 2.1.

Initially designate all symbols as leaf nodes for a tree. Now starting from the two least weight nodes, aggregate the pair into a new node. For example, in the above frequency chart, *Because* and *It* would be aggregated first. Repeat this process for the new set until the entire symbol set is represented by a single node. The result is shown in figure 2.6.

A Huffman code can be generated for each symbol by attaching a binary digit to each branch. Let's assign the binary digit 1 to the left branches and the binary digit 0 to the right branches. The symbol code is generated by following the path of branches from the top node to the symbol leaf node. In the case of the word *Bad*, the Huffman code is 1, and for the word *I'm*, the Huffman code is 011. A full table for each symbol is shown in table 2.2.

[4]1987 c MJJ Productions, Inc.

Phrase	Symbol	Frequency
Because	b	1
I'm	I	6
Bad	B	15
Come on	C	2
It	i	1
Really	R	6
You know	Y	4

Table 2.1: The frequency of words in the song *Bad*.

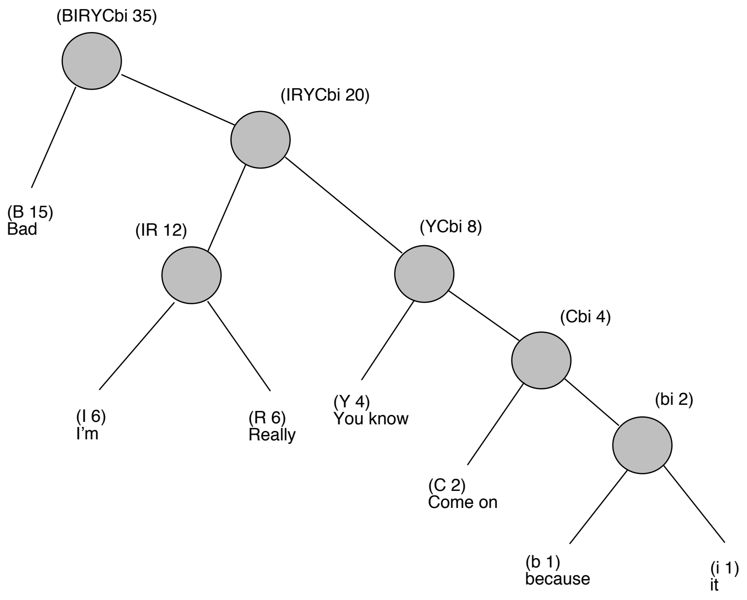


Figure 2.6: The Huffman tree for the lyrics to *Bad*.

Phrase	Symbol	Frequency	Code Length	Code
Because	b	1	5	00001
I'm	I	6	3	011
Bad	B	15	1	1
Come on	C	2	4	0001
It	i	1	5	00000
Really	R	6	3	010
You know	Y	4	3	001

Table 2.2: The Huffman codes for the words in *Bad*.

To encode a symbol, just output the code word to the bit stream. For example, output the code word 010 for *Really*. To decode from a stream of bits, start from the top node, and follow the left branch or right branch depending on the value taken from the bit stream. Continue until a leaf node is reached. The decoded symbol is the symbol associated with that leaf.

In the above example, the first few bits in the output bit stream are (bars are inserted for the reader's convenience)

00001—011—1—011—1—...

Our coding efficiency can be calculated by comparing the number of bits required to realize the lyrics. For the Huffman code above, the bit length is 15(1) + 16(3) + 2(4) + 2(5) = 81 bits. In comparison, for a three bit code, the bit length is 35(3) = 105 bits; for an ideal 7 symbol code, the bit length is $35 \log_2 7 = 98.3$ bits. The Huffman coder compresses the lyrics by about 20 percent; but this figure does not include the cost of transmitting the initial Huffman code table to the decoder.

2.9 Custom and Predefined Huffman Tables

Huffman tables can be predefined from statistics generated from sets of images or calculated custom on an image-by-image basis. Normally, custom Huffman tables reduce the number of bits sent by less than 10 percent over predefined tables. For the system presented here, the default method is to generate custom tables by a two pass system, though, optionally, internally or externally specified tables can be used instead.

2.10 Interleaving

An image can be composed of several components, for example, Red Green, and Blue. If we transfer these components one image color component at a time, we call it noninterleaved mode. This mode, although simpler, would require a frame buffer to store every component, except the last one, because a pixel's complete value is unknown until all of its component values are received. For example, we don't know the first pixel's RGB value without all of the R components and G components received first.

Interleaved mode solves this problem. We take a few blocks of one component, a few blocks of the next component, and so on, until we have blocks from all components. Then we can determine the pixel values from the subset of the image received. That is, supposing we send one Red block, one Green block, one Blue block, one Red block, etc., then after one block trio of RGB, we can determine the full 24 bit value for pixels within that block.

Since some color spaces, by convention, have decimated components, e.g. YCbCr space may decimate Cb and Cr horizontally, vertically, or both, the sampling rate for interleaving must be specified.

For a set of color components with various sampling frequencies, the minimum data unit is defined in terms of the frequency of the blocks. For example, 4:2:2, the YCbCr color file can be specified as (2h1v) Y (1h1v) Cb (1h1v) Cr. That would be

```
Y1 Y2 Cb1 Cr1    Y3 Y4 Cb2 Cr2 ...
```

Now 4:1:1 for YCbCr can be specified as (2h2v) Y (1h1v) Cb (1h1v) Cr. That means the Cb and Cr component would be decimated by a factor of 2, both vertically and horizontally.

```
Y1 Y2            Y5 Y6
Y3 Y4 Cb1 Cr1    Y7 Y8 Cb2 Cr2 ...
```

The sum of each component's horizontal frequency multiplied by the vertical frequency is the Minimum Data Unit (MDU); that is, the number of blocks that form a single unit of transmission.

2 10

0

If your image file is not flush with any of the blocks in the MDU, then the last value on the edge is replicated until the image becomes flush with the MDU block. Thus, in the 4:1:1 decimation example above, if the data only goes to the Y5 block, then Y6 Y7 Y8 are value-replicated extensions of the border in Y5. Very large frequency sampling ratios are not desirable because we code unnecessary blocks, such as in the above example, Y6 Y7 Y8.

2.10.1 Noninterleaved Mode

Block interleaving is the default for multiple components in an image scan. Should only one component be present in an image scan, then regardless of the sampling rate, the MDU is considered to be a single block. This is called noninterleaved mode.

2.11 Resynchronization

Entropy coding makes the compressed file extremely vulnerable to byte errors. To prevent catastrophic loss of the image, resynchronization[5] markers are placed throughout the file.

Resynchronization acts as a mechanism for robustness in the presence of random byte errors in the file, including byte slips and byte insertions. For most fail-safe storage media, resynchronization is not useful and by default, resynchronization is disabled. On the other hand, for potentially noisy channels or unstable storage, resynchronization is recommended and PVRG-JPEG implements resynchronization recovery.

[5]Resynchronization was the term used in the very early JPEG drafts, before Revision 6. The newer term is *restart* intervals, though since the coder was initially designed before this terminology change, it still uses the term resynchronization. We never bothered to rename the variables resynchronization to restart internally in the coder, it doesn't affect coding.

Chapter 3

Calling Parameters

The calling parameters are specified for the UNIX operating system. According to standard convention, when we describe an option it is placed between square brackets. These square brackets nest, so an optional argument of an optional argument would be nestled inside two pairs of brackets.

3.1 Encoder

The calling parameters to start the `jpeg` encoder is

```
jpeg -iw ImageWidth -ih ImageHeight [-JFIF] [-q(l) Q-Factor]
     [-a] [-b] [-d] [-k predictortype] [-n] [-o] [-y] [-z]
     [-p PrecisionValue] [-t pointtransform]
     [-r ResyncInterval] [-s StreamName]
     [[-ci ComponentIndex1] [-fw FrameWidth1] [-fh FrameHeight1]
      [-hf HorizontalFrequency1] [-vf VerticalFrequency1]
      ComponentFile1]
     [[-ci ComponentIndex2] [-fw FrameWidth2] [-fh FrameHeight2]
      [-hf HorizontalFrequency2] [-vf VerticalFrequency2]
      ComponentFile1]
      ....

-JFIF puts a JFIF marker. Don't change component indices.
-a enables Reference DCT.
-b enables Lee DCT.
-d decoder enable.
-[k predictortype] enables lossless mode.
-q specifies quantization factor; -ql specifies can be long.
-n enables non-interleaved mode.
-[t pointtransform] is the number of bits for the PT shift.
-o enables the Command Interpreter.
-p specifies precision.
-y run in robust mode against errors (cannot be used with DNL).
-z uses default Huffman tables.
      ....
```

We examine these parameters, case by case, below.

`ImageWidth` specifies the width of the original image. This should correspond to the width of the widest component and, thus, the width of the “original image”. All components have widths roughly corresponding to an integer decimation ratio from this specification.

`ImageHeight` specifies the height of the tallest component. This corresponds to the height of the "original image".

`-JFIF` specifies that a JFIF header is placed on the encoded stream. This is unnecessary for decoding.

`Q-Factor` option specifies a multiplicative factor for the quantization: each quantization coefficient of the default matrix is scaled by (Q-Factor/50). A Q-Factor of 0 is the same thing as a Q-Factor of 50 because it disables this function. `-q` specifies an 8 bit quantization matrix; `-ql` specifies a 16 bit quantization matrix, useful for 12 bit data.

`-a` enables the double-precision floating point Reference DCT. (Default is Chen DCT.)

`-b` enables the Lee DCT. (Default is Chen DCT.)

`-d` enables decoding. See below.

`-k predictortype` The lossless predictor type, specified as an integer between 1-7.

`-n` This option specifies that the files should not be transmitted in interleaved format.

`-o` signals that the command interpreter will read from the standard input.

`-p` Specifies the precision. Normally should be between 2-16 for lossless; 8 or 12 for DCT. If it is specified as a number greater than 8 then the input is considered to be unsigned shorts (16 bits, msb first). Not aggressively checked.

`-t pointtransform` Specifies the shifting (right) upon loading input and shifting (left) upon writing input. Generally used by the lossless mode only. Can be used by the DCT mode to add or subtract bits.

`-y` for decoding only, signals that *no* resynchronization is enabled, thus ignore any markers found in the data stream.

`-z` enables use of default Huffman tables. This converts the coding from a two-pass system using the first pass to generate custom tables to a one-pass system using internal default tables. With this option, the compression speed is nearly doubled, but because the internal tables are not custom to the image, the compressed file size increases slightly.

`ResyncInterval` specifies a resync interval for the input file–if set to 0 (default), resynchronization is disabled; otherwise it signifies the number of MDU between a resync marker.

`StreamName` is the place to load(decoder)/store(encoder) the coded image–if unspecified it defaults to `ComponentFile1.jpg`.

For every component in the image we have:

- `ComponentIndex` describes the component index where the file data should be associated with. The possible values are between 0 and 255. As a rule Y is in 1; Cb (or U) is in 2; Cr (or V) is in 3. The file specfications, if left undisturbed, will result in component location of 1 for the first component file, 2 for the second component file, and so on. If `-ci` is specified for the previous component file, then the next component index defaults to the previous component index plus 1.
- `FrameWidth` describes the actual width of the component. This should be determinable by the size of the original image (ImageHeight and ImageWidth) and the frequency sampling of that component. However, because the program assumes that the sampling component will round *up* to the nearest integer and other programs may not necessarily follow that convention, we allow precise specification of the FrameWidth. A specification that violates logical MDU blocks will flag warnings. If necessary, for the encoder, additional image space is padded out by pel replication; for the decoder, additional image space is padded out by zeroes (or the default operating system convention for filling disk "holes").

- `FrameHeight` describes the actual height of the component. Multiplied together with FrameWidth, this should equal the file size of the component.
- `Hor-Frequency` specifies the block sampling frequency of the component in the horizontal direction for every MDU transmitted.
- `Ver-Frequency` specifies the block sampling frequency of the component in the vertical direction. When multiplied together with the Horizontal frequency, it corresponds to the number of blocks of that component in the MDU.
- `ComponentFile` represents the directory path location of the th component file.

3.1.1 Encoding Examples

A command to encode a grayscale (single component) 320x200 file `foobar` with a resynchronization interval of 20 is

```
jpeg -iw 320 -ih 200 foobar -r 20
```

which would put a 1:1 interleaved single component JPEG compressed stream into `foobar.jpg`.

A command to encode a sequence of ten files, `dozen1` through `dozen10` with the first file having frequency of 12,12 (full size) of dimension 1200x1200, the second one having frequency of 7,7 with size 702x702 (some extra bits) and the rest having a frequency of 1,1 and dimension 100x100 to the file `HalfDozen` would use

```
jpeg -iw 1200 -ih 1200 -hf 12 -vf 12 dozen1\
-hf 7 -vf 7 -fw 702 -fh 702 dozen2\
dozen3 dozen4 dozen5 dozen6 dozen7 dozen8 dozen9 dozen10\
-s HalfDozen
```

A command to encode a sequence of files with dimension 128x128 in `image.Y`; 64x128 in `image.U`; 64x128 in `image.V`; to the output file `image.jpg` is given by

```
jpeg -iw 128 -ih 128 -hf 2 image.Y image.U image.V -s image.jpg
```

To do it losslessly with predictor type 7, just add `-k 7` to obtain

```
jpeg -iw 128 -ih 128 -k 7 -hf 2 image.Y image.U image.V -s image.jpg
```

3.2 Decoder

The decoding parameters are simpler because most of the information is already present in the header of the compressed file. The decoder is enabled with the file specification `-d`.

```
jpeg -d [-u] -s StreamName
      [ComponentFile1] [ComponentFile2] ...
```

An itemization of these parameters:

`-u` prevents the notification line describing the dimensions and the name of the file decoded.

`StreamName` is the JPEG compressed file generated by the encoder.

`ComponentFile` is where the decoder output is placed. Unless one knows exactly how many components are sent over and that they are sent over consecutively, it is rather difficult to anticipate where and in what order to place the component filenames. Thus, if they are not specified, they are placed in files designated by *StreamName.N*, where *N* is the number of the component. Furthermore, should more than one image be contained in the stream, the subsequent images and components will be put in files *StreamName.M.N* where *M* represents the current image index and *N* represents the component of that image.

3.2.1 Decoding Examples

To decode the file `foobar.jpg` into its components, type

```
jpeg -d -s foobar.jpg
```

The output will be placed in files `foobar.0 foobar.1 ...`, the number of which corresponds to as many files transmitted. A message will be sent to the standard output unless the `-u` option was specified, printing

```
> GW:320  GH:200  R:20
>> C:0  N:foobar.jpg.0  W:320  H:200  hf:1  vf:1
...
```

In this message, the following keys are used:

GH: Global height: this is the height of the image.

GW: Global width: this is the width of the image.

C: Component number retrieved in the scan.

N: Name of file where the component was saved to.

H: The height of the component file.

W: The width of the component file.

hf: The horizontal frequency sampling of the component file.

vf: The vertical frequency sampling of the component file.

In the above example, the global (frame) dimensions are 320x200 and the first component was saved in `foobar.jpg.0` with dimensions of 320x200, thus representing a horizontal and vertical sampling frequency of 1.

Since the order of transmission for encoding normally increases each subsequent component index by one, to decode the `HalfDozen` (generated above) into files `hd1` through `hd10` we can use

```
jpeg -d -s HalfDozen hd1 hd2 hd3 hd3 hd4\
hd5 hd6 hd7 hd8 hd9 hd10
```

To describe the filenames is easy in this case because we already know how many files have been encoded. However, for those filenames without an explicitly designated filename, the default filename `HalfDozen.i`, where `i` is the component index, is used.

3.3 Ranges for parameters

The header file in JPEG allows the following ranges for the parameters and so does the program.

Number of ComponentFiles per Scan 16, Frame 256.

ImageWidth: between 0 and 65535.

ImageHeight: between 0 and 65535.

FrameWidth: between 0 and 65535.

FrameHeight: between 0 and 65535.

HorizontalFrequency: between 1 and 15.

VerticalFrequency: between 1 and 15.

Number of Quantization tables: between 0 and 15.

Number of Huffman tables: between 0 and 15.

Resynchronization (restart): between 0 and 65535. Disabled with 0.

JPEG Baseline restricts these values to the following:

Number of ComponentFiles per Scan 10, Frame 256.

ImageWidth: between 0 and 65535.

ImageHeight: between 0 and 65535.

FrameWidth: between 0 and 65535.

FrameHeight: between 0 and 65535.

HorizontalFrequency: between 1 and 4.

VerticalFrequency: between 1 and 4.

Number of Quantization tables: between 0 and 3.

Number of Huffman tables: between 0 and 1.

Resynchronization (restart): between 0 and 65535. Disabled with 0.

3.4 Return Values

The return value on successful completion is a 0. Other values indicate errors as follows:

```
ERROR_NONE 0
ERROR_BOUNDS 1            /*Input Values out of bounds */
ERROR_HUFFMAN_READ 2      /*Huffman Decoder finds bad code */
ERROR_HUFFMAN_ENCODE 3    /*Undefined value in encoder */
ERROR_MARKER 4            /*Error Found in Marker */
ERROR_INIT_FILE 5         /*Cannot initialize files */
ERROR_UNRECOVERABLE 6     /*No recovery mode specified */
ERROR_PREMATURE_EOF 7     /*End of file unexpected */
ERROR_MARKER_STRUCTURE 8  /*Bad Marker Structure */
ERROR_WRITE 9             /*Cannot write output */
ERROR_READ 10             /*Cannot write input */
ERROR_PARAMETER 11        /*System Parameter Error */
ERROR_MEMORY 12           /*Memory exceeded */
```

Chapter 4

The Command Interpreter

The command interpreter is used to make custom JPEG files according to the specifications given in the reference document. This command interpreter can construct coded streams that are a superset of the JPEG baseline specifications. The decoder also accepts coded streams that are a superset of JPEG baseline specifications.

For a compressed file to be decoded, the input must be well formed inasmuch as all references to tables must be resolvable during the decoding process. It is necessary to revisit some of the concepts of the JPEG coder before the command interpreter can be used properly.

4.1 Terminology

We use the following terminology to specify the description of the commands allowed by our interpreter.

Braces are used to delimit expressions. A set of values between two braces indicates a range of occurrences of the prefix. For example, 3 5 indicates 3 to 5 occurrences of .

The * operator is the Kleene operator and designates zero or more occurrences of the prefix. Basically, a*, means zero or more occurrences of a, the set .

The + operator designates one or more occurrences of the prefix. Thus a+ is the set , the same as aa*.

Brackets [] are an overloaded definition. In some instances it is used to delimit an optional argument. Thus [a] means that the occurrence of "a" is optional. This is consistent with Extended Backus-Naur Form (EBNF). In terms of the `Definition` they also indicate a range. For example, [a-z] represents all elements from a to z and [â] means all characters not equivalent to a. This is consistent with the UNIX lexical convention. Finally, these are also valid tokens.

Parentheses () are used for grouping. In this context, an indicates an "or" function. For example indicates that either a or b can be used.

4.2 Well-formed JPEG files

A JPEG file typically consist of the following components,

```
{SOI Marker
 {Quantization Marker}*
 {Huffman Marker}*
 {Resync Interval Marker}*
 [Frame Marker
```

```
  {{Quantization Marker}*
   {Huffman Marker}*
   {Resync Interval Marker}*
   Scan Marker
   {Coded-Data
    Resync Marker}*
   [Define-Number-of-Lines Marker]
  }*]
 EOI Marker}+
```

Note that a Number of Lines marker can be sent any time after a multiple of row MDU, before the next marker code after the first Scan of an image. Baseline requirements have it sent exactly before the next marker code of the first scan. Furthermore there can be only one Define-Number-of-Lines marker per image. We also read it properly when it is placed after any resynchronization codes in the first scan.

4.3 Tokens

A token is the basic unit or word in the vocabulary of the command interpreter. It can be of several sorts.

A token can be an English word or a concatenated English word such as “STREAMNAME.” For English words the command interpreter is case insensitive. That means “StReamNaME,” “STREAMNAME,” and “streamname” are all identical to the interpreter.

A token can be an integer defined by the following nomenclature:

```
Definition: Digit -> [0-9]
            HexDigit -> ({Digit}|[a-fA-F])
            OctalDigit -> [0-7]
            DecInteger -> {Digit}+
            HexInteger -> 0[xX]{HexDigit}+
            OctInteger -> 0[oO]{OctalDigit}+
            HexInteger2 -> {HexDigit}+[Hh]
            OctInteger2 -> {OctalDigit}+[BCObco]
            CharInteger -> '([^\\]|\\([\n^\n]|{OctalDigit}{1,3}))'
Syntax:     Decimal-Integer = digits
            Hex-Integer = 0xhexdigits 0Xhexdigits
                    (hexdigits)h (hexdigits)H
            Octal-Integer = 0ooctdigits 0Ooctdigits
                    (octdigits)b (octdigits)B
                    (octdigits)c (octdigits)C
                    (octdigits)o (octdigits)O
            Character-Integer = 'character'

Example:    10 0x10 10H 10B '\n' 'G'
```

A token can be a string, which is defined according to the following nomenclature:

```
Definition: String -> \"([^\"]|\\\")*\"
Syntax:     "non-escaped-double-quote-character"
Example:    "Hello George"
            "Lord of the Flies"
            "Quotes \"\" are fun\n"
```

A token can be a comment (not a regular expression because it nests) and can be written by C's matching comment pairs "/* Comment-text */".

A token can be braces [], denoting some sort of array or repeated structure.

4.3.1 Special Characters

A special character in the definition of a character-integer or a string can be written with the escape character.

The escape character singles out the next character to be printed out verbatim, except for special cases. The following *one characters* denote special control characters.

b: backspace.

i: (horizontal) tab.

n: newline.

v: (vertical) tab.

f: form feed.

r: carriage return.

0: null.

If the string following the escape character is greater than one single digit (and less than or equal to three digits), it must be an octal code. An octal code is written as a sequence of two or more octal digits. Care must be taken when imbedding octal codes inside of other numerical digits. Thus `466` is undefined (exceeds 255) and `" 0466"` is the equivalent of `"&6"`.

4.4 Comments

First of all, we start out by defining what a comment is. Comments begin with a `/*` and ends with a `*/`. They nest, hence, the following examples are valid comments.

```
Definition: S -> S S |
                 /* (string!=*/) S (string!=*/) */ |
                 /* (string!=*/) */
Syntax:     None.
Example:    E/* Comment /* These comments,
            unlike those of C, nest... *//**/ */
            /* Simple Comment */
```

4.5 Printing Status

The current status of the "image"; that is, all the tables associated with the current state of the device, is obtained by

```
Definition: S -> PRINTIMAGE
Syntax:     PRINTIMAGE
Example:    PRINTIMAGE
```

The status of all of the components of the "image" can be obtained by the following command

```
Definition: S -> PRINTFRAME
Syntax:     PRINTFRAME
Example:    PRINTFRAME
```

The status of the current scan can be obtained by the following command

```
Definition: S -> PRINTSCAN
Syntax:     PRINTSCAN
Example:    PRINTSCAN
```

The `ECHO` command prints out a string to the standard output.

```
Definition: S -> ECHO string
Syntax:     ECHO "printable-characters"
Example:    ECHO "Play it, Sam, play it again."
```

4.6 Structure Definition

The structure must be specified before the streams can be opened. For example, we must know the filenames before we can open up the streams. The specification of the stream name is given by

```
Definition: S -> STREAMNAME string
Syntax:     STREAMNAME Filename

Example:    STREAMNAME "Output.Compressed"
```

Each "image" component, or color, resides on a separate file with a given frame-location, filename, horizontal frequency, vertical frequency. All components must be specified before a `WRITEFRAME` command is given. This is specified as

```
Definition: S -> COMPONENT
             ( integer [string integer integer integer] |
              [ {integer [string integer integer integer]}+ ] )
Syntax:     COMPONENT Index
                    [Filename Hor_freq Ver_freq Quant_table]
            COMPONENT [
            Index1 [Filename1 Hor_freq1 Ver_freq1 Quant_table1]
                      ...
            Indexn [Filenamen Hor_freqn Ver_freqn Quant_tablen]
                      ]
Example:    COMPONENT 0 ["Boats" 1 1 0]
            COMPONENT [ 10 ["Bo Derek" 3 2 1]
                       21 ["Bigblue" 1 1 1]
                      ... ]
```

A scan is a pass using some of the components present in the entire image. There can be up to 16 components in a single scan. For images with lots of components, we must use more than one scan to send all of the components over. The current specification of a scan overrides any previous specification; hence if there are multiple components to be interleaved in the scan, they must be specified all at once.

```
Definition: S -> SCAN ( integer [ integer integer ] |
                  [ {integer [ integer integer ]}+ ] )
Syntax:     SCAN Frame_index  [ Ac_Huffman Dc_Huffman ]
            SCAN [ {Frame_index
                       [ Ac_Huffman Dc_Huffman ] }* ]
Example:    SCAN 5 [0 1]
            SCAN [10 [0 1] 21 [0 0] ... ]
```

4.7 Table Specifications

A quantization table is a 64 element array consisting of the quantizer step for the corresponding DCT coefficient. The model for the quantizer allows for 64 input elements, and the unspecified elements in the array are set to 16. These 64 elements are input in scan left to right, top to bottom, representing increasing horizontal DCT frequency and increasing vertical DCT frequency, respectively.

There are two special keywords which can be substituted instead of the integer array brackets. They are LUMINANCEDEFAULT and CHROMINANCEDEFAULT and represent the internal luminance and chrominance quantization matrices.

```
Definition: S -> QUANTIZATION
                 ( {integer ([ { integer }{0,64} ] |
                   LUMINANCEDEFAULT | CHROMINANCEDEFAULT)} |
                  [ {integer ([ { integer }{0,64} ] |
                   LUMINANCEDEFAULT | CHROMINANCEDEFAULT)}+ ] )
Syntax:     QUANTIZATION Quant_index [ Qvalue1 ... ]
            QUANTIZATION [
                    Quant_index1 [ Qvalue11 ... ]
                    Quant_index2 [ Qvalue21 ... ]
                      ... ]
Example:    QUANTIZATION 0 [16 18 18 14 14 14 14 ... ]
    QUANTIZATION [ 0 LUMINANCEDEFAULT
                        1 CHROMINANCEDEFAULT ... ]
```

The custom Huffman tables can only be specified after the FREQUENCY command, which accumulates statistics using the installed parameter set-up. After the FREQUENCY command, each component will have a corresponding frequency table. We build a Huffman table for a given multiple of components by combining their frequency tables; for example, ACCUSTOM 0 [0 1] would accumulate the frequency statistics of the first and second components of the scan and put it in Huffman table location 0.

```
Definition: S -> (ACCUSTOM | DCCUSTOM)
                   (integer [ {integer}* ] |
                    [ {integer [ {integer}* ] }+ ])
Syntax:     ACCUSTOM Huffman-index-dest [freqindex ... ]
            DCCUSTOM Huffman-index-dest [freqindex ... ]
            ACCUSTOM [
                 Huffman-index-dest1 [freqindex11 ... ]
                 Huffman-index-dest2 [freqindex21 ... ]
                   ... ]
            DCCUSTOM [
                 Huffman-index-dest1 [freqindex11 ... ]
                 Huffman-index-dest2 [freqindex21 ... ]
                   ... ]
Example:    ACCUSTOM 0 [0]
            ACCUSTOM 1 [1 2]
            DCCUSTOM [0 [0] 1 [1 2]]
```

The ACSPEC and DCSPEC allows direct control over the Huffman tables. Two special cases, LUMINANCEDEFAULT and CHROMINANCEDEFAULT, may be used to access the internal default Huffman tables for both the AC and DC components. If a predefined table is to be loaded, the table must be specified by an array of Huffman bit-lengths for all codes, followed by an array of code values in order of increasing size. If any of the arrays are incomplete–the first one should have sixteen elements and the second one should have up to 256–the unspecified elements are considered zeroes.

```
Definition: S -> (ACSPEC | DCSPEC) (integer
           ([ {integer}{0,32} ]
            [ {integer}{0,257} ] |
            LUMINANCEDEFAULT|
            CHROMINANCEDEFAULT)) |
           [ {integer ([ {integer}{0,32} ]
                       [ {integer}{0,257} ] |
            LUMINANCEDEFAULT|
            CHROMINANCEDEFAULT))}+ ] )
Syntax:     ACSPEC Huffman-index
             [NumberBitlength1... ] [SmallestCode...]
             [ACDEFAULT]
            DCSPEC Huffman-index
     [NumberBitlength1... ] [SmallestCode...]
             [DCDEFAULT]
            ACSPEC [
                    Huffman-index1
             [NumberBitlength11... ] [SmallestCode1...]
                    Huffman-index2
             [NumberBitlength11... ] [SmallestCode2...]
                    Huffman-indexi
             [ACDEFAULT]
                    ...]
            DCSPEC [
                    Huffman-index1
             [NumberBitlength11... ] [SmallestCode1...]
                    Huffman-index2
             [NumberBitlength11... ] [SmallestCode2...]
                    Huffman-indexi
             [DCDEFAULT]
                    ...]
Example:    DCSPEC [0 LUMINANCEDEFAULT 1 CHROMINANCEDEFAULT]
            ACSPEC 0 LUMINANCEDEFAULT
```

```
DCSPEC [0 [0 1 5 1 1 1 1 1 1]
          [0 1 2 3 4 5 6 7 8 9 0xa 0xb]
        1 [0 3 1 1 1 1 1 1 1 1 1]
          [0 1 2 3 4 5 6 7 8 9 0xa 0xb]]
ACSPEC 0 [0x00 0x02 0x01 0x03 0x03 0x02
          0x04 0x03 0x05 0x05 0x04 0x04
          0x00 0x00 0x01 0x7d]
         [0x01 0x02 0x03 0x00 0x04 0x11
          0x05 0x12 0x21 0x31 0x41 0x06
          0x13 0x51 0x61 0x07
          0x22 0x71 0x14 0x32 0x81 0x91
          0xa1 0x08 0x23 0x42 0xb1 0xc1
          0x15 0x52 0xd1 0xf0
          0x24 0x33 0x62 0x72 0x82 0x09
          0x0a 0x16 0x17 0x18 0x19 0x1a
          0x25 0x26 0x27 0x28
          0x29 0x2a 0x34 0x35 0x36 0x37
          0x38 0x39 0x3a 0x43 0x44 0x45
          0x46 0x47 0x48 0x49
          0x4a 0x53 0x54 0x55 0x56 0x57
          0x58 0x59 0x5a 0x63 0x64 0x65
          0x66 0x67 0x68 0x69
          0x6a 0x73 0x74 0x75 0x76 0x77
          0x78 0x79 0x7a 0x83 0x84 0x85
          0x86 0x87 0x88 0x89
          0x8a 0x92 0x93 0x94 0x95 0x96
          0x97 0x98 0x99 0x9a 0xa2 0xa3
          0xa4 0xa5 0xa6 0xa7
          0xa8 0xa9 0xaa 0xb2 0xb3 0xb4
          0xb5 0xb6 0xb7 0xb8 0xb9 0xba
          0xc2 0xc3 0xc4 0xc5
          0xc6 0xc7 0xc8 0xc9 0xca 0xd2
          0xd3 0xd4 0xd5 0xd6 0xd7 0xd8
          0xd9 0xda 0xe1 0xe2
          0xe3 0xe4 0xe5 0xe6 0xe7 0xe8
          0xe9 0xea 0xf1 0xf2 0xf3 0xf4
          0xf5 0xf6 0xf7 0xf8
          0xf9 0xfa]
ACSPEC 1 [0x00 0x02 0x01 0x02 0x04 0x04
          0x03 0x04 0x07 0x05 0x04 0x04
          0x00 0x01 0x02 0x77]
         [0x00 0x01 0x02 0x03 0x11 0x04
          0x05 0x21 0x31 0x06 0x12 0x41
          0x51 0x07 0x61 0x71
          0x13 0x22 0x32 0x81 0x08 0x14
          0x42 0x91 0xa1 0xb1 0xc1 0x09
          0x23 0x33 0x52 0xf0
          0x15 0x62 0x72 0xd1 0x0a 0x16
          0x24 0x34 0xe1 0x25 0xf1 0x17
          0x18 0x19 0x1a 0x26
          0x27 0x28 0x29 0x2a 0x35 0x36
          0x37 0x38 0x39 0x3a 0x43 0x44
          0x45 0x46 0x47 0x48
```

```
                                        0x49 0x4a 0x53 0x54 0x55 0x56
                                        0x57 0x58 0x59 0x5a 0x63 0x64
                                        0x65 0x66 0x67 0x68
                                        0x69 0x6a 0x73 0x74 0x75 0x76
                                        0x77 0x78 0x79 0x7a 0x82 0x83
                                        0x84 0x85 0x86 0x87
                                        0x88 0x89 0x8a 0x92 0x93 0x94
                                        0x95 0x96 0x97 0x98 0x99 0x9a
                                        0xa2 0xa3 0xa4 0xa5
                                        0xa6 0xa7 0xa8 0xa9 0xaa 0xb2
                                        0xb3 0xb4 0xb5 0xb6 0xb7 0xb8
                                        0xb9 0xba 0xc2 0xc3
                                        0xc4 0xc5 0xc6 0xc7 0xc8 0xc9
                                        0xca 0xd2 0xd3 0xd4 0xd5 0xd6
                                        0xd7 0xd8 0xd9 0xda
                                        0xe2 0xe3 0xe4 0xe5 0xe6 0xe7
                                        0xe8 0xe9 0xea 0xf2 0xf3 0xf4
                                        0xf5 0xf6 0xf7 0xf8
                                        0xf9 0xfa]
```

For example, if the table consists of two Huffman codes corresponding to symbol 0 and 240 respectively, the first one of length 2 and the second one of length 4, the specification of the length array would be `[0 1 0 1]` and the codes array would be `[0 240]`. If we wanted to load this as the AC code table in index 5, we would use

```
ACSPEC 5 [0 1 0 1] [0 240]
```

In the actual specification of the AC codes, two special purpose codes are used in addition to the normal AC run-length codes. These are `0` and `240`, representing the `ZRL` and `EOB` codes respectively. These should always be specified.

4.8 Opening and Closing files

We start by opening up the stream write file. The StreamName must be defined before any of the write marker commands are used. The following command opens the stream output for writing.

```
Definition: S -> OPENSTREAM
Syntax:     OPENSTREAM
Example:    OPENSTREAM
```

To open up a particular set of read files specified by the Scan we use the following command. Before any reading can be performed, we must have used this function.

```
Definition: S -> OPENSCAN
Syntax:     OPENSCAN
Example:    OPENSCAN
```

To close the write stream we use

```
Definition: S -> CLOSESTREAM
```

```
Syntax:      CLOSESTREAM
Example:     CLOSESTREAM
```

After we are done with a scan, we must free up the input read pipes so that they may be reused for other possible images. Most Unix machines only permit 32 of these read pipes to be open at one time, thus closing them off is important.

```
Definition: S -> CLOSESCAN
Syntax:      CLOSESCAN
Example:     CLOSESCAN
```

4.9 Queuing up tables to send

We queue up a quantization matrix location to be sent by the next `WRITEQUANTIZATION` command by the `QSEND` command.

```
Definition: S -> QSEND integer
Syntax:      QSEND Quantization_index
Example:     QSEND 0
```

To queue up a DC or AC Huffman table location to be sent with the next `WRITEHUFFMAN` command, we use `ACSEND` and `DCSEND`.

```
Definition: S -> (ACSEND | DCSEND) integer
Syntax:      ACSEND Ac_Huffman_index
             DCSEND Dc_Huffman_index
Example:     ACSEND 0 ACSEND 1 DCSEND 0
```

4.10 Writing Marker Codes

The following sections detail exactly how to manipulate the marker codes to write information out to the stream.

The SOI or Start of Image marker must be written before every JPEG baseline file. It is written by

```
Definition: S -> WRITESOI
Syntax:      WRITESOI
Example:     WRITESOI
```

The EOI or End of Image marker must be written after encoding has been completed but before closing off the image. It is generally the last code in the file. It is written by

```
Definition: S -> WRITEEOI
Syntax:      WRITEEOI
Example:     WRITEEOI
```

The frame marker is used to write out the component specification. It must occur before any scans are written.

```
Definition: S -> WRITEFRAME
Syntax:     WRITEFRAME
Example:    WRITEFRAME
```

The `WRITESCAN` command is used to write out the scan specification marker and the subsequent compressed data.

```
Definition: S -> WRITESCAN
Syntax:     WRITESCAN
Example:    WRITESCAN
```

Once the quantization tables have been defined and queued up to send, we write them off to storage by the following command.

```
Definition: S -> WRITEQUANTIZATION
Syntax:     WRITEQUANTIZATION
Example:    WRITEQUANTIZATION
```

Similarly, once the AC and DC Huffman tables have been defined and queued up to send, we write them off to storage by the following command.

```
Definition: S -> WRITEHUFFMAN
Syntax:     WRITEHUFFMAN
Example:    WRITEHUFFMAN
```

Notification of a resync interval to the decoder is done by writing a resync marker (note that the `RESYNC` command should have been used beforehand to define an appropriate interval). This is done by

```
Definition: S -> WRITERESYNC
Syntax:     WRITERESYNC
Example:    WRITERESYNC
```

In order to write special marker codes, such as marker codes `APPn` we give the following command to directly write byte-aligned marker codes with the appropriate length parameter out to the stream. This is done by

```
Definition: S -> WRITESPECIAL ( marker [ {integer}* ] |
                { marker [ {integer}* ]}* )
Syntax:     WRITESPECIAL Marker_value [integer1 ...]
            WRITESPECIAL [ Marker_value1 [integer11 ...]
                     Marker_value2 [integer21 ...]
                     ... ]
Example:    WRITESPECIAL 0xe0 ['j' 'p' 'e' 'g' 0x00]
```

The text might also include “JFIF” should a special marker like that be required.

In order to directly write byte-aligned bytes (doesn’t consider the first marker byte or the length, but WYSIWYG) to the stream we have the command

```
Definition: S -> WRITEDIRECT [{integer}*]
Syntax:     WRITEDIRECT [integer1 integer2 ...]
Example:    WRITEDIRECT [ 0xff 0xe0 0x00 0x07
                    'j' 'p' 'e' 'g' 0x00]
```

4.11 Definition of Number of Lines

The Number of Lines command specifies the number of lines (height) of the image *after* the frame and scan headers have been read. It is used primarily for image scanners where the height of the picture is unknown at the start of encoding. Under baseline specifications, the place of the Number of Lines is after the scan has been encoded (and therefore an integer number of MDU rows have been coded.) We note, of course, that because of error properties, it is usually safer to put the DNL marker around the final resync command in the scan structure. We allow this to occur.

To enable the Number of Lines command, the command `SCANDNL` must have been specified *before* the frame marker is written. (DNL stands for “Define Number of Lines”.) The three forms for the `SCANDNL` command are as follows: (1) a number is specified which is the resync location to place the dnl marker; (2) the keyword `AUTO` is used which automatically places the dnl marker at the end of the first scan; (3) the keyword `ENABLE` is used which allows explicit writing of the Number of Lines marker by `WRITEDNL` command. If the number specified by (1) exceeds the last resync number, it is clipped to equal the last resync number.

```
Definition: S -> SCANDNL (AUTO | ENABLE | integer)
Syntax:     SCANDNL AUTO
            SCANDNL ENABLE
            SCANDNL resync-location
Example:    SCANDNL 10
```

The `WRITEDNL` command is used only with the `SCANDNL ENABLE` command. It explicitly places the Number of Lines marker at the particular stream location when the command was executed. Note that the Number of Lines marker should precede the first scan if this command is used.

```
Definition: S -> WRITEDNL
Syntax:     WRITEDNL
Example:    WRITEDNL
```

4.12 Statistics Commands

Once the Scan parameters have been defined, we can pass through the image files and compile frequency statistics of the run-length codes for compression. This must be done before constructing *custom* Huffman tables. We do this by the following command

```
Definition: S -> FREQUENCY
Syntax:     FREQUENCY
Example:    FREQUENCY
```

4.13 Parameter Definitions

The following commands define some simple parameters governing the coding structure.

The input I/O buffer can be specified before the "openscan" command by using the following command

```
Definition: S -> BUFFER integer
Syntax:     BUFFER Io_buffer_linesize
Example:    BUFFER 1024
```

The resync interval for error-recovery is enabled by a non-zero value. This represents the number of MDU between the placement of "resync markers." Use of resynchronization markers enables the decoder to resynchronize decoding upon byte error.

```
Definition: S -> RESYNC integer
Syntax:     RESYNC Resync_interval
Example:    RESYNC 50
```

The image height and image width denote the size of the original image. Basically, it is dependent of the frequency sampling rates of the components but is, in general, slightly "off" because of round-up and round-down errors when calculating file sizes through sampling frequencies.

```
Definition: S -> (IMAGEHEIGHT | IMAGEWIDTH) integer
Syntax:     IMAGEHEIGHT True_image_height
            IMAGEWIDTH True_image_width
Example:    IMAGEHEIGHT 486 IMAGEWIDTH 720
```

The frame and frame width concern the size of one particular component of the image. It is loosely related to the sampling rates and the size of the true image. This can be autosized from estimation of the decimation ratios from the original IMAGEHEIGHT and IMAGEWIDTH.

```
Definition: S -> (FRAMEHEIGHT | FRAMEWIDTH)
                    integer integer
Syntax:     FRAMEHEIGHT Component_index
                    Component_image_height
            FRAMEWIDTH Component_index
                    Component_image_width
Example:    FRAMEHEIGHT 0 200 FRAMEWIDTH 0 200
```

4.14 Defaults

At start, the resynchronization interval is 0 (disabled), the quantization matrices in slots 0 and 1 are the default quantization matrices but the Huffman tables in slots 0 and 1 are *not* loaded with any default tables.

4.15 Examples

To encode a single component file with a specialized quantization matrix cannot be done from the straight UNIX command-line. The following interpreted program does it.

```
/* This program sends a 128x128 file blkint.jpg.1
to the compressed file output.jpg using a predefined
quantization. */
STREAMNAME "output.jpg"
COMPONENT [0 ["blkint.jpg.1" 1 1 0]]
SCAN [0 [0 0]]
IMAGEWIDTH 128
IMAGEHEIGHT 128
QUANTIZATION 0 [
9 9 9 7 7 7 7 7
9 7 7 7 7 7 7 9
7 7 7 7 7 7 9 9
7 7 7 7 7 9 9 15
5 7 7 7 7 9 15 17
7 7 7 7 11 13 17 23
7 7 7 11 13 19 23 27
7 7 13 13 21 21 27 23
]
QSEND 0
OPENSTREAM
WRITESOI
WRITEFRAME
OPENSCAN
FREQUENCY
ACSPEC 0 LUMINANCEDEFAULT
DCSPEC 0 LUMINANCEDEFAULT
ACSEND 0
DCSEND 0
WRITEHUFFMAN
WRITEQUANTIZATION
WRITESCAN
CLOSESCAN
WRITEEOI
CLOSESTREAM
```

Let's take a more complicated example. Suppose we wanted to send over our own Huffman tables rather than combining them using frequency statistics and the `ACCUSTOM` and `DCCUSTOM` commands. The following program does it.

```
/* This program sends over a set huffman table
with 10 MDU resynchronization for a 128x128 file
blkint.jpg.1 to the compressed file output.jpg. */
STREAMNAME "output.jpg"
COMPONENT [0 ["blkint.jpg.1" 1 1 0]]
SCAN [0 [0 0]]
IMAGEWIDTH 128
IMAGEHEIGHT 128
RESYNC 10
```

```
QSEND 0
SCANDNL AUTO
OPENSTREAM
WRITESOI
WRITERESYNC
WRITEFRAME
OPENSCAN
FREQUENCY
DCSPEC [0 [0 1 5 1 1 1 1 1 1]
          [0 1 2 3 4 5 6 7 8 9 0xa 0xb]
        1 [0 3 1 1 1 1 1 1 1 1 1]
          [0 1 2 3 4 5 6 7 8 9 0xa 0xb]]
ACSPEC 0 [0x00 0x02 0x01 0x03 0x03 0x02
         0x04 0x03 0x05 0x05 0x04 0x04
         0x00 0x00 0x01 0x7d]
                     [0x01 0x02 0x03 0x00 0x04 0x11
                      0x05 0x12 0x21 0x31 0x41 0x06
                      0x13 0x51 0x61 0x07
                      0x22 0x71 0x14 0x32 0x81 0x91
                      0xa1 0x08 0x23 0x42 0xb1 0xc1
                      0x15 0x52 0xd1 0xf0
                      0x24 0x33 0x62 0x72 0x82 0x09
                      0x0a 0x16 0x17 0x18 0x19 0x1a
                      0x25 0x26 0x27 0x28
                      0x29 0x2a 0x34 0x35 0x36 0x37
                      0x38 0x39 0x3a 0x43 0x44 0x45
                      0x46 0x47 0x48 0x49
                      0x4a 0x53 0x54 0x55 0x56 0x57
                      0x58 0x59 0x5a 0x63 0x64 0x65
                      0x66 0x67 0x68 0x69
                      0x6a 0x73 0x74 0x75 0x76 0x77
                      0x78 0x79 0x7a 0x83 0x84 0x85
                      0x86 0x87 0x88 0x89
                      0x8a 0x92 0x93 0x94 0x95 0x96
                      0x97 0x98 0x99 0x9a 0xa2 0xa3
                      0xa4 0xa5 0xa6 0xa7
                      0xa8 0xa9 0xaa 0xb2 0xb3 0xb4
                      0xb5 0xb6 0xb7 0xb8 0xb9 0xba
                      0xc2 0xc3 0xc4 0xc5
                      0xc6 0xc7 0xc8 0xc9 0xca 0xd2
                      0xd3 0xd4 0xd5 0xd6 0xd7 0xd8
                      0xd9 0xda 0xe1 0xe2
                      0xe3 0xe4 0xe5 0xe6 0xe7 0xe8
                      0xe9 0xea 0xf1 0xf2 0xf3 0xf4
                      0xf5 0xf6 0xf7 0xf8
                      0xf9 0xfa]
ACSPEC 1 [0x00 0x02 0x01 0x02 0x04 0x04
          0x03 0x04 0x07 0x05 0x04 0x04
          0x00 0x01 0x02 0x77]
                     [0x00 0x01 0x02 0x03 0x11 0x04
                      0x05 0x21 0x31 0x06 0x12 0x41
                      0x51 0x07 0x61 0x71
                      0x13 0x22 0x32 0x81 0x08 0x14
                      0x42 0x91 0xa1 0xb1 0xc1 0x09
                      0x23 0x33 0x52 0xf0
                      0x15 0x62 0x72 0xd1 0x0a 0x16
                      0x24 0x34 0xe1 0x25 0xf1 0x17
                      0x18 0x19 0x1a 0x26
                      0x27 0x28 0x29 0x2a 0x35 0x36
                      0x37 0x38 0x39 0x3a 0x43 0x44
                      0x45 0x46 0x47 0x48
                      0x49 0x4a 0x53 0x54 0x55 0x56
                      0x57 0x58 0x59 0x5a 0x63 0x64
                      0x65 0x66 0x67 0x68
                      0x69 0x6a 0x73 0x74 0x75 0x76
                      0x77 0x78 0x79 0x7a 0x82 0x83
                      0x84 0x85 0x86 0x87
                      0x88 0x89 0x8a 0x92 0x93 0x94
                      0x95 0x96 0x97 0x98 0x99 0x9a
                      0xa2 0xa3 0xa4 0xa5
                      0xa6 0xa7 0xa8 0xa9 0xaa 0xb2
                      0xb3 0xb4 0xb5 0xb6 0xb7 0xb8
                      0xb9 0xba 0xc2 0xc3
                      0xc4 0xc5 0xc6 0xc7 0xc8 0xc9
                      0xca 0xd2 0xd3 0xd4 0xd5 0xd6
                      0xd7 0xd8 0xd9 0xda
                      0xe2 0xe3 0xe4 0xe5 0xe6 0xe7
                      0xe8 0xe9 0xea 0xf2 0xf3 0xf4
                      0xf5 0xf6 0xf7 0xf8
                      0xf9 0xfa]
ACSEND 0
DCSEND 0
WRITEHUFFMAN
WRITEQUANTIZATION
WRITESCAN
CLOSESCAN
WRITEEOI
CLOSESTREAM
```

Another example showing the use of multiple components is shown below. This uses just one quantization matrix (the luminance default) for the entire encoding, and uses default Huffman tables.

```
/* This program sends a 128x128 file blkint.jpg.1,
64x128 file blkint.jpg.2, 64x128 file blkint.jpg.3,
to the compressed file output.jpg. The define-number
of lines marker is also enabled. */
STREAMNAME "output.jpg"
COMPONENT [0 ["blkint.jpg.1" 2 1 0]]
COMPONENT [1 ["blkint.jpg.2" 1 1 0]]
COMPONENT [2 ["blkint.jpg.3" 1 1 0]]
SCAN [0 [0 0] 1 [0 0]2 [0 0]]
IMAGEWIDTH 128
IMAGEHEIGHT 128
SCANDNL AUTO
QUANTIZATION 0 LUMINANCEDEFAULT
QSEND 0
OPENSTREAM
```

```
WRITESOI
WRITEFRAME
OPENSCAN
ACSPEC 0 LUMINANCEDEFAULT
DCSPEC 0 LUMINANCEDEFAULT
ACSEND 0
DCSEND 0
WRITEHUFFMAN
WRITEQUANTIZATION
WRITESCAN
CLOSESCAN
WRITEEOI
CLOSESTREAM
```

4.16 Using the command interpreter

In order to access the encoding command interpreter, the `-o` option is used, with the program list of commands directed into `stdin`. As an example, from UNIX, use the following command for the program stored in `test.3stream`:

```
jpeg -o < test.3stream
```

Chapter 5

Program Documentation

5.1 Program Flow

The program consists of eight basic parts and ten overall files. The rough hierarchy of the files are shown in figure 5.1.

A brief description of the files and their contents are as follows:

`jpeg.c` This file contains most of the major blocks of the jpeg routines. The entry point `main()` resides in this file.

`lexer.c` This file contains most of the parsing code to drive the Command Interpreter. The important routine here is called `parse()` and executes the commands read in from the standard input.

`transform.c leedct.c chendct.c` These files contain most of the DCT transform: the reference DCT, the Lee DCT, and the Chen DCT. The quantization routines and the zig-zag routines are in the file `transform.c`.

`codec.c` This file contains the routines to encode the basic DCT block structure. They are direct emulations from those found in the JPEG reference.

`marker.c` This file contains the baseline marker code read and write routines.

`io.c` This file contains the Input/Output file management for the component files. The basic reads and writes at this point are on the "block" level.

`huffman.c` The Huffman encoding and decoding routines are placed here, along with the basic routines to construct suitable Huffman tables from the bitsize and value order.

`stream.c` The bit-level stream interface is in this file. In addition, much of the basic marker code recognition that cannot be abstracted to a higher level (e.g. `marker.c`) are also placed here. Generally, this recognition is on the marker-header level.

5.1.1 Data Flow

Encoder Dataflow

The encoder dataflow starts from the storage medium and flows into buffers for every component of the image contained in `io.c`. By the routines in `transform.c and chendct.c and leedct.c`, a block is read selectively from each buffer and is then transformed, quantized, bounded, zigzagged, creating an output array. This output array is then run-length encoded by `codec.c` and Huffman-encoded by `huffman.c` and placed out to the device via the stream interface in `stream.c`.

Again, the control of the dataflow is primarily by the `jpeg.c` and `lexer.c` file routines. The dataflow is shown in figure 5.2.

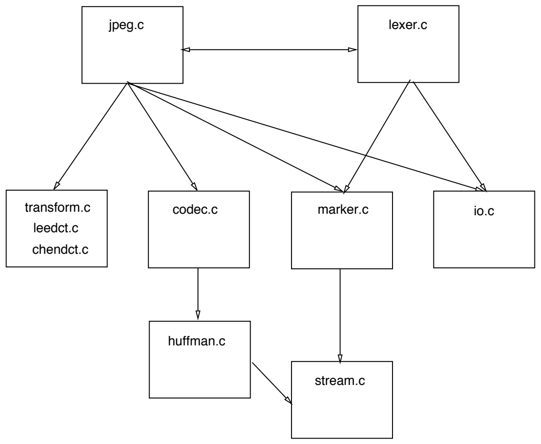


Figure 5.1: The file hierarchy of the jpeg package.

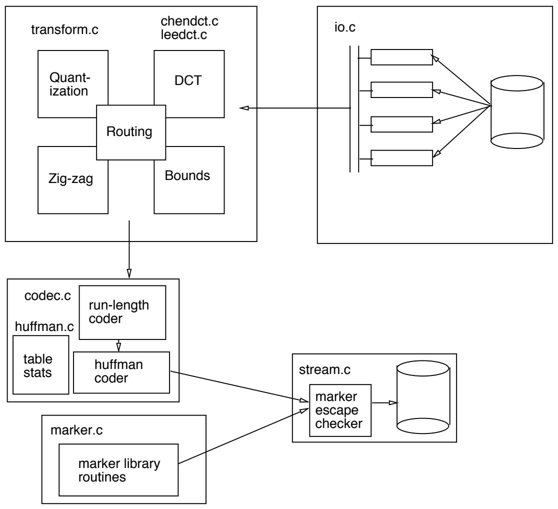


Figure 5.2: The encoder dataflow diagram.

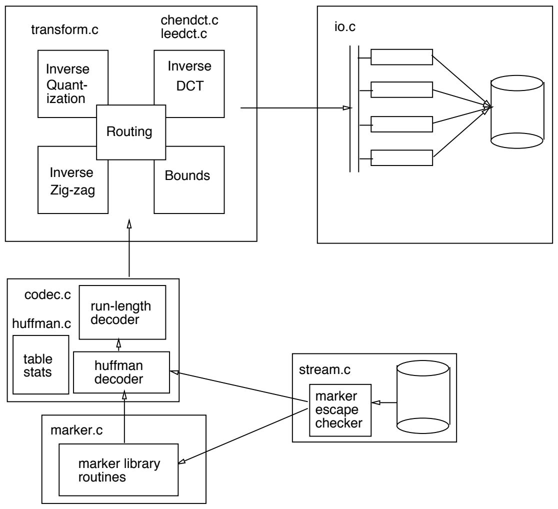


Figure 5.3: The decoder dataflow diagram.

Decoder Dataflow

The decoder dataflow is nearly identical to the encoder dataflow, except the direction of the paths are reversed. The compressed file is read via the stream interface and the result, if a marker, is decoded by `marker.c`, otherwise it is decoded by `huffman.c` which does the Huffman decoding and `codec.c` which does the inverse run-length decoding. Finally, the result is inverse zig-zagged, inverse quantized, inverse transformed, and bounded, with the output being sent to one of the buffers for each component maintained in `io.c`.

The control of the dataflow is primarily by the `jpeg.c` file routines. The dataflow is shown in figure 5.3.

5.1.2 Control Threads

On a related issue to the data flow is the process control threads. The start of the thread (no forks are made so there's only one thread) for the encoder and decoder is the routine `main()` in `jpeg.c`. The process threads differ once past this main routine.

Encoder Control Threads

The encode control thread is limited to routines in `main.c` and `lexer.c`. At the start, a decision is made whether the `Command Interpreter` is activated and the `parse()` function called. If not, the general `JpegEncodeFrame()` function is called to compress the given frame. This illustration is shown in figure 5.4.

Decoder Control Threads

The decode control thread is in `JpegDecodeFrame()` in `jpeg.c`.. The control first looks for all the markers it can find. Once it is finished, it assumes that the rest of the information must be the data from the previously read scan marker. Obviously, if no scan marker has been read, then errors occur with reading the data but we assume one of the previous markers was a scan marker.

The decoder visits several basic transform and coding routines much like the encoder does. The complete top-level control flow diagram is shown in figure 5.5.

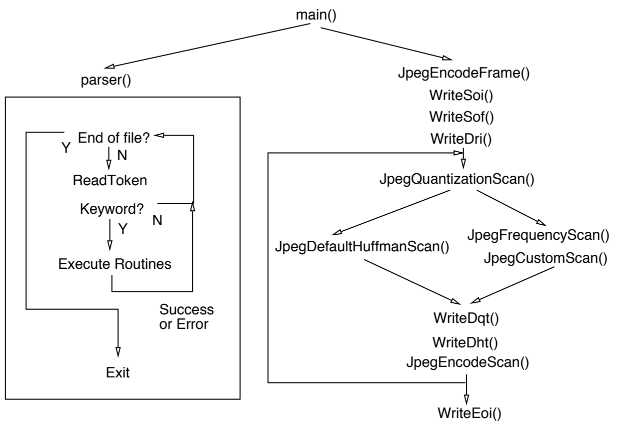


Figure 5.4: The encoder top-level control flow diagram.

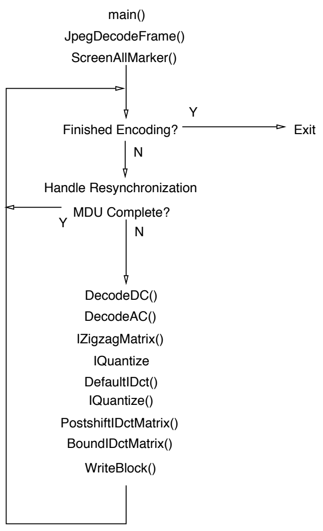


Figure 5.5: The decoder top-level control flow diagram.

5.2 Functional Description

In order to examine the program on a routine-by-routine basis, we have provided the following documentation, itemizing the routines in each file and describing, briefly, their purpose.

5.2.1 jpeg.c

```
int main(int, char **);
static void JpegEncodeFrame(void);
static void JpegDecodeFrame(void);
static void JpegDecodeScan(void);
static void JpegLosslessDecodeScan(void);
static void Help(void);

extern void PrintImage(void);
extern void PrintFrame(void);
extern void PrintScan(void);
extern void MakeImage(void);
extern void MakeFrame(void);
extern void MakeScanFrequency(void);
extern void MakeScan(void);
extern void MakeConsistentFileNames(void);
extern void CheckValidity(void);
extern int CheckBaseline(void);
extern void ConfirmFileSize(void);
extern void JpegQuantizationFrame(void);
extern void JpegDefaultHuffmanScan(void);
extern void JpegFrequencyScan(void);
extern void JpegCustomScan(int);
extern void JpegEncodeScan(void);

extern void JpegLosslessFrequencyScan(void);
extern void JpegLosslessEncodeScan(void);
```

main() (jpeg.c:102) is first called by the shell routine upon execution of the program.

JpegEncodeFrame() (jpeg.c:309) handles the basic encoding of the routines provided that CFrame and CImage are set up properly. It creates the appropriate CScan to handle the intermediate variables.

JpegQuantizationFrame() (jpeg.c:405) sets up the default quantization matrices to be used in the scan. Not to be used with user-specified quantization.

JpegDefaultHuffmanScan() (jpeg.c:441) creates the default tables for baseline use.

JpegFrequencyScan() (jpeg.c:497) assembles the frequency statistics for the given scan, making one AC Freq, DC Freq statistic per component specified. This function should be used before making custom quantization tables.

JpegCustomScan() (jpeg.c:566) assembles custom Huffman tables for the input. It defaults to baseline unless FULLHUFFMAN flag is set.

JpegEncodeScan() (jpeg.c:691) encodes the scan that is given to it. We assume that the quantization and the Huffman tables have already been specified.

JpegLosslessFrequencyScan() (jpeg.c:798) accumulates the frequencies into the DC frequency index.

JpegEncodeLosslessScan() (jpeg.c:1014) encodes the scan that is given to it by lossless techniques. The Huffman table should already be specified.

JpegDecodeFrame() (jpeg.c:1253) is used to decode a file. In general, CFrame should hold just enough information to set up the file structure; that is, which file is to be opened for what component.

JpegLosslessDecodeScan() (jpeg.c:1324) is used to losslessly decode a portion of the image called the scan. This routine uses the internal lossless buffers to reduce the overhead in writing. However, one must note that the overhead is mostly in the Huffman decoding.

JpegDecodeScan() (jpeg.c:1575) is used to decode a portion of the image called the scan. Everything is read upon getting to this stage.

PrintImage() (jpeg.c:1688) prints out the Image structure of the CURRENT image. It is primarily used for debugging. The image structure consists of the data that is held to be fixed even though multiple scans (or multiple frames, even though it is not advertised as such by JPEG) are received.

PrintFrame() (jpeg.c:1740) is used to print the information specific to loading in the frame. This corresponds roughly to the information received by the SOF marker code.

PrintScan() (jpeg.c:1778) is used to print the information in the CScan structure. This roughly corresponds to the information received by the Scan marker code.

MakeImage() (jpeg.c:1812) makes an image and puts it into the Current Image pointer (CImage). It initializes the structure appropriate to the JPEG initial specifications.

MakeFrame() (jpeg.c:1841) constructs a Frame Structure and puts it in the Current Frame pointer (CFrame).

MakeScanFrequency() (jpeg.c:1880) constructs a set of scan information for the current variables. These frequency markers are used for creating the JPEG custom matrices.

MakeScan() (jpeg.c:1916) is used for creating the Scan structure which holds most of the information in the Scan marker code.

MakeConsistentFileNames() (jpeg.c:1952) is used to construct consistent filenames for opening and closing of data storage. It is used primarily by the decoder when all the files may not necessarily be specified.

CheckValidity() (jpeg.c:1984) checks whether the current values in CFrame and CScan meet the internal specifications for correctness and the algorithm can guarantee completion.

CheckBaseline() (jpeg.c:2051) checks whether the internal values meet JPEG Baseline specifications.

ConfirmFileSize() (jpeg.c:2090) checks to see if the files used in the scan actually exist and correspond in size to the input given.

Help() (jpeg.c:2153) prints out general information regarding the use of this JPEG software.

5.2.2 codec.c

```
extern void FrequencyAC(int *);
extern void EncodeAC(int *);
extern void DecodeAC(int *);
extern int DecodeDC(void);
extern void FrequencyDC(int);
extern void EncodeDC(int);
extern void ResetCodec(void);
extern void ClearFrameFrequency(void);
extern void AddFrequency(int *, int *);
extern void InstallFrequency(int);
extern void InstallPrediction(int);
extern void PrintACEhuff(int);
extern void PrintDCEhuff(int);
extern int SizeACEhuff(int);
extern int SizeDCEhuff(int);

extern int LosslessDecodeDC(void);
extern void LosslessFrequencyDC(int);
extern void LosslessEncodeDC(int);
```

FrequencyAC() (codec.c:88) is used to accumulate the Huffman codes for the input matrix. The Huffman codes are not actually stored but rather the count of each code is stored so that construction of a Custom Table is possible.

EncodeAC() (codec.c:146) takes the matrix and encodes it by passing the values of the codes found to the Huffman package.

DecodeAC() (codec.c:211) is used to decode the AC coefficients from the stream in the stream package. The information generated is stored in the matrix passed to it.

DecodeDC() (codec.c:260) is used to decode a DC value from the input stream. It returns the actual number found.

FrequencyDC() (codec.c:295) is used to accumulate statistics on what DC codes occur most frequently.

EncodeDC() (codec.c:325) encodes the input coefficient to the stream using the currently installed DC Huffman table.

ResetCodec() (codec.c:360) is used to reset all the DC prediction values. This function is primarily used for initialization and resynchronization.

ClearFrameFrequency() (codec.c:378) clears all current statistics.

AddFrequency() (codec.c:405) is used to combine the first set of frequencies denoted by the first pointer to the second set of frequencies denoted by the second pointer.

InstallFrequency() (codec.c:427) is used to install a particular frequency set of arrays (denoted by the [index] scan component from the Scan parameters).

InstallPrediction() (codec.c:444) is used to install a particular DC prediction for use in frequency counting, encoding and decoding.

PrintACEhuff() (codec.c:458) prints out the [index] AC Huffman encoding structure in the Image structure.

SizeACEhuff() (codec.c:489) returns the size in bits necessary to code the particular frequency spectrum by the indexed ehuff.

PrintDCEhuff() (codec.c:512) prints out the DC encoding Huffman structure in the CImage structure according to the position specified by [index].

SizeDCEhuff() (codec.c:544) returns the bit size of the frequency and codes held by the indexed dc codebook and frequency.

LosslessFrequencyDC() (codec.c:568) is used to accumulate statistics on what DC codes occur most frequently.

LosslessEncodeDC() (codec.c:593) encodes the input coefficient to the stream using the currently installed DC Huffman table. The only exception is the SSSS value of 16.

LosslessDecodeDC() (codec.c:622) is used to decode a DC value from the input stream. It returns the actual number found.

5.2.3 huffman.c

```
static void CodeSize(void);
static void CountBits(void);
static void AdjustBits(void);
static void SortInput(void);
static void SizeTable(void);
static void CodeTable(void);
static void OrderCodes(void);
static void DecoderTables(void);

extern void MakeHuffman(int *);
extern void SpecifiedHuffman(int *, int *);
extern void MakeDecoderHuffman(void);
extern void ReadHuffman(void);
extern void WriteHuffman(void);
extern int DecodeHuffman(void);
extern void EncodeHuffman(int);
extern void MakeXhuff(void);
extern void MakeEhuff(void);
extern void MakeDhuff(void);
extern void UseACHuffman(int);
extern void UseDCHuffman(int);
extern void SetACHuffman(int);
extern void SetDCHuffman(int);
extern void PrintHuffman(void);
extern void PrintTable(int *);
```

CodeSize() (huffman.c:86) is used to size up which codes are found. This part merely generates a series of code lengths of which any particular usage is determined by the order of frequency of access. Note that the code word associated with 0xffff has been restricted.

CountBits() (huffman.c:150) tabulates a histogram of the number of codes with a give bit-length.

AdjustBits() (huffman.c:172) is used to trim the Huffman code tree into 16 bit code words only.

SortInput() (huffman.c:209) assembles the codes in increasing order with code length. Since we know the bit-lengths in increasing order, they will correspond to the codes with decreasing frequency. This sort is O(mn), not the greatest.

SizeTable() (huffman.c:235) is used to associate a size with the code in increasing length. For example, it would be 44556677... in huffsize[]. Lastp is the number of codes used.

CodeTable() (huffman.c:260) is used to generate the codes once the hufsizes are known.

OrderCodes() (huffman.c:294) reorders from the monotonically increasing Huffman-code words into an array which is indexed on the actual value represented by the codes. This converts the Xhuff structure into an Ehuff structure.

DecoderTables() (huffman.c:316) takes the Xhuff and converts it to a form suitable for the JPEG suggested decoder. This is not the fastest method but it is the reference method.

MakeHuffman() (huffman.c:348) is used to create the Huffman table from the frequency passed into it.

SpecifiedHuffman() (huffman.c:372) is used to create the Huffman table from the bits and the huffvals passed into it.

MakeDecoderHuffman() (huffman.c:399) creates the decoder tables from the Xhuff structure.

ReadHuffman() (huffman.c:414) reads in a Huffman structure from the currently open stream.

WriteHuffman() (huffman.c:454) writes the Huffman out to the stream. This Huffman structure is written from the Xhuff structure.

DecodeHuffman() (huffman.c:485) returns the value decoded from the Huffman stream. The Dhuff must be loaded before this function be called.

EncodeHuffman() (huffman.c:534) places the Huffman code for the value onto the stream.

MakeXhuff() (huffman.c:572) creates a Huffman structure and puts it into the current slot.

MakeEhuff() (huffman.c:591) creates a Huffman structure and puts it into the current slot.

MakeDhuff() (huffman.c:610) creates a Huffman structure and puts it into the current slot.

UseACHuffman() (huffman.c:629) installs the appropriate Huffman structure from the CImage structure.

UseDCHuffman() (huffman.c:650) installs the DC Huffman structure from the CImage structure.

SetACHuffman() (huffman.c:671) sets the CImage structure contents to be the current Huffman structure.

SetDCHuffman() (huffman.c:687) sets the CImage structure contents to be the current Huffman structure.

PrintHuffman() (huffman.c:703) prints out the current Huffman structure.

PrintTable() (huffman.c:784) prints out a table to the screen. The table is assumed to be a 16x16 matrix represented by a single integer pointer.

5.2.4 io.c

```
static BUFFER *MakeXBuffer(int, int);
static void WriteXBuffer(int, int *, BUFFER *);
static void ReadXBuffer(int, int *, BUFFER *);
static void ReadResizeBuffer(int, BUFFER *);
static void FlushBuffer(BUFFER *);
static void BlockMoveTo(void);
static void ReadXBound(int, int *, BUFFER *);
static void WriteXBound(int, int *, BUFFER *);

static void LineMoveTo(void);

extern void ReadBlock(int *);
extern void WriteBlock(int *);
extern void ResizeIob(void);
extern void RewindIob(void);
extern void FlushIob(void);
extern void SeekEndIob(void);
extern void CloseIob(void);
extern void MakeIob(int, int, int);
extern void PrintIob(void);
extern void InstallIob(int);
extern void TerminateFile(void);

extern void ReadLine(int, int *);
extern void ReadPreambleLine(int, int *);
extern void WriteLine(int, int *);
extern void LineResetBuffers(void);
```

MakeXBuffer() (io.c:81) constructs a holding buffer for the stream input. It takes up a size passed into it and returns the appropriate buffer structure.

ResizeIob() (io.c:116) is used to resize the Iob height and width to conform to that of the CScan. This is used for the dynamic Number-of-lines rescaling.

MakeIob() (io.c:136) is used to create an Iob structure for use in the CScan structure. An Iob consists of several Buffer structures with some additional sizing information. The input flags set up the parameters of the stream.

PrintIob() (io.c:222) is used to print the current input buffer to the stdio stream.

WriteXBuffer() (io.c:249) writes out len elements from storage out to the buffer structure specified. This is can result in a multiple of len bytes being written out depending on the element structure.

ReadXBuffer() (io.c:321) is fetches len amount of elements into storage from the buffer structure. This may actually amount to an arbitrary number of characters depending on the word size.

ReadResizeBuffer() (io.c:404) reads len bytes from the stream and puts it into the buffer.

FlushBuffer() (io.c:453) saves the rest of the bytes in the buffer out to the disk.

ReadBlock() (io.c:481) is used to get a block from the current Iob. This function returns (for the JPEG case) 64 bytes in the store integer array. It is stored in row-major form; that is, the row index changes least rapidly.

WriteBlock() (io.c:531) writes an array of data in the integer array pointed to by store out to the driver specified by the IOB. The integer array is stored in row-major form, that is, the first row of (8) elements, the second row of (8) elements....

BlockMoveTo() (io.c:578) is used to move to a specific vertical and horizontal location (block wise) specified by the current Iob. That means you set the current Iob parameters and then call BlockMoveTo().

RewindIob() (io.c:617) brings all the pointers to the start of the file. The reset does not flush the buffers if writing.

FlushIob() (io.c:674) is used to flush all the buffers in the current Iob. This is done at the conclusion of a write on the current buffers.

SeekEndIob() (io.c:709) is used to seek the end of all the buffers in the current Iob. This is done at the conclusion of a write, to avoid DNL problems.

CloseIob() (io.c:749) is used to close the current Iob.

ReadXBound() (io.c:762) reads nelem elements of information from the specified buffer. It detects to see whether a load is necessary or not, or whether the current buffer is out of the image width bounds.

WriteXBound() (io.c:797) writes the integer array input to the buffer. It checks to see whether the bounds of the image width are exceeded, if so, the excess information is ignored.

InstallIob() (io.c:819) is used to install the Iob in the current scan as the real Iob.

TerminateFile() (io.c:838) is a function that ensures that the entire file defined by the Iob is properly flush with the filesize specifications. This function is used when some fatal error occurs.

ReadLine() (io.c:895) reads in the lines required by the lossless function. The array *store should be large enough to handle the line information read.

In total, there should be (HORIZONTALFREQUENCY+1) * nelem (VERTICALFREQUENCY+1) elements in the *store array. This forms a matrix with each line consisting of:

[PastPredictor 1 element] nelem* [HORIZONTALFREQUENCY elements]

And there are (VERTICALFREQUENCY+1) of such lines in the matrix:

Previous line (2**Precision-1) if beyond specifications of window Active line 1... ... Active line VERTICALFREQUENCY...

ReadPreambleLine() (io.c:941) reads the first line of the *store array for the WriteLine() companion command. It reads it so that prediction can be accomplished with minimum effort and storage.

This command is executed before decoding a particular line for the prediction values; WriteLine() is called after the decoding is done.

WriteLine() (io.c:979) is used to write a particular line out to the IOB. The line must be of the proper form in the array for this function to work.

In total, there should be (HORIZONTALFREQUENCY+1) * nelem (VERTICALFREQUENCY+1) elements in the *store array. This forms a matrix with each line consisting of:

[PastPredictor 1 element] nelem* [HORIZONTALFREQUENCY elements]

And there are (VERTICALFREQUENCY+1) of such lines in the matrix:

Previous line (2**Precision-1) if beyond specifications of window Active line 1... ... Active line VERTICALFREQUENCY...

LineResetBuffers() (io.c:1028) resets all of the line buffers to the (2D̂ataPrecision-1) state. The previous state is the default prediction. This commmand is used for resynchronization. The implementation here does a trivial resetting.

EFUNC

LineMoveTo() (io.c:1052) is used to move to a specific vertical and horizontal location (line wise) specified by the current Iob. That means you set the current Iob parameters and then call LineMoveTo().

5.2.5 chendct.c

```
extern void ChenDct(int *, int *);
extern void ChenIDct(int *, int *);
```

ChenDCT() (chendct.c:74) implements the Chen forward dct. Note that there are two input vectors that represent x=input, and y=output, and must be defined (and storage allocated) before this routine is called.

ChenIDCT() (chendct.c:192) implements the Chen inverse dct. Note that there are two input vectors that represent x=input, and y=output, and must be defined (and storage allocated) before this routine is called.

5.2.6 leedct.c

```
extern void LeeIDct(int *, int *);
extern void LeeDct(int *, int *);
```

LeeIDct (leedct.c:97) is implemented according to the inverse dct flow diagram in the paper. It takes two input arrays that must be defined before the call.

LeeDct (leedct.c:312) is implemented by reversing the arrows in the inverse dct flow diagram. It takes two input arrays that must be defined before the call.

5.2.7 lexer.c

```
extern void initparser(void);
extern void parser(void);

static int hashpjw(char *);
static LINK * MakeLink(int, char *, int);
static struct id * enter(int, char *, int);
static int getint(void);
static char * getstr(void);
```

initparser() (lexer.c:354) is used to place the Reserved Words into the hash table. It must be called before the parser command is called.

hashpjw() (lexer.c:383) returns a hash value for a string input.

MakeLink() (lexer.c:407) is used to construct a link object. The link is used for the hash table construct.

enter() (lexer.c:443) is used to enter a Reserved Word or ID into the hash table.

getint() (lexer.c:487) takes an integer from the input.

getstr() (lexer.c:507) gets a string from the input. It copies the string to temporary storage before it returns the pointer.

parser() (lexer.c:599) handles all of the parsing required for the Command Interpreter. It is basically a while statement with a very large case statement for every input. The Command Interpreter is essentially driven by the keywords. All unmatched values such as integers, strings, and brackets, are ignored.

5.2.8 marker.c

```
extern void WriteSoi(void);
extern void WriteEoi(void);
extern void WriteJfif(void);
extern void WriteSof(void);
extern void WriteDri(void);
extern void WriteDqt(void);
extern void WriteSos(void);
extern void WriteDht(void);
extern void ReadSof(int);
extern void ReadDqt(void);
extern void ReadDht(void);
extern void ReadDri(void);
extern void ReadDnl(void);
extern int CheckMarker(void);
extern void CheckScan(void);
extern void ReadSos(void);
extern void MakeConsistentFrameSize(void);
```

WriteSoi() (marker.c:57) puts an SOI marker onto the stream.

WriteEoi() (marker.c:72) puts an EOI marker onto the stream.

WriteJfif() (marker.c:87) puts an JFIF APP0 marker onto the stream. This is a generic 1x1 aspect ratio, no thumbnail specification.

WriteSof() (marker.c:122) puts an SOF marker onto the stream.

WriteDri() (marker.c:160) writes out a resync (or restart) interval out to the stream. If unspecified, resync is not enabled.

WriteDnl() (marker.c:178) writes out a number of line marker out to the stream. Note that we must have defined number of lines before as 0.

WriteDqt() (marker.c:197) writes out the quantization matrices in the CImage structure.

WriteSos() (marker.c:251) writes a start of scan marker.

WriteDht() (marker.c:287) writes out the Huffman tables to send.

ReadSof() (marker.c:335) reads a start of frame marker from the stream. We assume that the first two bytes (marker prefix) have already been stripped.

ReadDqt() (marker.c:382) reads a quantization table marker from the stream. The first two bytes have been stripped off.

ReadDht() (marker.c:471) reads a Huffman marker from the stream. We assume that the first two bytes have been stripped off.

ReadDri() (marker.c:516) reads a resync interval marker from the stream. We assume the first two bytes are stripped off.

ReadDnl() (marker.c:536) reads a number of lines marker from the stream. The first two bytes should be stripped off.

CheckMarker() (marker.c:569) checks to see if there is a marker in the stream ahead. This function presumes that ungetc is not allowed to push more than one byte back.

ReadSos() (marker.c:595) reads in a start of scan from the stream. The first two bytes should have been stripped off.

CheckScan() (marker.c:674) sets the MDU dimensions for the CScan structure.

MakeConsistentFrameSize() (marker.c:703) makes a consistent frame size for all of the horizontal and vertical frequencies read.

5.2.9 stream.c

```
extern void initstream(void);
extern void pushstream(void);
extern void popstream(int);
extern void bpushc(int);
extern int bgetc(void);
extern void bputc(int);
extern void mropen(char *, int);
extern void mrclose(void);
extern void mwopen(char *, int);
extern void swbytealign(void);
extern void mwclose(void);
extern long mwtell(void);
extern long mrtell(void);
extern void mwseek(long);
extern void mrseek(long);
extern int megetb(void);
extern void meputv(int,int);
extern int megetv(int);
extern int DoMarker(void);
extern int ScreenMarker(void);
extern void Resync(void);
extern void WriteResync(void);
extern int ReadResync(void);
extern int ScreenAllMarker(void);
extern int DoAllMarker(void);

static int pgetc(void);
```

initstream() (stream.c:133) initializes all of the stream variables– especially the stack. Not necessary to call unless you wish to use more than one stream variable.

pushstream() (stream.c:155) pushes the currently active stream into its predefined location.

popstream() (stream.c:182) gets the specified stream from the location. If there is already a current active stream, it removes it.

brtell() (stream.c:219) is used to find the location in the read stream.

brseek() (stream.c:228) is used to find the location in the read stream.

bpushc() (stream.c:237) is used to unget a character value from the current stream.

bgetc() (stream.c:246) gets a character from the stream. It is byte aligned and bypasses bit buffering.

bgetw() (stream.c:256) gets a msb word from the stream.

bputc() (stream.c:265) puts a character into the stream. It is byte aligned and bypasses the bit buffering.

pgetc() (stream.c:277) gets a character onto the stream but it checks to see if there are any marker conflicts.

mropen() (stream.c:320) opens a given filename as the input read stream.

mrclose() (stream.c:357) closes the input read stream.

mwopen() (stream.c:375) opens the stream for writing. Note that reading and writing can occur simultaneously because the read and write routines are independently buffered.

swbytealign() (stream.c:407) flushes the current bit-buffered byte out to the stream. This is used before marker codes.

mwclose() (stream.c:427) closes the stream that has been opened for writing.

mwtell() (stream.c:447) returns the bit position on the write stream.

mrtell() (stream.c:460) returns the bit position on the read stream.

mwseek (stream.c:473) returns the bit position on the write stream.

mrseek() (stream.c:505) jumps to a bit position on the read stream.

megetb() (stream.c:521) gets a bit from the read stream.

meputv() (stream.c:543) puts n bits from b onto the writer stream.

megetv() (stream.c:593) gets n bits from the read stream and returns it.

DoMarker() (stream.c:631) performs marker analysis. We assume that the Current Marker head has been read (0xFF) plus top information is at marker_read_byte.

ScreenMarker() (stream.c:770) looks to see what marker is present on the stream. It returns with the marker value read.

Resync() (stream.c:826) does a resync action on the stream. This involves searching for the next resync byte.

WriteResync() (stream.c:882) writes a resync marker out to the write stream.

ReadResync() (stream.c:898) looks for a resync marker on the stream. It returns a 0 if successful and a -1 if a search pass was required.

ScreenAllMarker() (stream.c:944) looks for all the markers on the stream. It returns a 0 if a marker has been found, -1 if no markers exist.

DoAllMarker() (stream.c:963) is the same as ScreenAllMarker except we assume that the prefix marker-byte (0xff) has been read and the second byte of the prefix is in the marker_byte variable. It returns a -1 if there is an error in reading the marker.

5.2.10 transform.c

```
extern void ReferenceDct(int *, int *);
extern void ReferenceIDct(int *, int *);
extern void TransposeMatrix(int *, int *);
extern void Quantize(int *, int *);
extern void IQuantize(int *, int *);
extern void PreshiftDctMatrix(int *, int);
extern void PostshiftIDctMatrix(int *, int);
extern void BoundDctMatrix(int *, int);
extern void BoundIDctMatrix(int *, int);
extern void ZigzagMatrix(int *, int *);
extern void IZigzagMatrix(int *, int *);
extern int *ScaleMatrix(int, int, int, int *);
extern void PrintMatrix(int *);
extern void ClearMatrix(int *);

static void DoubleReferenceDct1D(double *,double *);
static void DoubleReferenceIDct1D(double *, double *);
static void DoubleTransposeMatrix(double *, double *);
```

ReferenceDct() (transform.c:85) does a reference DCT on the input (matrix) and output (new matrix).

DoubleReferenceDCT1D() (transform.c:124) does a 8 point dct on an array of double input and places the result in a double output.

ReferenceIDct() (transform.c:145) is used to perform a reference 8x8 inverse dct. It is a balanced IDCT. It takes the input (matrix) and puts it into the output (newmatrix).

DoubleReferenceIDct1D() (transform.c:184) does an 8 point inverse dct on ivect and puts the output in ovect.

TransposeMatrix (transform.c:205) transposes an input matrix and puts the output in newmatrix.

DoubleTransposeMatrix (transform.c:223) transposes a double input matrix and puts the double output in newmatrix.

Quantize() (transform.c:241) quantizes an input matrix and puts the output in qmatrix.

IQuantize() (transform.c:273) takes an input matrix and does an inverse quantization and puts the output int qmatrix.

PreshiftDctMatrix() (transform.c:298) subtracts 128 (2048) from all 64 elements of an 8x8 matrix. This results in a balanced DCT without any DC bias.

PostshiftIDctMatrix() (transform.c:314) adds 128 (2048) to all 64 elements of an 8x8 matrix. This results in strictly positive values for all pixel coefficients.

BoundDctMatrix() (transform.c:329) clips the Dct matrix such that it is no larger than a 10 (1023) bit word or 14 bit word (4095).

BoundIDctMatrix (transform.c:350) bounds the inverse dct matrix so that no pixel has a value greater than 255 (4095) or less than 0.

IZigzagMatrix() (transform.c:369) performs an inverse zig-zag translation on the input imatrix and places the output in omatrix.

ZigzagMatrix() (transform.c:388) performs a zig-zag translation on the input imatrix and puts the output in omatrix.

ScaleMatrix() (transform.c:407) does a matrix scale appropriate to the old Q-factor. It returns the matrix created.

PrintMatrix() (transform.c:438) prints an 8x8 matrix in row/column form.

ClearMatrix() (transform.c:461) sets all the elements of a matrix to be zero.

Bibliography

[1] William B. Pennebaker and Joan L. Mitchell, “Standardization of Color Image Data Compression. I. Sequential Coding,” preprint *Electronic Imaging ‘89 East*, Boston, MA, October 1989.

[2] Graham P. Hudson, Hiroshi Yasuda, and Istvan Sebestyen, “The International Standardisation of a Still Picture Compression Technique,” *GLOBECOM ‘88*, JPEG-249, November, 1988.

[3] U.S. Senate, *Committee on Interstate and Foreign Commerce, Advisory Committee on Color Television.* “The Present Status of Color Television – *Report of the Advisory Committee on Color Television to the Committee on Interstate and Foreign Commerce*,” United States Senate, 81st Congress, 2nd Session, Document No., 197, Washington D.C., 1950.

[4] Boris Townsend, *PAL Colour Television*, Cambridge at the University Press, 1970.

[5] Arun N. Netravali and Barry G. Haskell, *Digital Pictures*, Plenum Press, 1988.

[6] Byeong G. Lee, “A New Algorithm to Compute the Discrete Cosine Transform,” *IEEE Trans. Acoust., Speech, Signal Processing*, vol. **ASSP-32**, No. 6, pp. 1243-1245, December 1984.

[7] W. A. Chen, C. Harrison, and S. C. Fralick, “A Fast Computational Algorithm for the Discrete Cosine Transform,” *IEEE Trans. Commun.*, vol. **COM-25**, No. 9, pp. 1004-1011, September, 1977. (See also COM-31, pp. 121-123.)

[8] Ephraim Feig, “A fast scaled-DCT algorithm,” *SPIE Image Processing Algorithms and Techniques*, vol. **1244**, pp. 2-13, 1990.

[9] David A. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” *Proc. IRE*, pp. 1098-1101, September 1952.

[10] JPEG Committee Draft ISO/IEC CD 10198-1 (3/15/1991).

[11] Gregory K. Wallace, “The JPEG Still Picture Compression Standard,” *Communications of the ACM*, vol. **34**, No. 4, pp. 30-44, April 1991.

[12] William B. Pennebaker and Joan L. Mitchell, *JPEG Still Image Data Compression Standard*, Van Nostrand Reinhold, 1993.

[13] Eric Hamilton, “JPEG File Interchange Format, Version 1.02,” September 1, 1992.

Index