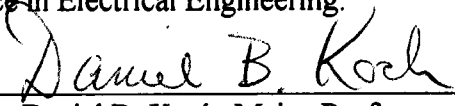
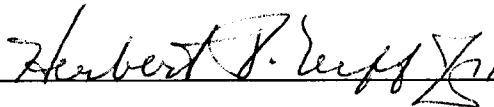


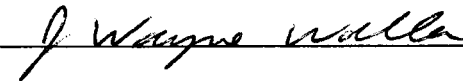
To the Graduate council:

I am submitting herewith a thesis written by Constantine Peter Bandy entitled "A Low Bandwidth, Still Image Transmission System." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science in Electrical Engineering.


Dr. Daniel B. Koch, Major Professor

We have read this thesis
and recommend its acceptance:





Accepted for the Council:


Associate Vice Chancellor
and Dean of The Graduate School

A LOW BANDWIDTH, STILL IMAGE TRANSMISSION SYSTEM

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

Constantine Peter Bandy

December 1994

DEDICATION

This thesis is dedicated to my loving wife

Beth Susan Bandy

who has encouraged and supported

me throughout my schooling.

ABSTRACT

In the event of a major earthquake in or near Tennessee Wing of the Civil Air Patrol (TN CAP) is tasked to perform aerial reconnaissance of Tennessee roadways and bridges. To enhance the analysis of aerial observations, TN CAP has proposed broadcasting amateur television from aircraft cockpits to ground receiving stations. However, because of the large bandwidth required for video transmission and the low transmitter power available in the small single engine CAP aircraft cockpits, the range of video transmission is very limited. As an alternative to amateur television transmission, this thesis develops a software based, near real time, continuous-tone still image transmission system.

A digitized copy of a single, color picture at television resolution contains nearly one million bytes. Digital imaging is, therefore, not often implemented due to high storage or transmission costs, even when image capture and display devices are quite affordable. However, by employing a high quality image compression technique, images may be compressed from 1/10 to greater than 1/50 [21] their uncompressed size without visibly affecting image quality. These compressed images can then be transmitted over a low bandwidth channel in a reasonable time.

The major issues in developing this system were the image compression and data transmission techniques. The selected image compression technique is based on the Joint Photographic Expert Group's Still Image Data Compression Standard, known officially as ISO DIS 10918-1 or CCITT Recommendation T.81.

The system's data link will use the Civil Air Patrol's high speed (9600 bits per second) packet radio (CCITT X.25) for air to ground broadcast. The prototype system described in this work employs a telephone modem link using various modulation techniques: 2400 bps (Quadrature Amplitude Modulation, CCITT V.22); 14,400 bps (Trellis coded Modulation/Quadrature Amplitude Modulation CCITT V.32bis); and data compression (CCITT V.42bis) and error correction (CCITT V.42).

TABLE OF CONTENTS

SECTION	PAGE
1.0 INTRODUCTION.....	1
1.1 General.....	1
1.2 System Design Goal.....	1
1.3 System Description.....	2
1.4 Image Compression and Display.....	4
1.5 Data Transmission Aspects.....	6
1.6 Thesis Organization.....	6
2.0 BACKGROUND.....	8
2.1 Color and Resolution.....	8
2.2 Entropy of an Image.....	9
2.3 Image Compression Alternatives.....	9
2.4 Data Transmission Alternatives.....	13
3.0 THE JPEG BASELINE IMAGE COMPRESSION METHOD.....	20
3.1 General.....	20
3.2 The Two Dimensional Discrete Cosine Transform.....	20
3.3 Quantization of the Discrete Cosine Transformation Coefficients.....	21
3.4 Entropy Coding.....	23
3.5 Compressed Image Data Format.....	27

SECTION	PAGE
4.0 THE IMAGE COMPRESSION ALGORITHM.....	30
4.1 Software Considerations and Image Formatting	30
4.2 Discrete Cosine Transforms and Quantization Algorithms.....	31
4.3 Entropy Coding Algorithm.....	33
5.0 THE IMAGE DECOMPRESSION ALGORITHM	44
5.1 General	44
5.2 Identifying and Decoding the Variable Length Encoded Data Stream	45
6.0 THE DATA LINK	55
6.1 General	55
6.2 Initializing the Serial Data Port and Modem.....	55
6.3 Data Buffering.....	61
6.4 Hardware Flow Control.....	64
6.5 The Modem Interrupt Service Function	65
6.6 Establishing a Data Link Connection.....	65
7.0 DISPLAYS, USER COMMANDS, AND SYSTEM INTEGRATION	68
7.1 The System Displays and Video Memory	68
7.2 The Setup and Terminal Screens	72
7.3 Image Display Screen.....	73
7.4 User Commands.....	74
7.5 System Integration	74

SECTION	PAGE
8.0 RESULTS AND CONCLUSIONS.....	79
8.1 System Performance Testing.....	79
8.2 Conclusions.....	81
8.3 Future Directions.....	84
LIST OF REFERENCES.....	86
APPENDICES	90
APPENDIX A. CODE AND QUANTIZATION TABLES	91
APPENDIX B. PROGRAM SOURCE CODE.....	110
VITA	155

LIST OF TABLES

TABLE		PAGE
1.	Image Resolution and Uncompressed Image Size.....	4
2.	Huffman Coding of DC Coefficients	25
3.	Huffman Coding of AC Coefficients	26
4.	Image Resolution, File, and Segment Attributes	31
5.	Number of Multiplications for Various 2-D DCT Algorithms	32
6.	System Operating Commands	74
7.	Source Code Files, Functions, and Function Descriptions.....	77
8.	Overall System Performance	80
9.	Prototype System Performance Results for Compression, Decompression, and Image Quality	82
10.	Compressed Image Data Transmission Time Using Various Transfer Methods.....	83

LIST OF FIGURES

FIGURE	PAGE
1. System Block Diagram	3
2. Image Compression Processing Steps	5
3. Predictors for Lossless Coding	11
4. DCT Basis Functions	14
5. Signal-Space Diagram of (a) Uncoded 16-ary QAM and (b) 32-ary TCM	17
6. Frame Format For Bit Oriented Protocols	17
7. Zigzag Ordering of DCT Coefficients	22
8. Sample Compression Values of Quantized Luminance DCT Coefficients	24
9. Application Specific Data Field	29
10. Forward DCT Algorithm Implementation	34
11. Inverse DCT Algorithm Implementation	35
12. DCT <i>Normalization</i> Matrix (N)	36
13. Illustration of Huffman Code Pointer	37
14. Huffman Code Pointer Flow Chart	38
15. Source Code For Creating Huffman Code Pointers	39
16. Illustration of Bit Stiffing Routine	41
17. Flow Chart for Bit Stuffing Routine	42
18. Bit Stuffing Routine	43
19. Variable Length DC Code Identification Source Code	45
20. Illustration of Variable Length DC Code Identification Routine	46
21. Source Code For Bits Replacement Routine	47
22. Illustration of Bit Shifting and Replenishing Routine	48
23. Illustration of Decoding AC Code Entries	51
24. Source Code for AC Code Word Search	52
25. Image Display Characteristics	54

FIGURE	PAGE
26. Illustration of an Internal Modem with a FIFO UART	58
27. Functions CheckTypePort and SetSpeed Source Code	59
28. Function SetInterrupts Source Code	60
29. Input FIFO Buffer Operation	61
30. SetBuffers Function Source Code and Operation	62
31. Functions Read and Write Source Code	63
32. Functions Queue and Dequeue Source Code	64
33. Modem Interrupt Service Function Source Code	66
34. Terminal Display Screen	69
35. Setup Display Screen	70
36. Image Display Screen	71
37. Flow Chart for Function void main(void)	76

1.0 INTRODUCTION

1.1 General

In the event of an earthquake affecting Tennessee, the Tennessee Emergency Management Agency has requested disaster recovery assistance from the Tennessee Wing of the Civil Air Patrol (TN CAP). Specific functions assigned to TN CAP include: reconnaissance of major Tennessee roadways and bridges; evaluation of smaller airfields for integrity of runways and availability of water, power and fuel; courier runs and radio relay. Aerial reconnaissance will be used to determine the continuity of roadways and the integrity of runways at small air fields.

This thesis develops a prototype of a near real time, continuous-tone, still image transmission system suitable for assisting the TN CAP perform their low level aerial earthquake damage assessment mission. By mounting a video camera in a low flying, single engine TN CAP airplane cockpit, roadways and airports can be observed. Video image frames can then be captured, digitized, compressed and transmitted, via a low bandwidth radio frequency (RF) link, to TN CAP mission control where they can be stored, reconstructed and displayed. The information contained in these images will be used to direct needed relief efforts. The prototype system will be suitable for fixed image transmission using a telephone and modem.

1.2 System Design Goal

The desired system will transmit images of acceptable quality, such that received images provide sufficient detail to identify ground damage to roadway surfaces, such as rocks blocking traffic lanes or inoperable bridges. When low level aerial reconnaissance is performed the target area is under surveillance for only a short time. Multiple images of an area may be needed to provide a complete picture. Therefore, the system must provide a high quality image in a short time. This criteria is influenced by the picture resolution, color content, image compression technique, and data transmission method.

A single, digitized, color picture at television resolution contains nearly one million bytes. Thus, a viable digital image transmission system using a low bandwidth channel will

require significant image compression without visibly affecting image quality. Assuming a twenty-five to one compression ratio, a one megabyte color image after compression will contain approximately 40,000 bytes. Also assuming a data transmission rate of 14,400 bits per second, the compressed image can be transmitted in about thirty seconds. Based on these assumptions, the prototype system's primary design goal is to capture, compress, transmit, and display two high resolution color images or several lower resolution images within one minute.

An air crew's primary duty is to pilot the aircraft. Therefore, a secondary system design goal is to minimize operator actions. The system operator's primary interface will be via a computer keyboard. Once the communications channel is established, capture and display of images should need only a single operator action. Compression and transmission of the captured image should also require only a single operator action as well. If the operator desires to terminate transmission, provisions should be available to terminate an image transmission and return to video capture mode without loss of the communications channel.

The TN CAP is an all volunteer organization which receives little outside funding. Therefore, cost is a major factor in providing any auxiliary equipment to enhance mission effectiveness. The TN CAP has a state-wide packet radio, which operates at data rates of up to 9600 bits per second, has a proven reliability, and many members own their own terminal. Therefore, based on its availability, the existing packet radio will be used as the air to ground communications link.

1.3 System Description

A low data rate, video image transmission system's key elements are video frame capturing, image compression, encoding, buffering, and transmission, followed by reception, decoding, decompression, storage and display. A system block diagram is shown in Figure 1.1. Video signals are recorded then captured using a video frame grabber. Next follows image compression, data encoding and data buffering which will be accomplished via software. Data will be transmitted from air to ground, using the existing packet radio and terminal network connector as the data link. The receiving stations will consist of a personal computer and a packet radio and terminal network connector. The image will be stored, decoded, decompressed and displayed using software.

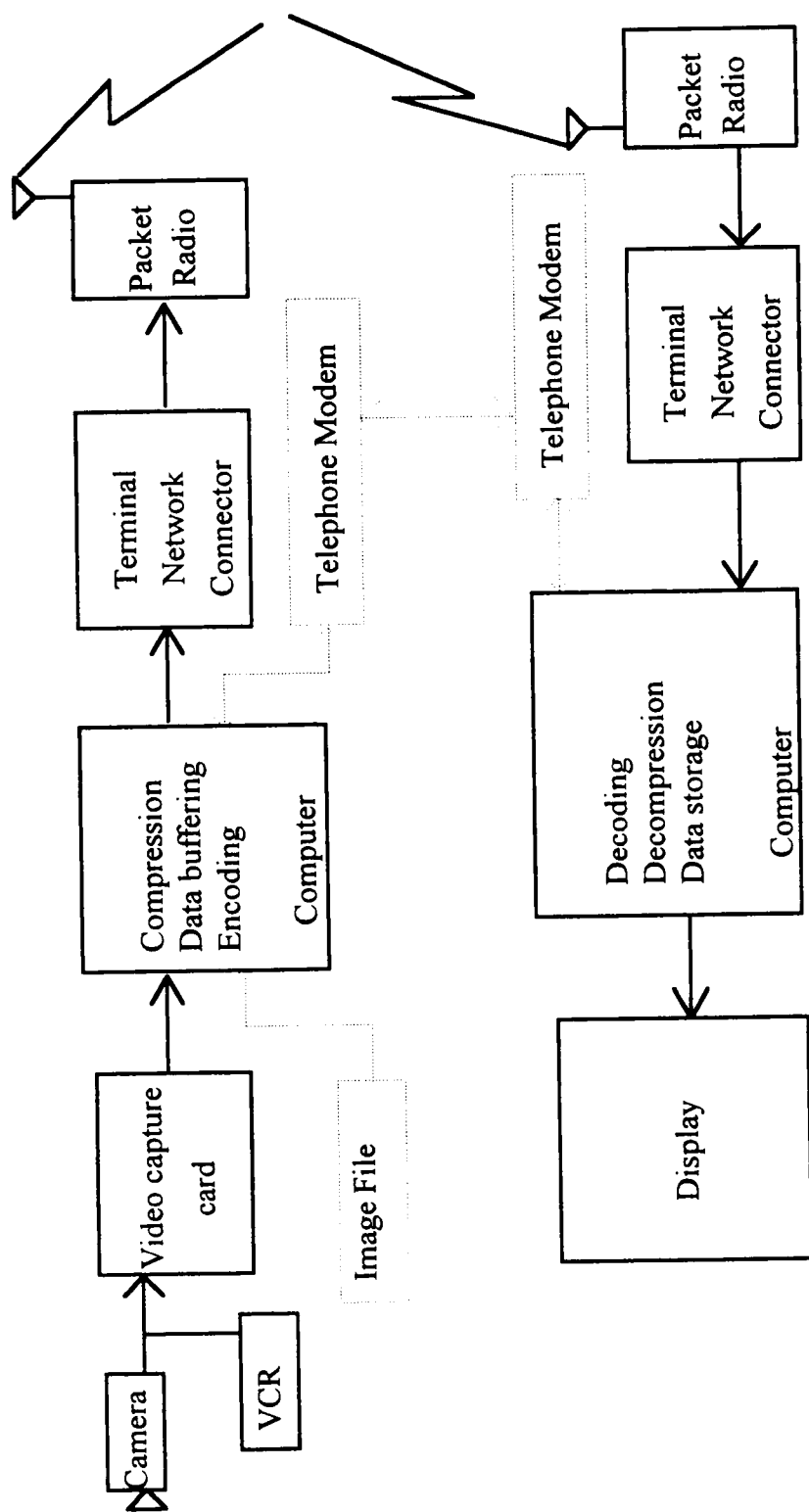


Figure 1. System Block Diagram. The elements contained in locks with dotted lines replace their solid line equivalents in the prototype system.

1.4 Image Compression and Display

An acceptable image compression technique will provide a quality image without restriction on scene content, complexity or range of colors, and have tractable computational complexity to make software implementation viable using today's available personal computer processors. After researching image compression methods, the Joint Photographic Expert Group (JPEG) baseline method was selected. This image compression technique uses a two dimensional discrete cosine transform (2D-DCT). The processing steps for 2D-DCT based image compression are shown in Figure 2.

Image resolution, color content and the image scene determine the amount of information contained in an image. The image resolution influences the time to compress and transmit the image, and the memory required to display the image. Color images, likewise, contain more information than gray scale (black and white) images, thus requiring more time or compression and transmission.

Video images consist of 525 vertical lines of which about 490 are actually active in synthesizing the image. These decompressed images will be displayed using the computer's monitor. The ability of a computer monitor to display images is influenced by the type of video graphics adapter, and the adapter's available memory. Most commonly available computer displays are capable of showing true color (each color represented by 24 bits) images in resolutions up to 640 horizontal by 480 vertical picture elements, or pixels. Table 1 shows the system's selected video resolutions and the amount of information associated with the image.

Table 1. Selected Image Resolutions and Uncompressed Image Sizes

Image Resolution	Image Height (Pixels)	Image Width (Pixels)	Uncompressed Gray Scale Image (Bytes)	Uncompressed Color Image (Bytes)
Low	160	120	19200	57600
Medium	320	240	76800	230400
High	640	480	307200	921600

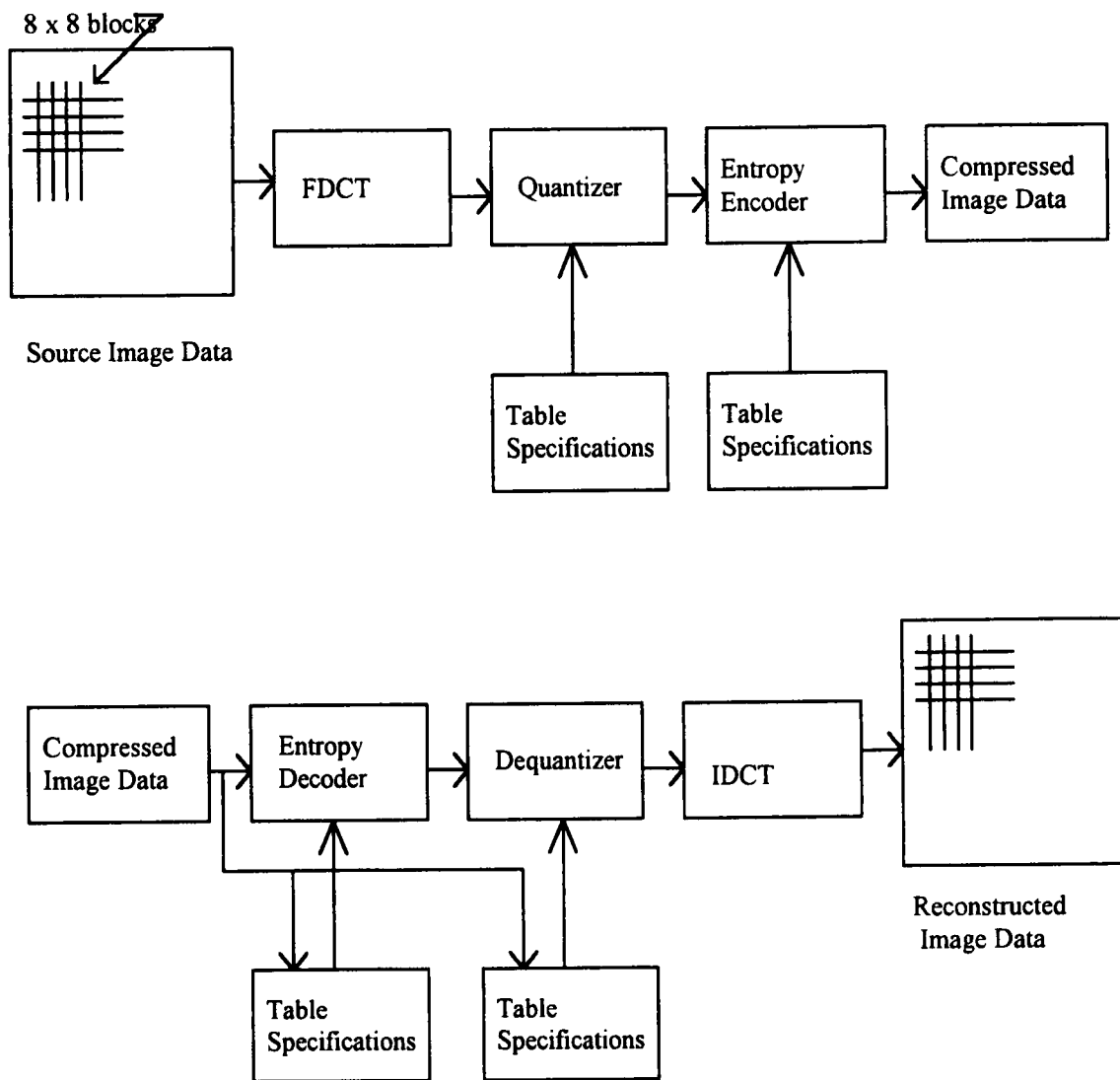


Figure 2. Image Compression Processing Steps

1.5 Data Transmission Aspects

Air to ground transmission of the compressed image will be via the TN CAP's existing packet radio which utilizes CCITT X.25 protocol. This protocol permits the error free transfer of data at speed up to 9600 bits per second between two users or, in broadcast mode, between one user and many users. The system is used statewide, has a high availability, and demonstrated reliability.

However, for this thesis, a fixed communications link prototype is implemented. This fixed data link consists of a telephone modem data link using various modulation techniques and protocols. The types of modem connections considered are 2400 bit per second (bps) using Quadrature Amplitude Modulation (V.22); 14,400 bps using Trellis Coded Modulation/Quadrature Amplitude Modulation (V.32bis); and the use of data compression (V.42bis) and error correcting (V.42) protocols. Image compression and data transmission can occur either simultaneously or serially. However, the speed of the image compression technique, the time needed to transmit the data, and the availability of data buffering influence the choice of a particular data transmission method. This system uses data buffering and interrupt driven input/output devices to permit data compression and decompression simultaneously with data transmission.

1.6 Thesis Organization

This thesis describes the image compression and transmission system's development and comments on its performance. Chapter 2, Background, examines the explored image compression and data transmission alternatives. Next, chapter 3 describes the selected image compression method. Chapter 4, highlights the implementation of the image compression algorithm, and chapter 5 does the same for the image decompression algorithm. Next, chapter 6 describes the data links operation and implementation. Chapter 7 details the integration of the separate subsystems and describes the main program's operation. The system's integration details the implementing of the all user interfaces, controls, the terminal display and graphics displays. Lastly, an objective review of the system's performance is presented in Chapter 8,

Conclusions. Code and quantization tables used by the system are given in Appendix I, and the system's source code is presented in Appendix II.

2.0 BACKGROUND

2.1 Color and Resolution

An image is composed of individual picture elements (pixels). An image's resolution is usually described by the number of discrete horizontal and vertical pixels. For example 640 x 480 refers to a grid of 640 horizontal by 480 vertical pixels. This size is common for many Video Graphics Array (VGA) displays [10]. According to the Trichrominance theory of color vision, an arbitrary color can be matched by combining appropriate amounts of any three primary colors. In color graphics displays and video receivers the three primary colors of red (R), green (G), and blue (B) are superimposed to produce the desired color. In a monochrome display, each pixel is set to a value corresponding to the brightness or luminance of an image.

In 1952, the National Television System Committee (NTSC) chose a color coordinate system of Y, I and Q for color television image transmission. The I and Q signals provide the chrominance or color information of the image, while Y contains the luminance information. Using this representation for the image was desirable for two reasons. Firstly, since the Y signal contains the luminance information, it alone can be used to display monochrome images. Secondly, it is possible to limit the bandwidth of the I and Q signals without noticeable image degradation.

Computation of Y, I and Q is made using normalized, gamma corrected, values of R, G, and B such that equal amounts of each are used to produce reference white of unity luminance. Gamma correction takes into account the nonlinear relationship between the signal voltage and light intensity found in many TV cameras and displays. The NTSC transformation between normalized, gamma corrected R, G, B and the Y, I, Q coordinate system is [13]:

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.523 & 0.311 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix} \quad (2.1)$$

2.2 Entropy of an Image

Most images contain a significant amount of superfluous information. Once the pixel values (volts) in an image have been quantized into discrete levels, all possible values for each image component, such as luminance, are not equally likely. The pixel value can thus be regarded as a random variable. Although the statistical properties of an image are not usually known, its statistical redundancy can be removed while still providing the information necessary to reproduce the image.

Suppose a pixel is quantized to k bits; each pixel can then be represented uniquely by one of 2^k possible code words. If however, not all quantized values are equally likely, a reduction in the average number of bits needed to specify the image can be achieved by specifying shorter code words for values having a high probability of occurrence and longer code words to those having a low probability of occurrence. This variable word-length coding is referred to as entropy encoding.

For example, if a quantized level n has probability of occurrence of $P(n)$ and is assigned a code word of length $L(n)$ bits, the average code word length is

$$\bar{L} = \sum_{n=0}^{n=2^k} L(n) * P(n) \text{ bits/pixel.} \quad (2.2)$$

The average length has a lower bound which is referred to as the *entropy* of the random variable. This lower bound is:

$$H(N) = - \sum_{n=0}^{n=2^k} P(n) * \log_2 P(n) \text{ bits/pixel.} \quad (2.3)$$

2.3 Image Compression Alternatives

Image compression techniques can be grouped into two broad categories -- lossless (invertible) or lossy (non-invertible). Lossless compression is achieved by removing the statistical dependence in an image and transmitting only the uncorrelated information necessary to reconstruct the original image. While lossy coding schemes, besides removing *statistical redundancy* from an image, also attempt to remove *subjective redundancy* from an image. Subjective redundancy refers to data characteristics that can be removed without complaint by

a human observer (i.e. higher spatial frequencies). Therefore, in a lossy compression scheme, the pixel values of the reconstructed image only approximate the original image.

“ Lossy compression techniques usually capitalize on the *a priori* assumption that their decompressed images will only be presented to the human eye. The lossless compression constraint may arise in applications where preserving exact fidelity is a technical or legal requirement (e.g., in satellite or medical image processing), or where some form of lossy compression has already occurred (e.g., the black/white thresholding done in digitizing facsimile images) and further loss may not be desirable.”[1]

2.3.1 Lossless Image Compression

Lossless image compression systems may remove some of the subjective qualities of an image, however, this reduction is accomplished prior to image compression. Such is the case where the chrominance components of a color video image are bandlimited before their transmission and subsequent compression.

One simple lossless compression method is to determine the relative frequency of each discrete pixel value in the image, and then encode the image using a variable length code. This method provides only minimal compression since it does not take advantage of the pixel-to-pixel correlation found in most images. Its second drawback is the need to determine the picture's statistics before encoding the image.

Lossless compression methods generally employ some method of removing pixel to pixel correlation prior to encoding. The contents of most images reveal that nearby pixels are most heavily correlated. One technique of reducing the correlation between pixels is to take the difference between the current pixel and the previous pixel and entropy encode the difference.

A variation of this simple difference method is presented by Whyte and Saywood [23]. In this variant, a reference pixel is compared to the current pixel value to obtain a prefix and suffix value. These prefix and suffix values are then Huffman encoded. This lossless algorithm was tested with a database containing nine images of varying sizes and resolutions. Each image used eight bits per pixel. The results were compared to similar methods using various coding schemes and to the calculated entropy of the simple difference model. The compression

achieved for the nine images was between 3 and 5 bits per pixel, which was often close to the calculated entropy of the simple difference model. However, this compression is insufficient to consider using a similar technique for the time-sensitive, low-bandwidth transmission system needed for the application presented here.

Additional compression can be achieved by using adjacent pixel values to predict the current pixel value, then encode the difference between the current pixel and the predicted value. One limitation in employing a predictive coding scheme is that only samples previously encoded and, therefore, available to both encoder and decoder may be used for prediction. Several methods of predictive lossless coding were adopted in JPEG's recommended still image compression standard. These methods are shown in Figure 3. The compression achieved from predictive methods on test images is also insufficient to justify their use in this application.

Lossless techniques have been examined for compression of video images. Bizon, et. al [2] explored image compression using differential pulse code modulation (DPCM) of the composite color video signal at four times the color subcarrier frequency. In this technique, data with a high probability of occurrence is represented with short code words, while "not-so-

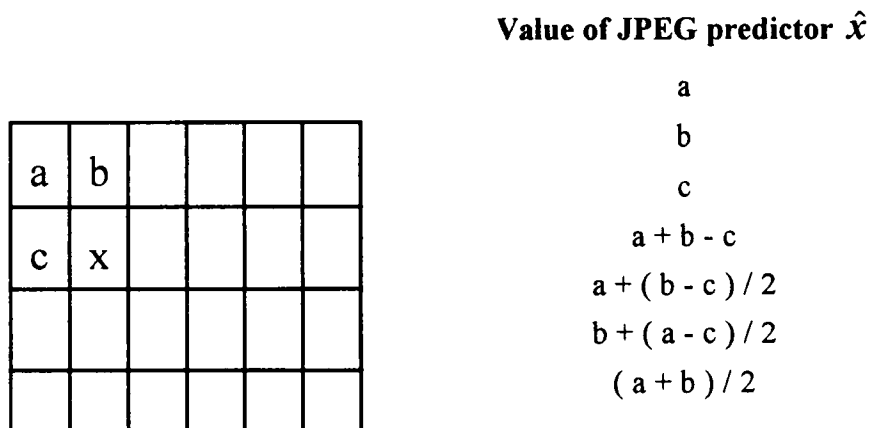


Figure 3. Predictors for lossless coding. In lossless, predictive encoding, the transmitted value for pixel $x_i = \hat{x} - x$, where $\hat{x}$ is the predictor for x . A JPEG system predicts x using one of the above weightings of previously transmitted neighboring pixels a , b , and c .

likely" data is assigned longer code words. Using this compression algorithm and eight bit samples NASA Lewis Research Center obtained a compression of over 75 percent (greater than a 4:1 compression ratio) with no visible degradation. The cost and complexity of this approach prohibit considering this technique for this application.

2.3.2 Lossy Image Compression Techniques

There are a number of competing methods for lossy image compression, capable of shrinking images 1/10 to 1/50 their uncompressed size [21]. In 1986, a joint ISO/CCITT committee, entitled the Joint Photographic Experts Group (JPEG), was formed to start compiling an international standard for continuous-tone, still image compression. A continuous-tone image, unlike facsimile (binary), displays the image with sufficiently small quantization such that colors appear continuous. An eight bit pixel representation is considered the minimum for a continuous-tone image although up to 12 quantization bits may actually be perceived. After evaluation of various methods of lossless image compression techniques, JPEG chose the two dimensional discrete block cosine transform (2-D DCT) as their baseline system.

"Transform coding has been found experimentally to have relatively good capability for bit-rate reduction, and this comes about mainly from two mechanisms. First, not all of the transform domain coefficients need to be transmitted in order to maintain good image quality, and second, the coefficients that are coded need not be represented with full accuracy." [13]

Haines and Chuang [4], using the JPEG baseline method, conducted an extensive study of the acceptability of video compression for life science experiments. They concluded that "the JPEG standard provided acceptable still-frame imagery of plants [vegetation] at compression as high as 120:1, depending upon particular scene content. Resolution by itself was the most important image characteristic for the still-frame imagery, ...".

The 2-D DCT lends itself particularly well to image compression for two main reasons. First, it produces coefficients that are largely uncorrelated, allowing each basis function coefficient to be treated independently without loss of compression efficiency. Furthermore, since the 2-D DCT basis functions increase in horizontal frequency from left to right and

vertical frequency from top to bottom, their coefficients can be quantized using visually-weighted values. Figure 4 shows the 2-D DCT basis functions.

The JPEG baseline sequential method is capable of achieving the desired compression, and transform coding is computationally feasible using today's fastest personal computers; therefore, it was selected as the backbone of this image transmission system. The JPEG baseline sequential method is described in Chapter 3, and its implementation is detailed in Chapter 4 and 5.

2.4 Data Link Alternatives

Two methods will be utilized to transmit compressed images for this application: one for air to ground transmission, and one for fixed communications. For communications from air to ground, the Civil Air Patrol's packet radio (CCITT X.25 protocol) network will be used. The CAP's packet radios will accept a high speed modem input, providing the modem's application software handles the X.25 protocol. A description of the X.25 protocol can be found in [11]. Since only a limited number of member owned packet radio stations are being operated, for this prototype fixed data link system, a more conventional and available telephone and modem will be used.

Telephone modems represent an effective data transmission method between personal computers at fixed locations. Modems are primarily concerned with transmission of data across a physical link. Today's more advanced modems, however, have built in microprocessors capable of providing data compression and error correction, thus ensuring reliable high speed data transmission. In addition, these microprocessors also allow the use of sophisticated modulation techniques to effectively exploit the full voice bandwidth and provide transmission at rates up to 14,400 bits per second or higher.

The first modems were external and interfaced the personal computer (data terminal equipment) through a serial data link. These modems operated at low speeds (300 to 1200 bps) and were adequate for asynchronous communications - where characters are sent intermittently such as a keystroke, or in bursts such as a line of text. The modem framed the

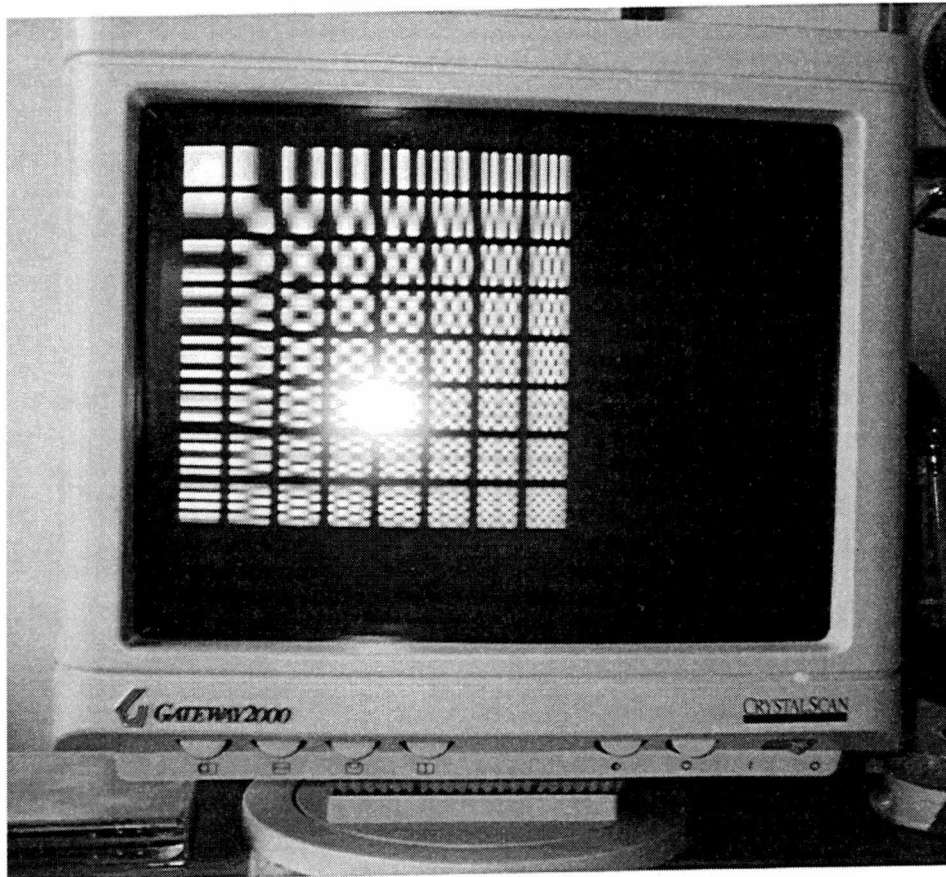


Figure 4. DCT Basis Functions.

data by adding a start bit, one or more stop bits, and if desired, a parity bit. Integrity of the data was not guaranteed. The 300 bps modem used Frequency Shift Keying (FSK) as its modulation technique (CCITT V.21 and Bell 103) while the 1200 bps modems used Differentially encoded Phase Shift Keying (DPSK) specified in (CCITT V.22 and Bell 212A).

The next generation of modems used a Quadrature Amplitude Modulation technique where each symbol transmitted four information bits. Commercially available 2400 Hz modems conform to CCITT V.22bis. The 2400 bps data rate made the modem a feasible alternative for transferring files between computers. However, unlike text transmissions, errors in file transmission are generally unacceptable. A number of file transfer protocols (i.e.; Kermit, Xmodem, Ymodem, ...) have been developed which detect transmission errors and request retransmission. File transfer protocols, which are implemented by the data transmission equipment (Personal Computer), permit nearly error free transmission of data.

File transfer protocols break the file into a series of packets. Each packet contains a file header, a portion of the data, and a check sum. Protocols adhere to a specific set of rules that govern the start of the transfer, packet size, packet content, acknowledgment, and transfer completion. The price for using a file transfer protocol is a reduced data throughput.

To achieve data transmission rates of 4800 and 9600 bps, the follow-on generation of modems employ a Trellis Coded Modulation (TCM) scheme which is specified in V.32. The TCM scheme combines multilevel/phase modulation with a state-oriented trellis coding scheme. The modulation technique defined in V.32 is QAM. The addition of echo cancellation and other line conditioning techniques enable V.32bis modems to achieve data rates up to 14,400 bps. Modems meeting the V.32 series standards fall back to a lower data rate if line conditions degrade.

In QAM, an In-phase and Quadrature-phase carrier signal are transmitted. Both are varied in amplitude and phase. For a 9600 bps QAM modem, the phase and the amplitude states are assigned two data bits each and the symbol rate is 2400 Hz. The basic form for QAM is:

$$s(t) = c_m \cos[\omega t + \theta_m] \quad (2.4)$$

However, to more effectively exploit the entire channel bandwidth, V.32 series modems combine Quadrature Amplitude Modulation with trellis encoding. Coding schemes generally employed to improve error performance do so at the expense of channel bandwidth. In the case of a band limited channel like that of a telephone link, attempting to increase the transmission bit rate by increasing the size of the symbol set, will be accompanied by an increase in the error rate for the same average output power. By restricting the allowable state transitions, TCM increases the distance between signals likely to be confused; thus reducing the error rate without increasing the average power. The TCM coding gains are, therefore, realized without bandwidth expansion, but with an increase in decoder complexity [18]. Figure 5 shows the signal set for both the V.32 uncoded 16-ary QAM and TCM/QAM.

To encode using TCM, V.32 increases the signal set from sixteen to thirty two symbols. The data bits are then encoded at a higher rate and mapped into the set of thirty two channel symbols. Error performance is now dependent on the minimum distance between members of the set of allowable code symbol sequences. It has been shown that using TCM a net error performance gain of 3 to 6 dB can be achieved without bandwidth expansion [20].

To improve the modem's throughput for file transfer, manufacturers next incorporated data compression and error correction features. For a modem to increase data throughput by compressing the data, it must receive data from the PC at a higher data rate than the modem connection, then compress the data so that the information can be passed at the channel rate. These features increase a modem's complexity. In addition to an adjustable rate serial port, the modem's also have transmit and receive data buffers and a central processing unit. The combination of a higher serial port rate and data buffers allow the modem to apply data compression and error correction without interrupting transmission to wait for data.

The transmission of asynchronous bytes of information places a high overhead on data transmission as one start and at least one stop bit are transmitted with each eight bit character. These modems no longer transmit bytes with start and stop bits, but employ High-level Data-Link Control (HDLC), a bit oriented protocol. A detailed example of HDLC can be found in [6]. Figure 6 shows a general frame format for a bit oriented protocol. For frames of

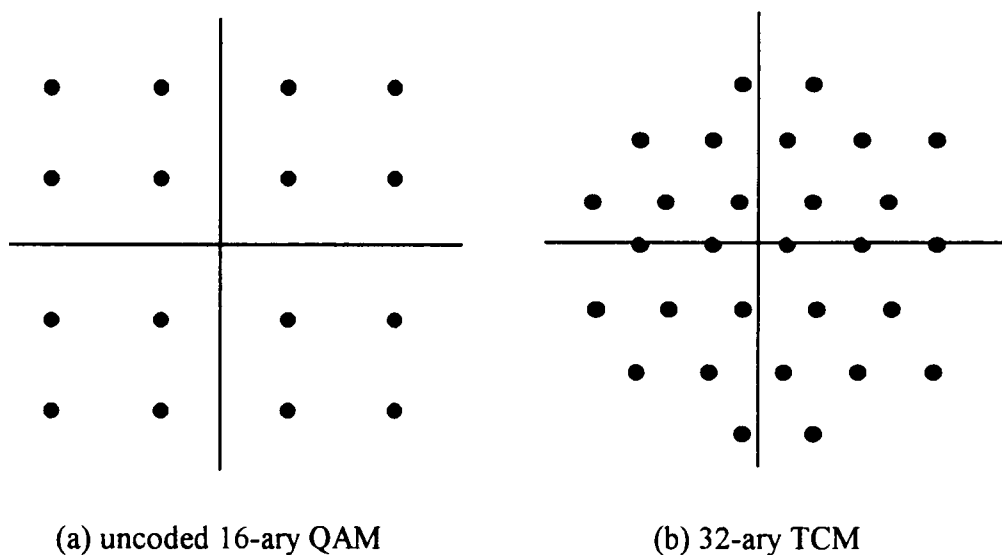
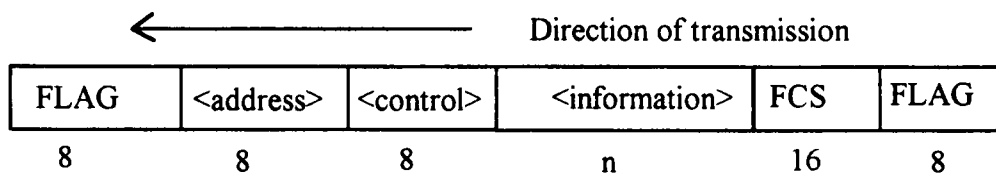


Figure 5. Signal-Space diagram of (a) uncoded 16-ary QAM and (b) 32-ary TCM



- FLAG** is the bit pattern 0111 1110, which marks the beginning and end of a frame
- <address>** contains the ID of the sender or receiver of this frame
- <control>** contains bits that describe the frame type and provide other information of control and flow of data
- <information>** is the network layer PDU; the information field may contain from 0 to any number of bits
- FCS** is the frame check sequence; it is the result of a calculation based on the preceding octets and is used for error detection

Figure 6. Frame Format For Bit Oriented Protocols[6].

reasonable lengths, the removal of the asynchronous start and stop bits more than compensates for the addition of the start and check bit sequences.

In order to achieve interpretability of error correcting and data compressing modems, the CCITT released V.42 and V.42bis standards that specify error correction and data compression protocols, respectively. When these modems connect to one another they negotiate the data transmission rate and protocols to be utilized during data transmission.

The V.42 error correction method is based on a sixteen bit cyclic redundancy check sequence appended to the data. This sequence is calculated by the receiver from the preceding data and compared to the sequence received from the transmitter. If the received sequence matches that of the transmitter, the block is accepted. Otherwise, a negative acknowledgment is sent to the transmitting modem. Upon receipt of this negative acknowledgment the data is retransmitted. When a predetermined number of errors occur in the same block, the connection is broken. Most modems also incorporate a second proprietary, error correction protocol, MNP (Microcomputing Network Protocol) as an alternative or when V.42 error correction is not available. MNP was offered before the adoption of V.42.

The V.42bis data compression technique is a textual substitution scheme that can achieve a 2.3:1 compression ratio of English text. It is based on Ziv and Lempel's second algorithm [24]. Since V.42 data compression is very susceptible to errors, it is required to use V.42 error correction features when employed.

In V.42bis, a code word dictionary is maintained by both the transmitter and receiver. The first received character is placed in the first dictionary slot, the next unique character in the second dictionary slot, etc.. The combination of these characters is then placed in slot 256 of the dictionary. The repeated sub-strings found in English text can thus be exploited to provide data compression. Once the dictionary is full, the least recently used entries, which are not prefixes for other code words, are deleted. The implementation of the coding scheme is left to the modem designer. The length of the code words and the size of the dictionary are negotiated when the initial connection is made.

A V.42 modem can shift at will between compressed mode and "transparent mode" through the use of a two character escape sequence. In transparent mode data is not

compressed, but the dictionary is still updated. In addition to entering transparent mode, the transmitter can, at its discretion, send an escape code sequence that discards the current dictionary. Highly compressed data will not deliver performance improvement and, in the unlikely event that an uncorrelated, full character set is transmitted, data expansion is possible [19].

The Huffman coding used in JPEG image compression like the V.42bis compression is very susceptible to transmission errors. The added marker segments and file header information can benefit from V.42bis compression. Because of scene content, it is unlikely to produce an uncorrelated data sequence. Based on this background information, the system design incorporates a V.32bis modem for its high transmission rate and a V.42bis modem for its error correction and data compression capabilities. Since these specifications are downward compatible, the system will operate with V.22 modems as well, but without an error correction method it will not perform satisfactorily in a noisy environment.

The system's default modem settings are: a bit rate of 14400; a serial port rate of 38,400; error correction; and, data compression. With these settings, the modem will attempt to connect at the highest throughput based on line conditions and the capabilities of the connecting modem. In order to give the system operator maximum flexibility the system allows a knowledgeable user to select any desired modem settings.

Next, the JPEG baseline image compression method is discussed in detail.

3.0 THE JPEG BASELINE IMAGE COMPRESSION METHOD

3.1 General

The JPEG baseline sequential process has three main elements: the discrete cosine block transform, quantization of the transform coefficients, and entropy encoding of the quantized values. In addition to these main elements, the JPEG compressed image data stream contains frame headers and markers which provide the decoder with information necessary to properly interpret the data, retrieve the quantization coefficients, perform the inverse transformation and properly reconstruct the image. Prior to performing JPEG compression, a color image is transformed into its luminance (Y) and chrominance (I and Q) components. These color components are interleaved.

3.2 The Two Dimensional Discrete Cosine Transform

The JPEG baseline system defines an eight by eight block containing sixty-four pixels, where a 2-D DCT is computed for each block. A two dimensional block transform may be computed directly or by taking the one dimensional transform of each row, then, using the resulting transform coefficients, taking the one dimensional transform of each column. The exact method of performing the transform is left to the system designer.

A one dimensional DCT appropriate for JPEG is defined by Rau and Yip [16] as

$$X(m) = \left(\frac{2}{N}\right)^{1/2} k_m \sum_{n=0}^{N-1} x(n) \cos\left[\frac{(2n+1)m\pi}{2N}\right] \quad (2.4a)$$

and its inverse is defined as

$$x(m) = \left(\frac{2}{N}\right)^{1/2} k_m \sum_{n=0}^{N-1} X(n) \cos\left[\frac{(2n+1)m\pi}{2N}\right] \quad (2.4b)$$

where

$$\begin{aligned} x(m) &= \text{1 - D sample value,} \\ X(m) &= \text{1 - D DCT coefficient,} \\ k_m &= 1/\sqrt{2} \text{ for } m = 0, \\ k_m &= 1 \text{ otherwise..} \end{aligned}$$

The JPEG standard defines the forward 2-D DCT as

$$S(v, u) = \frac{C(v)}{2} \frac{C(u)}{2} \sum_{y=0}^7 \sum_{x=0}^7 s(y, x) \cos[(2x+1)u\pi / 16] \cos[(2y+1)v\pi / 16] \quad (2.5a)$$

and the inverse 2-D DCT as

$$s(y, x) = \frac{C(v)}{2} \frac{C(u)}{2} \sum_{y=0}^7 \sum_{x=0}^7 S(v, u) \cos[(2x+1)u\pi / 16] \cos[(2y+1)v\pi / 16] \quad (2.5b)$$

where

$$\begin{aligned} C(u) &= 1/\sqrt{2} \text{ for } u = 0, \\ C(u) &= 1 \text{ for } u > 0, \\ C(v) &= 1/\sqrt{2} \text{ for } v = 0, \\ C(v) &= 1 \text{ for } v > 0, \\ s(y, x) &= \text{2-D sample value,} \\ S(v, u) &= \text{2-D DCT coefficient.} \end{aligned}$$

The 2-D DCT describes an image's pixel values as a weighted linear combinations of the transformation's basis functions (which represent spatial frequencies). Using the inverse transformation, the pixels can be reconstructed. To achieve dramatic image compression, the JPEG baseline system employs a coarse, non-linear, amplitude scaling of spatial frequency coefficients (referred to by JPEG as quantization). This quantization exploits the facts that natural images tend to have a concentration of lower spatial frequencies, and, that to human observers, the loss of an image's higher spatial frequencies does not cause a subjectively perceptible degradation in an image's quality.

3.3 Quantization of the Discrete Cosine Transformation Coefficients

Quantization of the DCT coefficients is the second major element necessary to produce image compression. It is very important since it removes many transform coefficients -- especially those for the higher spatial frequencies. Once the 2-D DCT coefficients have been calculated they are arranged in terms of increasing spatial frequencies, resulting in the zigzag arrangement, shown in Figure 7. The zigzag ordering increases the probability that the remaining non-zero, coefficients will have lower index numbers within the 8x8 coefficient matrix. Since a complete representation of the quantized coefficients can be made as soon as

the last non-zero term is identified, fewer coefficients will be needed to represent the zigzag ordered transform coefficients. The encoding procedure exploits this property by placing an end of block (EOB) symbol when the last non-zero, diagonally indexed coefficient is found..

Quantization values are selected for each spatial frequency (basis function coefficient) at an amplitude that is just detectable by the human eye. The basis function coefficients are then divided by this value and the result rounded to an integer value. The inverse process, dequantization, consists of multiplying the scaled-down coefficients by the quantization value to reconstruct the DCT coefficient. Although the reconstructed value is not equal to the original value, the difference between the quantized and unquantized values should not be readily detected by the human eye.

The JPEG standard leaves the selection of quantization tables to the system designer. However, it contains a set of quantization tables which have been determined to provide good results for a large variety of images. Since the human eye is not as sensitive to spatial variations in the two chrominance coordinates, separate quantization tables are provided for the luminance and chrominance components of color images. In light of the low frequency nature

0	1	5	6	14	15	27	28
2	4	7	13	16	26	29	42
3	8	12	17	25	30	41	43
9	11	18	24	31	40	44	53
10	19	23	32	39	45	52	54
20	22	33	38	46	51	55	60
21	34	37	47	50	56	59	61
35	36	48	49	57	58	62	63

Figure 7. Zigzag ordering of DCT coefficients

of the chrominance components, JPEG also permits sub-sampling of the chrominance components (i.e., sampling at a half or quarter the luminance component rate). Once the DCT coefficients have been quantized they represent a lossy version of the original image. The final step in compressing the image is to entropy encode the DCT quantized coefficients.

3.4 Entropy Coding

The JPEG standard defines two entropy encoding methods for the quantized coefficients - Huffman coding and arithmetic coding. Both coding methods are specified for all JPEG modes of operation. Huffman and arithmetic codes may be pre-defined by assuming the statistics of the coefficients or computed specifically for each image after an initial statistics-gathering pass of the data is performed. The system presented in this work uses a Huffman code with predefined code assignments thus avoiding the delay of a statistics gathering pass, at a slight loss of coding efficiency. This course of action may be exploited by the modem's built-in data compression scheme. In addition, since luminance and chrominance image statistics are very different, they are represented by different Huffman codes.

For the system presented in this work, each 8x8 block of image data represents a minimum coded unit (MCU). In monochrome images, only the luminance component is used; therefore, it alone represents an MCU. For color images, an MCU is composed of an interleaved pattern - luminance (Y) component followed by the first chrominance component (I) then the remaining chrominance component (Q). This interleaved pattern (Y,I,Q) represents an 8x8 image block and is repeated for the entire image. Throughout the image, blocks are ordered from left to right across the image, then from top to bottom, down the image.

Although the system presented here samples each color component at the same rate, JPEG allows for reduced sampling rates in the chrominance color components - by factors of 2 or 4. In system's which have a different number of samples for each color component, the MCU's interleaved components correspond to the same portion of the image (an 8x8 block at the lowest sampling rate). For example, suppose that I is sampled at 1/2 the rate of Y, and Q is sampled at 1/4 the rate of Y. In this case, each MCU will represent the same section of the

image, the 8x8 block representing Q, with the following repeated interleaved pattern: Y,Y,Y,Y,I,I, and Q.

Recall that as a result of the quantization process, a matrix representing a block of quantized coefficients will have mostly zero coefficients, and that proceeding zigzag through the quantized coefficient matrix, the greater the likelihood of a value being zero. Figure 8 shows quantized Luminance DCT coefficients used to illustrate the encoding procedure.

As stated earlier, the luminance and chrominance coefficients use different Huffman codes. In addition, for each color component's 8x8 block, the encoding process encompasses two statistical models to encode the quantized coefficients - one for the first coefficient, and a second for the remaining coefficients. The first coefficient, which represents the dc spatial frequency, (DC) is differentially encoded, using the previous DC value for that color component. Using the assignments shown in Table 2, the DC difference value is represented by a DC SIZE symbol. The symbol equals the number of additional bits needed to represent its encoded additional bits. A DC Huffman code is assigned to the DC SIZE symbol. The additional bits are sent unchanged. As can be seen in Table 2, the additional bits equal the DC difference value itself when it is greater than zero, otherwise they represents its ones complement.

13	-10	1	0	0	0	0	0
-2	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Figure 8. Sample Compression Values of Quantized Luminance DCT Coefficients.

Table 2. Huffman coding of DC Coefficients

DC Difference Value	DC SIZE Symbol	Encoded Additional bits
0	0	n/a
-1, 1	1	0, 1
-3,-2, 2,3	2	00,01, 10,11
-7,...,-4, 4,...,7	3	000,...,011, 100,...,111
-15,...,-8, 8,...,15	4	0000,...,0111, 1000,...,1111
-31,...,-16, 16,...,31	5	00000,...,01111, 10000,...,11111
-2047,...,-1048, 1024,...,2047	10	

Using the sample values in Figure 8, and assuming a previous DC value of 0, the DC term would be represented by its SIZE (4) and the additional bits representing its difference value (13). The SIZE symbol will be represented by a Huffman code assignment (101) while the additional bits are represented by the value (1101). The first coefficient is, therefore, represented by the bit stream: 101 1101.

The AC terms are next encoded in zigzag sequence. The AC coefficients are represented by a RUNLENGTH/SIZE symbol pair followed by additional bits which represent the non-zero coefficient's value. The RUNLENGTH is determined by the number of previous zero coefficients. The AC SIZE symbol, like the DC SIZE symbol is the number of additional bits needed to represent the non-zero coefficient's value. The combination RUNLENGTH/SIZE symbol is assigned a single Huffman code. RUNLENGTHS are limited to runs of 15 zeros. Sixteen consecutive zeros are assigned RUNLENGTH/SIZE (15,0). However, after the last non-zero coefficient in a block is encountered, the end of block (EOB) symbol (0,0) is appended to the data stream. The assignments for the AC values are shown in Table 3. Again, like the DC case, the additional bits equal the AC value itself when it is greater than zero, otherwise they represents its ones complement.

Considering the sample data in Figure 8, the first AC term in zigzag sequence has RUNLENGTH zero and value, -10. Using Table 3 the first AC coefficient is represented by the combined RUNLENGTH/SIZE symbol (0,4) and the additional bits representing (-10). The symbols pair (0,4) is assigned a Huffman code and the additional bits are encoded as their ones complement. Continuing this process, the RUNLENGTH/SIZE symbol and value representation of the sample data in Figure 8, is (0,4)(-10) (0,-2)(2) (2,1)(1) (2,1)(1) (EOB). Using the JPEG luminance Huffman code symbol assignments, found in Appendix I, the following bit assignments are made:

(0,4)	1000
(0,2)	10
(2,1)	11100
(2,1)	11100
(EOB)	1100.

Using the additional bit assignments from Table 3, encoded AC data is: 1000 0101 10 01 11100 1 11100 1 1100. The entire code representing this 8x8 luminance is therefore: 101 1101 1000 0101 10 01 11100 1 11100 1 1100; a total of 35 bits.

Table 3. Huffman Coding of AC Coefficients

AC Value	AC SIZE Symbol	Additional bits values
-1,1	1	0,1
-3,-2,2,3	2	00,01,10,11
-7,...,-4,4,...,7	3	000,...,011,100,...,111
-15,...,-8, 8,...,15	4	0000,...,0111,1000,...,1111
-31,...,-16, 16,...,31	5	00000,...,01111,10000,...,11111
-4095,...,-2048, 2048,...,4095	12	

Huffman codes are variable length and uniquely determinable. No code word is the prefix for any other code word. Each code word begins with a one. The JPEG standard contains Huffman codes with symbol assignments which have been generated based on the average statistics of many different images. This system's design utilizes these Huffman codes which can be found in Appendix I.

One measure of an image's size is its total number of bits per pixel (bpp). In this example, 35 bits are needed to represent 64 pixels for a ratio of 0.55 bpp. In the original image, 8 bits were used to represent each pixel. Therefore, this image block has achieved a compression ratio of approximately 15:1.

3.5 Compressed Image Data Format

In order to ensure compatibility between JPEG encoded images and to obtain the correct image decompression, the JPEG data is stored in a well defined, byte oriented, format. The format defines parameters, markers, and entropy-coded data segments. Each marker is four bytes; the first two bytes being the hexadecimal digits FF. Markers are used to: identify the start of frame, scan and data segments, and transmit quantization tables, Huffman tables, and restart markers. The specific format and hierarchical syntax for these segments can be found in the JPEG specification [15]. JPEG allows an abbreviated format for table specifications provided by means other than the compressed image data.

The system presented in this work is primarily used to quickly transmit captured video images. The real-time video is also being recorded and will be available a short time later. Transmitting images using a fully compliant JPEG image format adds bytes of data and places extra complexity in the decoder structure. It would have to sort through and discard much unnecessary data. Therefore, a modified JPEG data format is used for this image transmission system.

The appropriate JPEG Start of Image (SOI) marker is used, but the hierarchical frame and scan fields are replaced by an application specific marker. The application specific marker (ASM) identifies the data that provides the necessary image attributes information. The

entropy coded data follows the application specific data field. Restart markers (RST) are optional. The modified JPEG format is shown in Figure 9.

Converting the system's compressed image to a fully compliant JPEG format is a trivial exercise. The appropriate header can easily be added, once the compressed image is saved to a file. In this modified format the abbreviated JPEG format is used to define the table specifications; they are known *a priori*.

The RST marker serves no purpose in the released, executable version of this program, and has been removed from this code. It was used during debugging to check if the number of decoded MCUs corresponded to the number of transmitted MCUs. It could, with more than a little effort, be used to skip to a new image start point and recover from transmission errors.

Next, Chapter 4 describes the software algorithm used to implement the JPEG baseline system and modified data format. The software is written in C using the Borland C++ Version 4.0 compiler [3]. Chapter 4 discusses, in detail, the DCT algorithm, image quantization, and bit manipulation routines necessary to implement variable length encoding using a byte oriented machine.

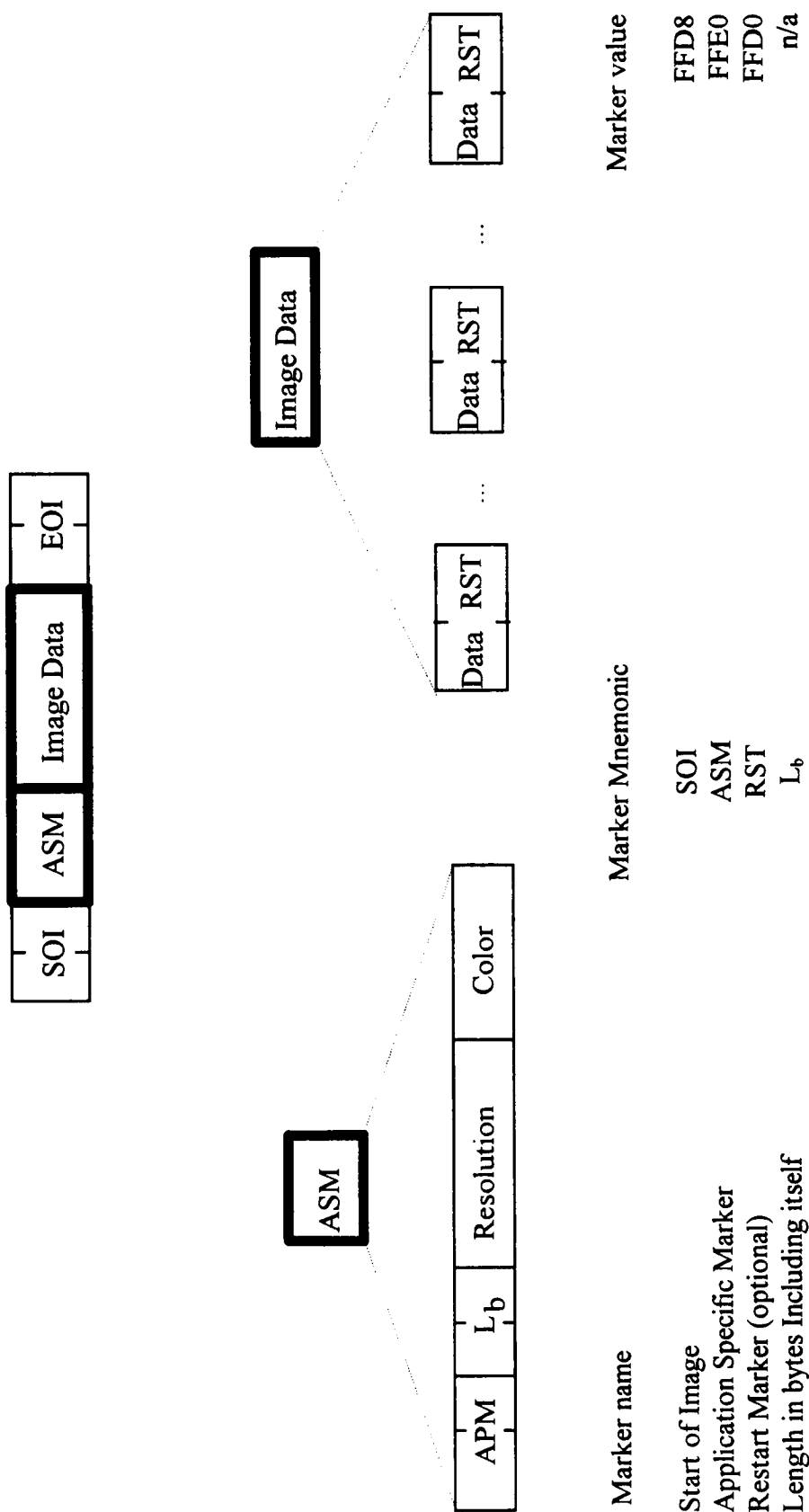


Figure 9. Application Specific Data Field. This field represents the data segment which provided image resolution and color information.

4.0 THE IMAGE COMPRESSION ALGORITHM

4.1 Software Considerations and Image Formatting.

This chapter discusses the choice of programming language, identifies necessary compiling information, and highlights the development of the image compression routine. The topics touched on are the stored image format, the selection and implementation of a DCT algorithm, image quantization, and encoding the quantized coefficients. The source code for the image compression algorithm is found in Appendix II.

This system's software is written entirely in the C programming language, using Borland C+ Version 4.0. The C programming language was selected since it has the ability to perform bit manipulations. Implementing variable length encoding in a byte oriented computer requires a significant amount of bit manipulation. The algorithms were implemented with execution speed as the primary goal. In light of this, most of the code is written in-line, avoiding more eloquent functional calls and object oriented programming. The compiler options were set to Borland's default choices for maximum execution speed.

In programs running under the DOS operating system, memory models determine the addressing methods for data and program segments thereby effecting execution speed. The less address bytes needed to point to a memory location, the quicker the program's execution. To maximize execution speed a compact memory model was chosen. This memory model limits program code size to 64K bytes and individual arrays to 64K bytes. Arrays, which are stored in the data segment, may take all available memory up to DOS's 640K byte limit. A description of the C programming language and compiler options can be found in [3][17].

The flow of the compression algorithm is (1) prepare header information; (2) read a segment of the picture (R, G, B color components) and store in memory (as Y, I, Q color components); (3) assemble an 8x8 block for each MCU color component; (4) perform the discrete cosine transformation; (5) quantization of the transform coefficients; (6) encode the transformed coefficients and transmit each completed word (two bytes); (7) repeat for the remaining MCUs and segments; and (8) append the end of image marker segment. The

compressed image data is written to a buffer to await transmission. Data transmission is interrupt driven allowing data output to be simultaneous with image compression.

To place a high resolution image, which has over 900K bytes, into computer memory, it must be broken into smaller pieces. Table 4 defines the image resolutions and the size of their reduced segments. For a high resolution color image, an image segment corresponds to four rows of eighty MCUs, where each MCU consists of one 8 x 8 block for each color component.

4.2 Discrete Cosine Transforms and Quantization Algorithms

To make computation of the DCT feasible for this application, an efficient computational algorithm is needed. Initial attempts to increase the computation speed of the DCT exploited the relationship between it and the discrete Fourier transform to utilize the so called fast Fourier transform (FFT) algorithms. These early attempts have been improved upon by fast computational algorithms that have been obtained through factorization of the DCT matrix. When the component factors are sparse, the decomposition represents a fast algorithm. A 2-D DCT algorithm may be computed directly or by taking a 1-D DCT first by each row then by each column. A thorough study of the DCT is found in [16].

Table 4. Image Resolution, File, And Segment Attributes.

Resolution Descriptor	Image Dimensions (pixels)	Image Size (bytes)	Segment Size (bytes)	Number of Segments	MCUs per segment
High	640 x 480	921600	61440	15	320
Medium	320 x 240	230400	38400	6	200
Low	160 x 120	57600	57600	N/A	300

‘There are many ways to measure complexity and efficiency of an implementation or algorithm, and a final assessment depends on both the available technology and intended application.’ [14] For a digital computer the number of multiplication and additions is directly related to computational speed when algorithms are implemented.

The selection of a DCT algorithm for this application was based on two factors: adequate speed and ease of programming. Ease of programming leans heavily towards a 1-D DCT algorithm. In addition, for a 1-D DCT, the inverse transform needs only be computed for rows which have non-zero coefficients. Very often only one or two rows have non-zero coefficients. Table 5 summarizes the number and type of computations required for implementing various DCT algorithms. Multiplication by the normalization constant is not considered in the values shown in Table 5. This omission is justified since, in this implementation, the normalization coefficients are included with the quantization coefficients to eliminate the calculation.

Evaluation of the data reveals the direct computation of the forward 2-D DCT needs fewer multiplications, thus executes fastest. However, for the inverse operation, a 1-D algorithm, in many cases requires fewer multiplications and, may often execute faster. Referring to Figure 8, a transform coefficient on the fifth through eighth rows (i.e., index 10, 19, or greater) must be present before the direct 2-D DCT computation is advantageous.

Table 5. Number Of Multiplications Needed For Various 2-D DCT Algorithms.

	2-D DCT Algorithms		1-D Algorithms Acting on Rows and Columns	
	by FFT	[5]	[9]	[22]
	$(3/8)N^2 \log_2(N^2)$	$N^2 \log_2(N^2) - N^2$	$(N^2/2) \log_2(N^2)$	$N^2 \log_2(N^2) - (3/2)N^2 + 4$
Forward	320	144	192	292
Inverse	320	144	108 - 192	184 - 292

The overall speed of this system, for a high resolution image, is dictated by the data transmission rate; the design specifications are based on it. Execution speed is, therefore, most critical once the data becomes available for restoring the compressed image. Since the proposed JPEG quantization tables (see Appendix I) favor removing high frequency image content, and because most images are predominantly, especially in their chrominance components, low frequency, the 1-D DCT algorithm developed by Lee [9] was chosen for implementation. The flow graphs for this algorithm are shown in Figures 10 and 11.

In calculating the DCT, the normalization constant $(2/N)^{1/2}$ multiplies all resulting coefficients while $k_m = 1/\sqrt{2}$ multiplies only the first coefficient (see equation 2.4).

Since the DCT is first applied to each row, then to each column, each of the 64 coefficients are multiplied twice by the normalization constant. In addition, each element in the first row and first column is multiplied once by $1/\sqrt{2}$, except the first term which is multiplied twice. Figure 12 shows the resulting forward DCT *normalization* matrix (N).

Quantization is main cause for lossy image compression, but its implementation is trivial. Once the 2-D DCT is completed, the resulting coefficients are divided by their corresponding quantization table values. In order to eliminate divisions, and reduce the number of multiplications needed to compute the DCT, the DCT *normalization* matrix (N) is multiplied by the reciprocal of the quantization table (Q) and the result (X) multiplies the transform coefficients. This calculation is illustrated in equation 4.1.

$$X_{ij} = N_{ij} * \frac{1}{Q_{ij}} \quad \text{for } i, j = 1, \dots, 7. \quad (4.1)$$

4.3 Entropy Coding Algorithm

Once the coefficients have been quantized, they represent the image less its *subjective redundancy*. The objective of entropy coding is to minimize the number of bits needed to transmit these coefficients. The process uses differential encoding to assign code symbols to the DC value and a combination of zero run length and size to assign code symbols to the AC coefficients. Separate Huffman codes are assigned to the AC and DC symbols.

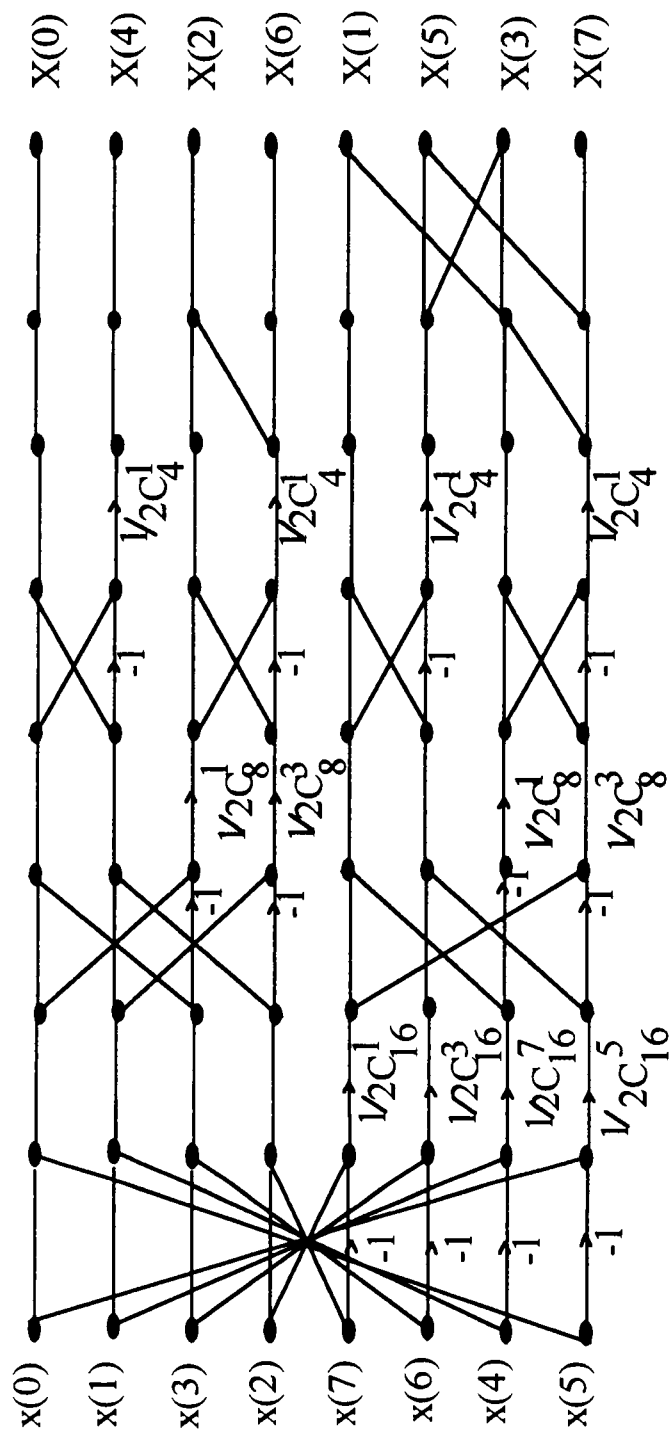


Figure 10. Forward DCT Algorithm Implementation. Normalization constants are included with the appropriate quantization table coefficients. The Symbol C_y^x refers to $\cos(x\pi/y)$.

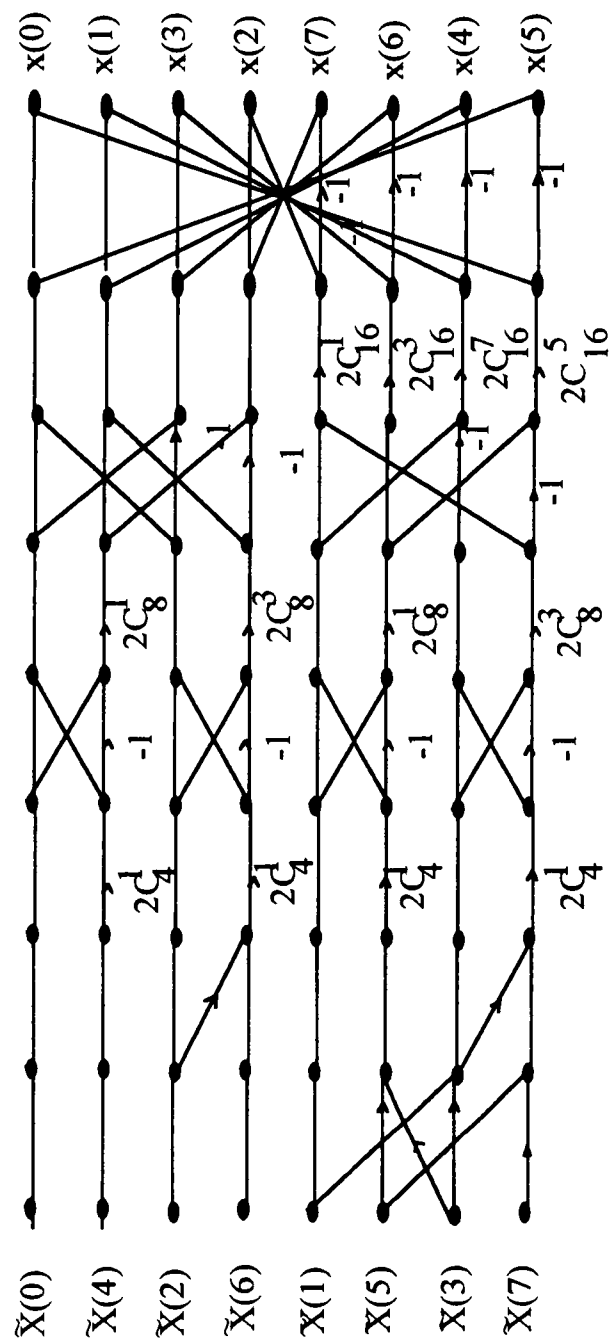


Figure 11. Inverse DCT Algorithm Implementation. Normalization constants are included with the appropriate quantization table coefficients. The Symbol C_r^z refers to $\cos(z\pi/r)$.

$1/8$	$\sqrt{2}/8$	$\sqrt{2}/8$	$\sqrt{2}/8$	$\sqrt{2}/8$	$\sqrt{2}/8$	$\sqrt{2}/8$	$\sqrt{2}/8$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$
$\sqrt{2}/8$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$	$1/4$

Figure 12. DCT *Normalization* Matrix (N)

First, the previous DC term is subtracted from the current DC term, and the result is stored. Its DC SIZE symbol is then calculated and stored. Next, the remaining non-zero AC values are stored, in zigzag order, for later use. The zero runs, which are the next coefficients RUNLENGTH symbol, are stored in the upper four bits of a character array. The $\log_2(|\text{non-zero coefficient}|)$, which is the corresponding SIZE symbol, is stored as the lower four bits of the same character. Each character then forms a pointer to the proper Huffman code entry. The DC SIZE value, likewise, can be used to point to the proper DC Huffman code. Figure 13 illustrates the relationship between the AC codes symbols, SIZE and RUNLENGTH of the non-zero values,. Figures 13 and 14, respectively, show the flow chart and source code to create the pointer.

RUNLENGTH	SIZE	Code Symbol	Comment
0	0	(0,0)	End of Block (EOB)
0	1	(0,1)	
1	A	(0,A)	
2	1	(2,1)	
2	A	(2,A)	
F	0	(15,0)	Sixteen zeros
F	1	(15,1)	
F	A	(15,1)	

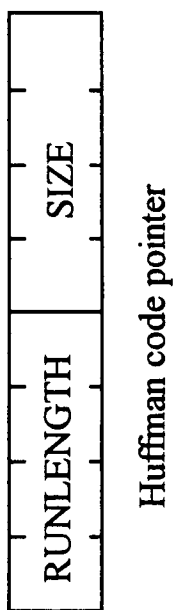


Figure 13. Illustration of Huffman Code Pointer. The Huffman code pointer is constructed from RUNLENGTH/SIZE symbol pair. The code symbols (x,0), and (x,B) through (x,F) are not used since $0 < \text{SIZE} < 11$.

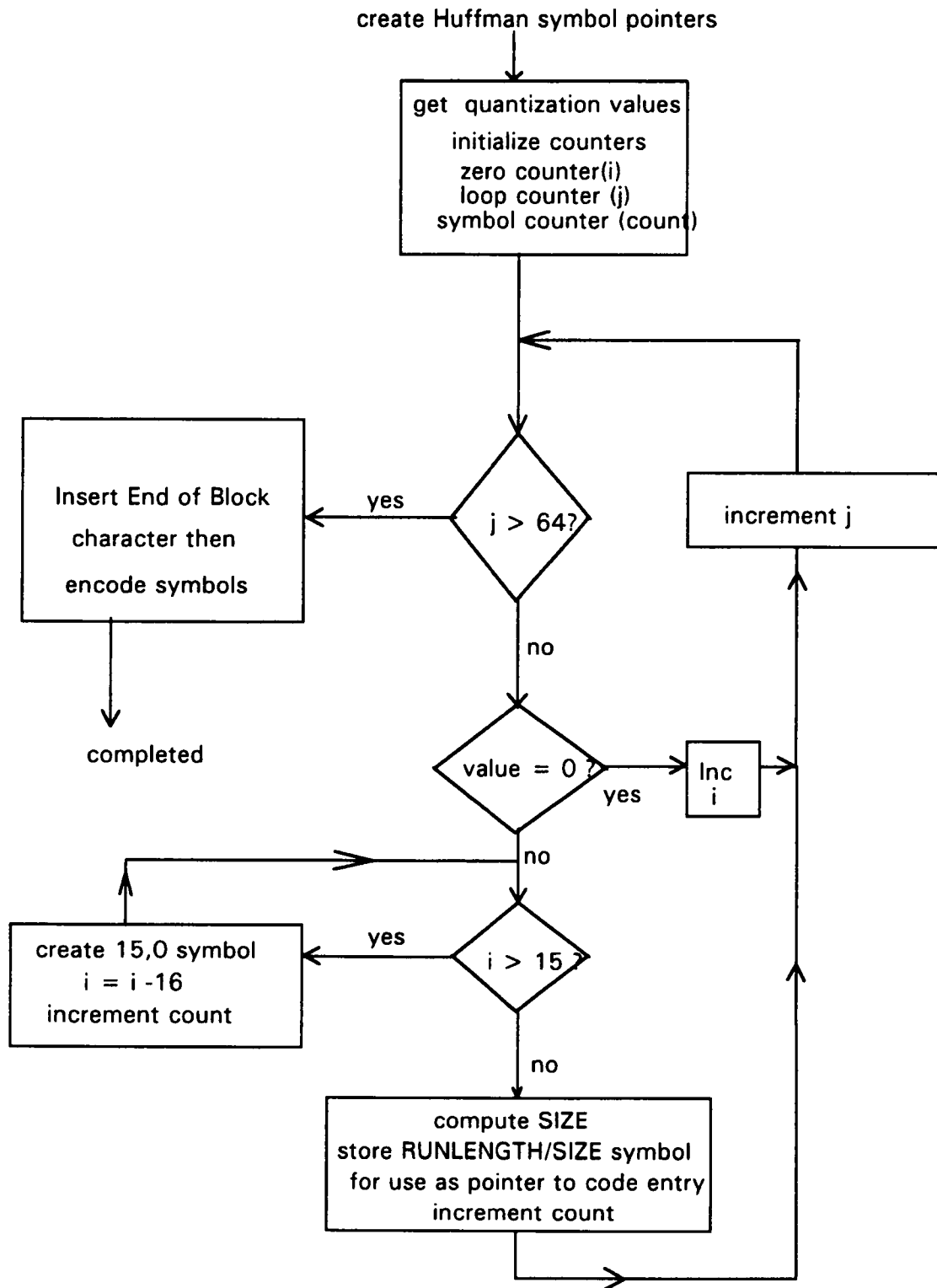


Figure 14. Huffman Code Pointer Flow Chart

<pre> i = 0; for (j=1;j<64;j++) { if (pre_entropy[diag_index[j]] == 0) i++; else { while (i > 15) { run_length[count] = 240; value[count] = 0; count++; i = 16; } i = i << 4; value[count] = pre_entropy[diag_index[j]]; temp1 = 0; temp = abs(pre_entropy[diag_index[j]]); while (temp > 0) { temp >>= 1; temp1++; } run_length[count] = i temp1; count++; i = 0; } } run_length[count] = 0; value[count] = 0; count++; </pre>	<pre> // Count = 0; // RUNLENGTH counter // loop counter // If value = 0 increment RUNLENGTH // If current value > 0 then // Is RUNLENGTH greater than 15? // yes, store as run_length of 15 zeros // with a value of zero // point to next code entry // reduce RUNLENGTH by 16 zeros // repeat until RUNLENGTH < 16. // move RUNLENGTH to upper 4 bits // store number in 'value' // determine SIZE = log₂(value) // counter for SIZE // temp = absolute value of 'value' // divide temp by 2 // increment SIZE // when temp = 0 then temp1 = SIZE // upper bits = RUNLENGTH, lower bits = SIZE // increment array counter // Get next entry // end of loop, last entry checked // append end of block character // run_length = value = 0 [EOB] // counts number of entries in value </pre>
---	---

Figure 15. Source Code For Creating Huffman Code Pointers

As stated previously, the encoded coefficients are represented by their Huffman symbols followed by additional bits representing their value. Once all the code pointers are calculated, the DC Huffman code and its additional bits are output. The additional bits equal the ones complement, the coefficient's value for a negative number, or its absolute value, for a positive number. The AC coefficients are then similarly, encoded and output.

Variable length code words cannot be placed into the output data queue directly. They must first be stuffed into multiple bytes as computers are byte oriented machines. Since the code lengths range up to sixteen bits, they are stuffed into two-byte words prior to transmission. When shifting bits into a partially filled word, three cases may occur -- either their are too few, too many, or exactly the right number of empty bits. This bit stuffing procedure

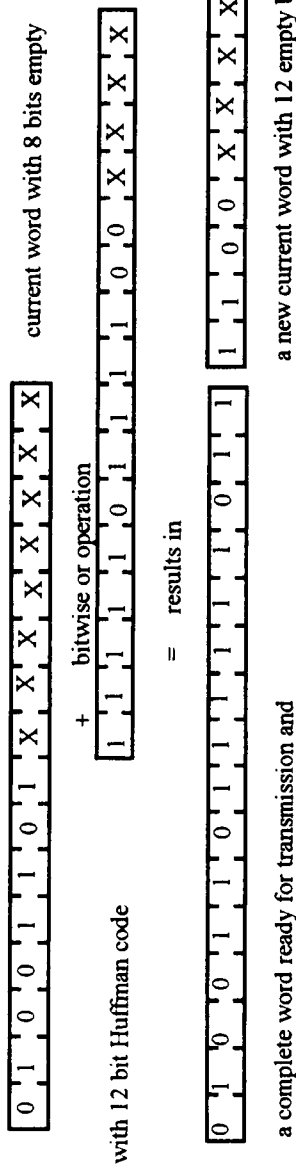
involves tracking the number of bits available and the number of bits being pushed into the word. The bits shifting routine is illustrated in Figure 16, its flow chart shown in Figure 17, and source code in Figure 18.

In a color image the luminance components are processed first, followed by the two chrominance components. After each MCU component is processed, its counter is incremented. Once the number of MCUs corresponding to the image segment size has been reached, the next segment is processed, and so on until the image is completed. The final step is to append end of image marker segment (EOI) to the output stream. Before the EOI is output, any unfilled word is stuffed with ones then queued to the output data stream. Lastly the system returns to terminal mode, ready to transmit another image.

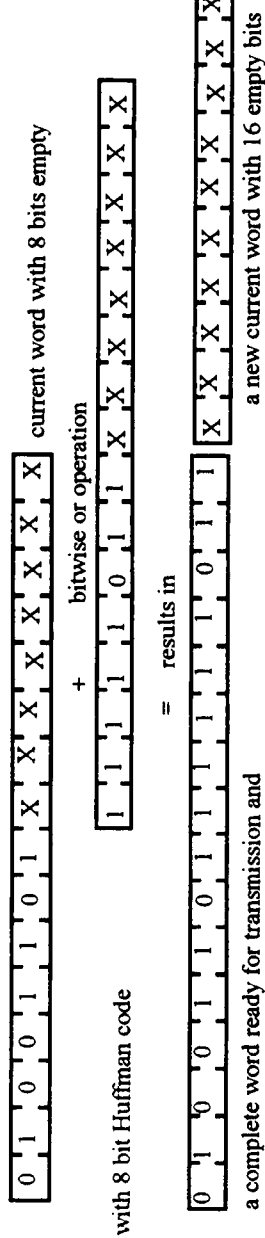
Both AC and DC Huffman codes are pre-assigned for luminance and chrominance color components and stored in arrays in memory. The luminance and chrominance quantization tables are also pre-determined and stored in arrays. Quantization coefficients are scaled by the normalization matrix prior to their use. For a color image, each MCU is composed of one component of Y, I, and Q. In monochrome images an MCU has only the luminance component.

Next, Chapter 5.0 discusses the implementation of the reconstruction of the image from the compressed image data. It highlights variable length code identification and interpretation, bit manipulation and shifting routines, inverse quantization, inverse transformation, and reconstruction of the image's pixel values.

CASE: bits empty < variable length code



CASE: bits empty = variable length code



CASE: bits empty > variable length code

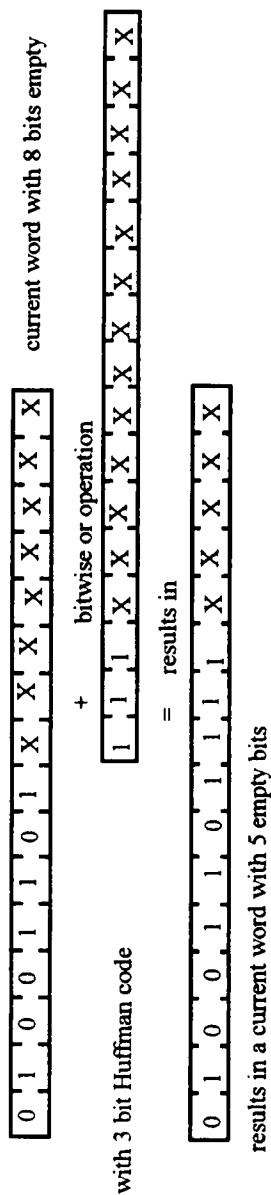


Figure 16. Illustration of Bit Stuffing Routine

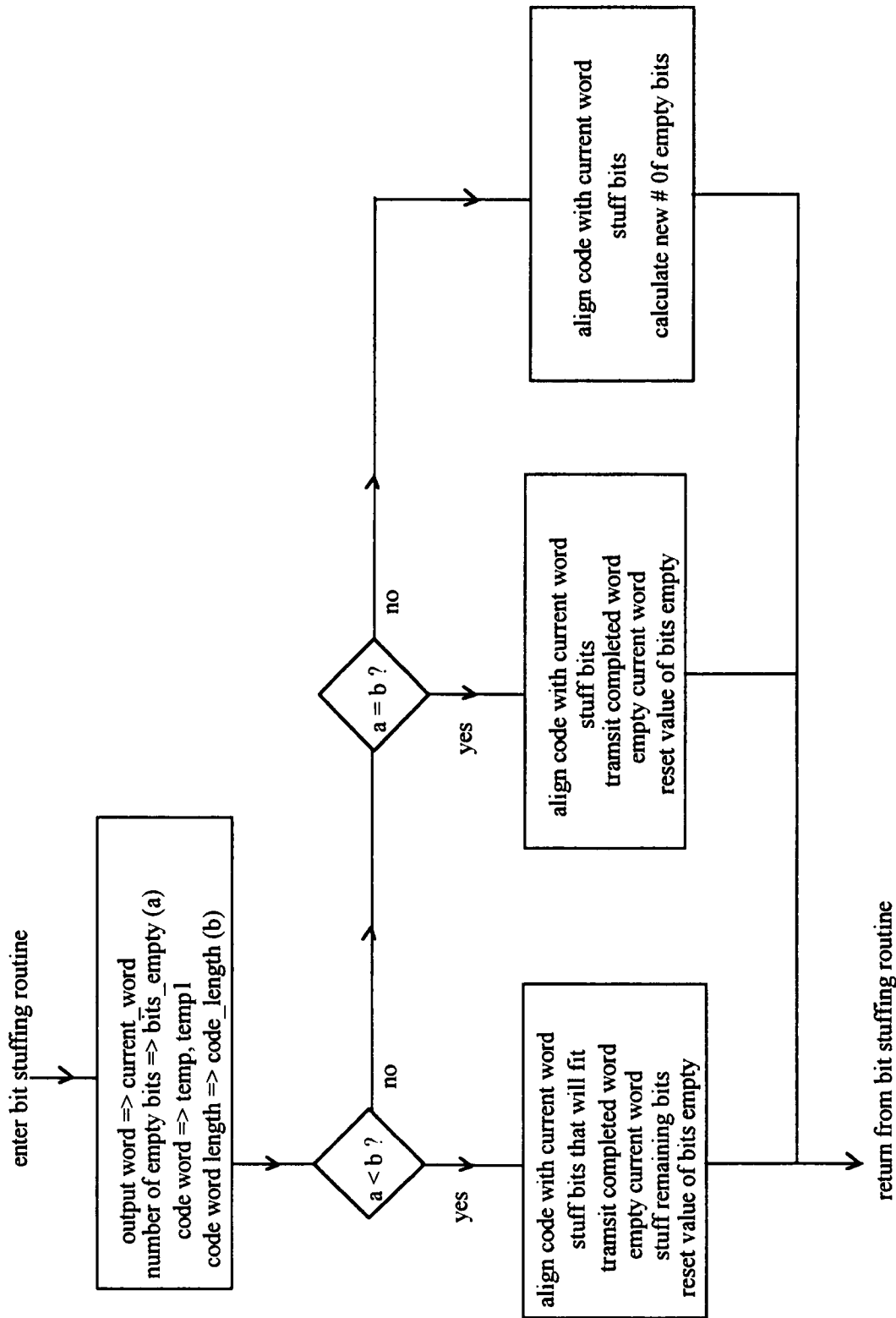


Figure 17. Flow Chart for Bit Stuffing Routine.

```

if (bits_empty < code_length){
    temp1 = temp;
    temp = temp >> bits_filled;
    current_word |= temp;
    current_word = SWITCH(current_word);
    Write(current_word);
    if ((current_word << 8) == 0xff00)
        Write(0x00);
    current_word = SWITCH(current_word);
    Write(current_word);
    if ((current_word << 8) == 0xff00)
        Write(0x00);
    current_word = (temp1 << bits_empty);
    bits_filled = code_length - bits_empty;
    bits_empty = 16 - bits_filled;
}
else {
    if (bits_empty > code_length){
        current_word |= (temp >> bits_filled);
        bits_filled += code_length;
        bits_empty -= code_length;
    }
    else {
        current_word |= (temp >> bits_filled);
        current_word = SWITCH(current_word);
        Write(current_word);
        if ((current_word << 8) == 0xff00) Write(0x00);
        current_word = SWITCH(current_word);
        Write(current_word);
        if ((current_word << 8) == 0xff00) Write(0x00);
        bits_empty = 16;
        bits_filled = 0;
        current_word = 0;
    }
}
}

// temp holds code word
// code_length = word length code.
// Enough space in current word?
// No. Create second code word
// align bits
// add bits to current word
// exchange first & second bytes
// output first byte
// shift out first byte
// if byte equals marker flag, pad
// reverse byte order
// output next byte
// current word now shifted to zero
// if byte equals marker flag, pad
// place remaining bits in new word
// calculate bits empty
// calculate bits filled

// Sufficient space in current word
// Too much space?
// Yes, then align and add value
// calculate bits filled
// and bits empty

// exact amount of space
// Yes, then align and add value
// reverse byte order
// output first byte
// shift & pad, if necessary
// reverse byte order
// output second byte
// shift & pad, if necessary
// set bits empty
// set bits filled
// current_word now empty

```

Figure 18. Bit Stuffing Routine. An unsigned integer, called `current_word`, holds the encoded data. Integers are stored lower order byte first. `SWITCH` reverses byte order for output.

5.0 THE IMAGE DECOMPRESSION ALGORITHM

5.1 General

This chapter highlights the reconstruction of the compressed image data received across the data link and stored in the input queue. Items presented are: variable length code identification and interpretation; bit manipulation and bit shifting routines; inverse quantization, transformation, and image color restoration.

The reconstruction algorithm, `expand.cpp`, searches the incoming data for the start of image marker (SOI) and data segment marker. The data segment identifies whether the image is color and specifies resolution. This information is needed to initialize the various parameters necessary to decode and display the image. Before the markers are decoded, reception can be aborted simply by striking a key. However, once the data stream marker is recognized, the reconstruction begins and the decoding process cannot be aborted.

Next, the received data is decoded, and the quantized coefficients are restored and transformed to recreate the luminance (Y) and chrominance (I and Q) values. Because the coefficients were quantized, the reconstructed values only approximate those of the original image. Data is decoded, one MCU at a time. For monochrome images, an MCU contains only the Y component; color images contain the I and Q components as well. The decoding process uses pointers to retrieve the appropriate decoded values. These pointers differ for each color component. Each MCU color component (Y, I, and Q) is restored in turn. Once all color components for the MCU are recovered, each pixel in the MCU is transformed into its R, G, and B color representation and displayed.

5.2 Identifying and Decoding the Variable Length Encoded Data Stream

At the start of the data decoding algorithm, one 16 bit word (*current_word*) holds the first two data bytes; a second 16-bit word (*word_in*) holds the next two data bytes. The decoding process begins by recognizing the DC code which resides in the highest order bits of *current_word*. The variable length DC codes are ordered according their length, which

corresponds to their relative frequency of appearance. A brute force code identification method is used to determine the DC code word.

This process begins by masking the lower order bits in *current_word* to reveal the smallest possible potential code entry. This value (stored in *temp*) is then checked against each code word of the same length until a match is found. If no match is found, the size of the mask is decreased and the process is repeated until a match is found. An average of 4 searches is needed to identify each code word. However, since the DC values are ordered in their relative frequency of appearance, varying only slightly between adjacent blocks, and the DC value is differentially encoded, the brute force technique often works on the first or second attempt. Therefore, it is an adequate and efficient method of identifying the DC code word.

If all DC code entries have been checked, and a match is not found, the decoding process continues assuming a zero DC value. This condition, which results from a communications error, coding error, or transmission termination by the sender, is identified when an end of block (EOB) symbol is not identified for any color component. The image restoration process then aborts.

The DC code does not itself represent the DC coefficient. It provides the SIZE symbol (*add_bits_length*), the number of additional bits appended to the code word, which represents the differentially encoded, quantized DC coefficient. Figure 19 shows the source code for identifying the code word and SIZE symbol. Its use is illustrated in Figure 20.

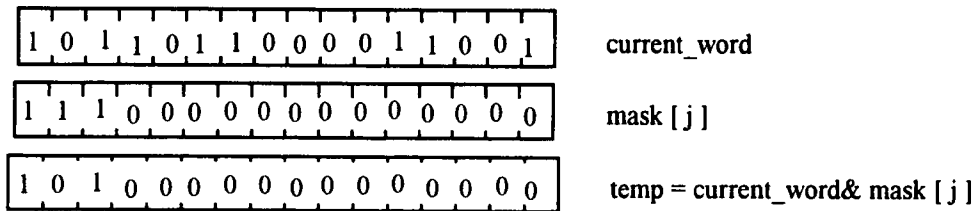
```

j = 1;                                     // j is an index to the correct code size
                                           // code length = j + 1
while (j<10) {                             // search in length order
    temp = current_word & un_mask[j];      // unmask correct number of data bits
    for (k = DC_index[j-1]; k < DC_index[j]; k++){ // k points to proper length code
        if (temp == DC_code[k]){           // If it matches.
            add_bits_length = k;            // set associated additional bits length
            bits_shifted = j + 1;           // and set number of bits to be discarded
            j = 12;                         // set flag to exit while loop
            break;                          // break the for loop
        }                                  // If not a match, continue.
    }                                     // check next code of same length
    j++;                                  // increment code length pointer
                                           // check next length code
}

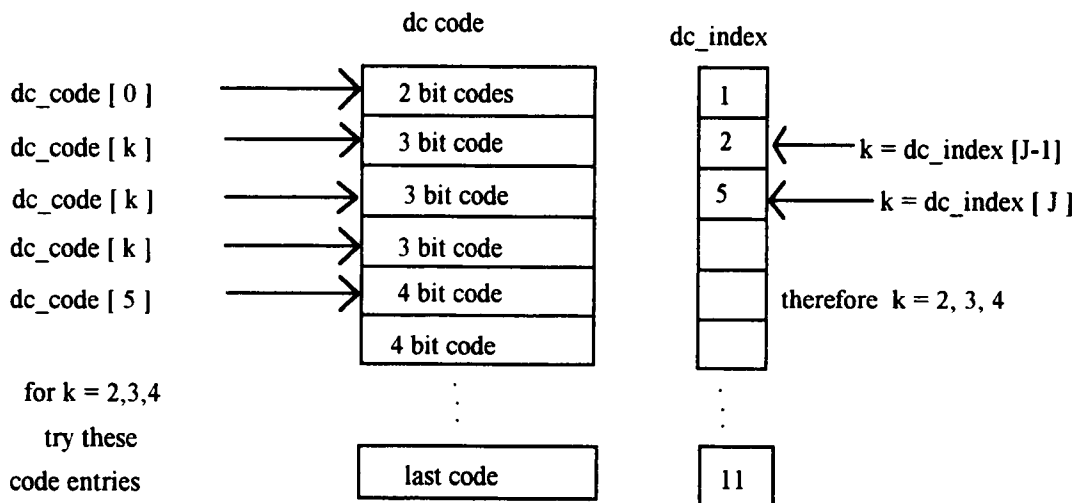
```

Figure 19. Variable Length DC Code Identification Source Code

Assume $J = 3$



temp is compared with dc_code[k]



If no match is found, increase j and look again

Once a match is found $k = \text{SIZE}$,
the length of the additional bits representing the
differentially encoded dc quantization coefficient.

Figure 20. Illustration of Variable Length DC Code Identification Routine.

Once *add_bits_length* is found, the bits in *current_word* are left shifted the length of the identified DC code to remove the decoded bits. Next, *current_word* is replenished by stealing bits from *word_in*. If, in this process, *word_in* is emptied, a new *word_in* is retrieved from the data stream. There are three cases which must be explored when replenishing the discarded bits: when more than enough bits are available, when less than enough bits are available, and when the exact number of bits are available. Figure 21 provides the source code for this routine and Figure 22 illustrates its operation.

If, when replenishing *word_in*, an end of image marker (EOI) is identified, no more data is available and none is retrieved. Fortunately, none is needed. This marker flag indicates either the end of the image or that the image transmission has been terminated.

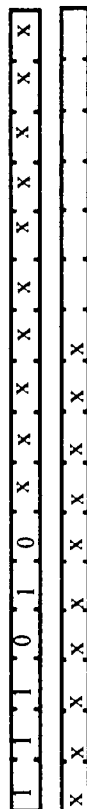
```

current_word <<= bits_shifted;           // shift out code from current word
temp = word_in;                          // retrieve bits in next word to fill gap
if (bits_shifted < bits_available) {     // more than enough bits in next word
    temp >>= 16 - bits_shifted;          // yes, align words
    current_word |= temp;                // refill current word
    word_in <<= bits_shifted;            // shift bits out next word
    bits_available -= bits_shifted;      // reduce number of bits available in
}                                         // the next word
else {                                    // else,
    if (bits_shifted > bits_available) { // too few bits?
        temp >>= 16 - bits_shifted;     // yes, align bits
        current_word |= temp;           // add bits to word
        if (marker_flag == 0)           // get next word, if not end flag
            marker_flag = Word_in(word_in);
        temp = word_in;                 // copy next word to temp
        bits_shifted -= bits_available; // set number of bits remaining
        bits_available = 16;            // and bits available
        temp >>= 16 - bits_shifted;     // align bits
        current_word |= temp;           // refill current word
        word_in <<= bits_shifted;       // shift bits out next word
        bits_available -= bits_shifted; // reduce number of bits available
    }                                   // the in next word
    else {                              // else, must be the exact number of bits
        temp >>= 16 - bits_shifted;     // align words
        current_word |= temp;           // add bits
        if (marker_flag == 0)           // get next word, if not end flag
            marker_flag = Word_in(word_in);
        bits_available = 16;            // reset bits available for new word
    }
}

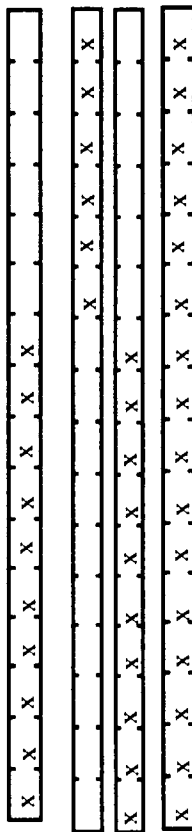
```

Figure 21. Source Code for Bits Replacement Routine

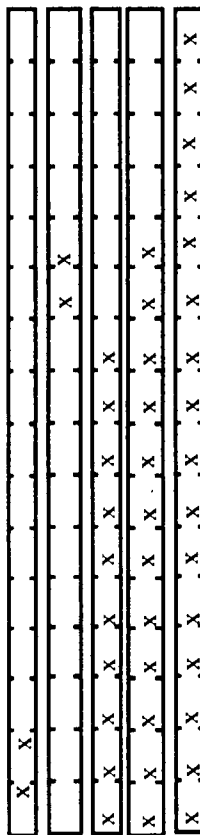
bits_shifted = code_length



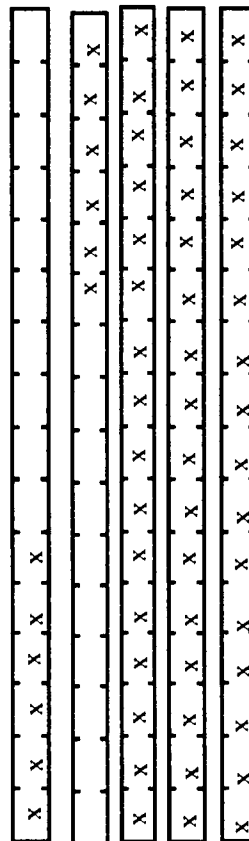
CASE 1: bits_available > bits shifted



CASE 2: bits_available < bits shifted



CASE 3: bits_available = bits shifted



old current_word
current_word << bits_shifted

word_in with sufficient bits_available

word_in >> bits_shifted
current_word << bits_shifted
new current_word

word_in with insufficient bits_available
word_in >> bits_shifted
current_word << bits_shifted
new current_word
get new word_in and complete like CASE 1

word_in with exactly enough bits_available
word_in >> bits_shifted
current_word << bits_shifted
new current_word
get next word_in

Figure 22. Illustration of Bit Shifting and Replenishing Routine.

The additional bits appended to the DC code represent the differentially encoded, quantized DC coefficient. *Value* is the array that holds the quantized transform coefficients. The bits representing the DC coefficient are moved to *value[0]*. Zero is a valid number of additional bits. It corresponds to the number zero. If *value[0]* does not equal zero, but is less than $2^{(add_bits_length - 1)}$ it represents the ones complement of the differentially encoded coefficient; otherwise, it represents the differentially encoded coefficient exactly. *Value* is compared against $2^{(add_bits_length - 1)}$ and if needed, it is complemented to recover the appropriate value. Once the differentially encoded DC term is determined, it is added to the previous DC value to recover the quantized coefficient. The decoded bits are then, as before, discarded and replenished from *word_in*.

The recovery of the quantized AC coefficients is a little more challenging for two reasons. There are a substantial number of code entries, and each code represents both a RUNLENGTH (number of zeros between non zero values) and SIZE symbol. The number of AC code entries makes a brute force identification method undesirable. Indeed, it would require a large amount of valuable time. Therefore, a more sophisticated search, which takes advantage of the code structure, must be implemented.

The Huffman codes used in this application have the unique properties that: (1) no code word is the prefix for another code word; (2) all but the first two code words have leading '1' bits which correspond closely to the length of the code; (3) the lion's share of the codes are 16-bits long with more than 8 leading '1' bits; and (4) all 16-bit code words increase in value by one from the previous code word. Since the code words are stored in a look up table in increasing value, each of these properties can be exploited to minimize the number of comparisons which must be made.

By checking the number of leading '1' bits, one can begin a search by pointing to the shortest code word having this number of leading '1' bits. Using this technique, the 16-bit code words require 1 try, the remaining code words need an average of 3.4 comparisons for the luminance coefficients, and 4.3 comparisons for the chrominance coefficients. Since the longer search paths are less likely, actual performance should surpass this average.

A 16-bit code word can be determined without additional searches because (1) the code pointer is now at the first 16-bit entry, and (2) each 16-bit code is one greater than the previous code word. These facts are exploited by subtracting the first 16-bit code entry from the unknown 16-bit code word and adding the result to the code pointer. The code pointer now points to the correct code entry.

Once the correct code word has been identified, the code word pointer can be used to retrieve the RUNLENGTH and SIZE symbol needed to determine the next non-zero quantized coefficient. Figure 23 illustrates its operation while Figure 24 provides the source code for this search technique.

Just as in the DC case, the code word bits are discarded and new bits are shifted to fill the void. Now, as before, the additional bits representing the quantized coefficient's are recovered and complemented when necessary. This number is stored in array *value* in its order of appearance. In the AC case, the code word pointer also retrieves the associated zero run length and stores it in an array called *run_length*, at the same position as the non-zero, quantized coefficient.

The *run_length* and *value* arrays are used to reconstruct a diagonally ordered 8 x 8 block matrix called *array*. Each *run_length* is associated with the DC coefficient. Using the associated *run_length*, the quantized coefficients are placed in their proper location in *array*. The correct placement of the coefficient is accomplished by placing the DC coefficient in *array[0]*, then advancing diagonally, *run_length[i]* places, and putting *value[i]* in this location. When *run_length[i]* and *value[i]* equal zero, the symbol represents an end of a block (EOB).

The process of decoding the AC coefficients continues until (1) an EOB symbol is found, (2) more than 65 code entries have been identified, or (3) image transmission has been terminated. The number 65 accounts for there being 64 entries and an EOB symbol. If case (2) or (3) image reconstruction is aborted and an error message is displayed.

The process of decoding the AC coefficients continues until (1) an EOB symbol is found, (2) more than 65 code entries have been identified, or (3) image transmission has been terminated. The number 65 accounts for their being sixty-four entries and an EOB symbol. If case (2) or (3) image reconstruction is aborted and an error message is displayed

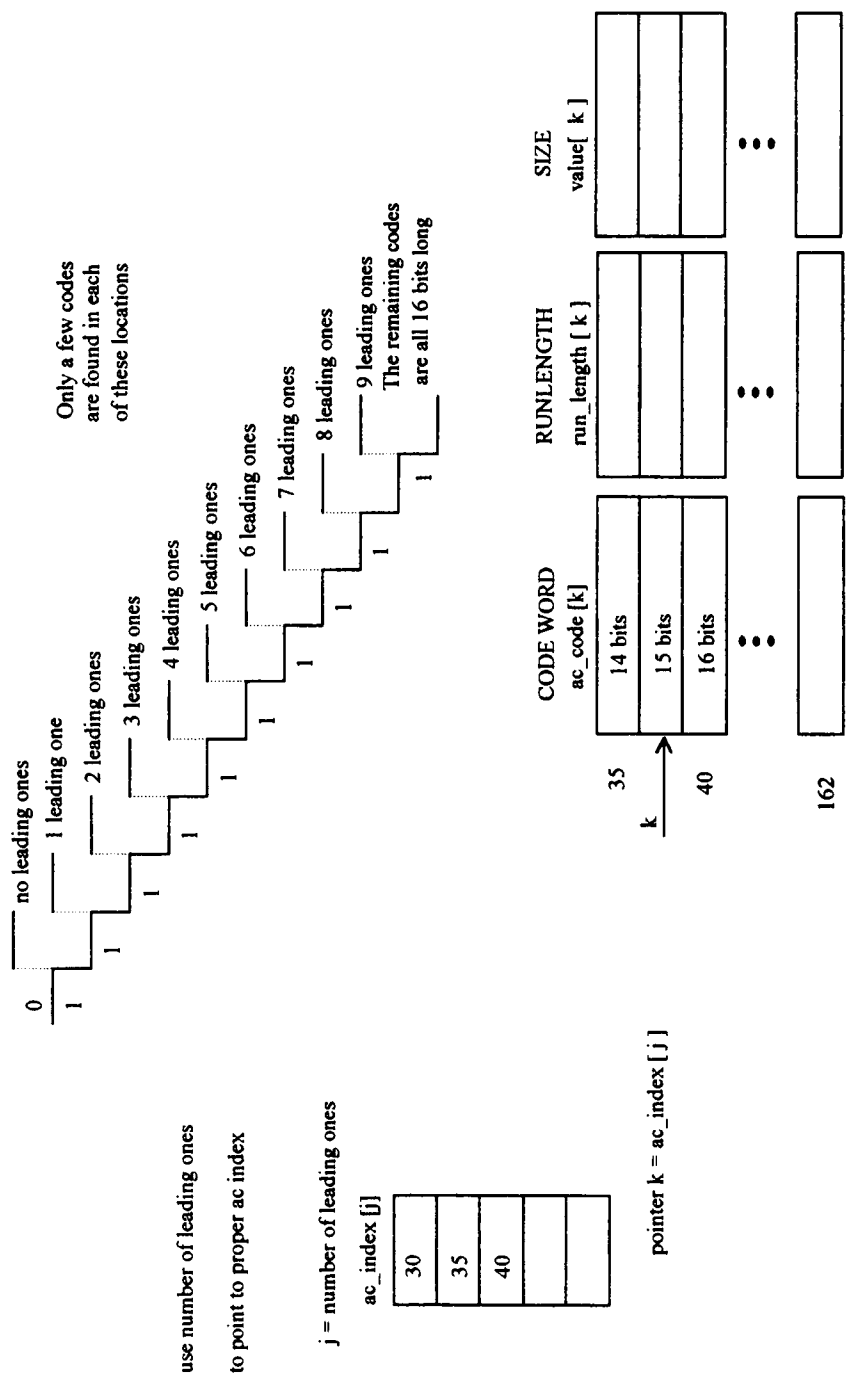


Figure 23. Illustration of Decoding AC Code Entries. When pointer, k, equals 40, all remaining code entries as well as the unknown code word, are 16-bits long. Furthermore, the difference between the unknown code and AC_code[k] equals the difference between the unknown code and its location. Therefore, by setting k equal to k = k + (unknown code - AC_code [k]), it will point to the correct code. This pointer (k) can now be used as an index to the desired RUNLENGTH and SIZE symbols.

```

j = 1;                                // j holds number of leading '1's plus 1
temp = current_word;                  // temp holds variable length code
while ((temp & 0x8000) == 0x8000){    // while current bit is '1'
    temp <<= 1;                        // shift bit out
    j++;    // increment code length pointer
}                                     // end of while loop
while (j<16) {                        // while code length is valid
    temp = current_word & un_mask[j]; // unmask first code
                                        // with correct number of leading '1s'
    for (k = AC_index[j-1]; k < AC_index[j]; k++){ // check all codes of this length
        if (j == 15)                  // if code length is
            k += (temp - AC_code[k]); // 16 point to proper entry
        if (temp == AC_code[k]){      // code match?
            run_length[i] = run_index[k]; // yes, set run_length
            add_bits_length = value_index[k]; // set number of additional bits
            bits_shifted = j + 1;       // set number of bits to throw away
            j = 16;                     // flag to end while loop
            break;                      // break to exit for loop
        }                             // or check next entry
    }                                 // increment k
    j++;    // increment code length
}                                     // return to top of while loop
                                        // to check the next code_length
                                        // with the same number of leading '1s',

```

Figure 24. Source Code for AC Code Word Search

Once the quantized coefficients are placed in their proper locations, they are multiplied by the quantization/normalized matrix, which is identical to the one used during compression. The largest non-zero row in the diagonally ordered matrix is determined. The number of rows needing to be transformed and a pointer to the coefficient array are passed to IDCT, the inverse discrete cosine transform function.

The inverse transformation is the last step in restoring the compressed image data into a lossy version of the original image. If a monochrome image is to be transmitted only its Y component is present and equal intensity R, G, B colors are used to display the image. If a color picture is being transmitted, all three color components must be recovered and transformed to generate the correct R, G, B colors components. The image is displayed by passing the R, G, and B pixel values, and their display location (horizontal and vertical screen coordinates) to the appropriate *put_pixel_xxx* subroutine. Each *put_pixel_xxx* subroutine services a specific VGA adapter and video mode (xxx).

In this program, each recovered 8x8 block of data is displayed from left to right, for 8 pixels (three bytes per pixel), then the row is incremented and the column pointer is reset. Once all 64 pixels are displayed, the block pointer is advanced one 8-pixel wide column. When the rightmost last column of 8 rows is displayed, the column is reset to its leftmost value and the row pointer is advanced 8 rows. Figure 25 shows the mapping of each pixel to a display location. Documented source code for module `expand.cpp` is in Appendix II.

Chapter 6 presents the setup and operation of the telephone data modem. For the system presented in this work, the modem is configured as an interrupt driven input and output device using hardware flow control. In this configuration, the modem interrupts the CPU when received data is available and the computer's processor (CPU) notifies the modem when it can accept data. Both transmitted and received data is buffered to maintain the continuous operation of the modem and computer which operate at different speeds.

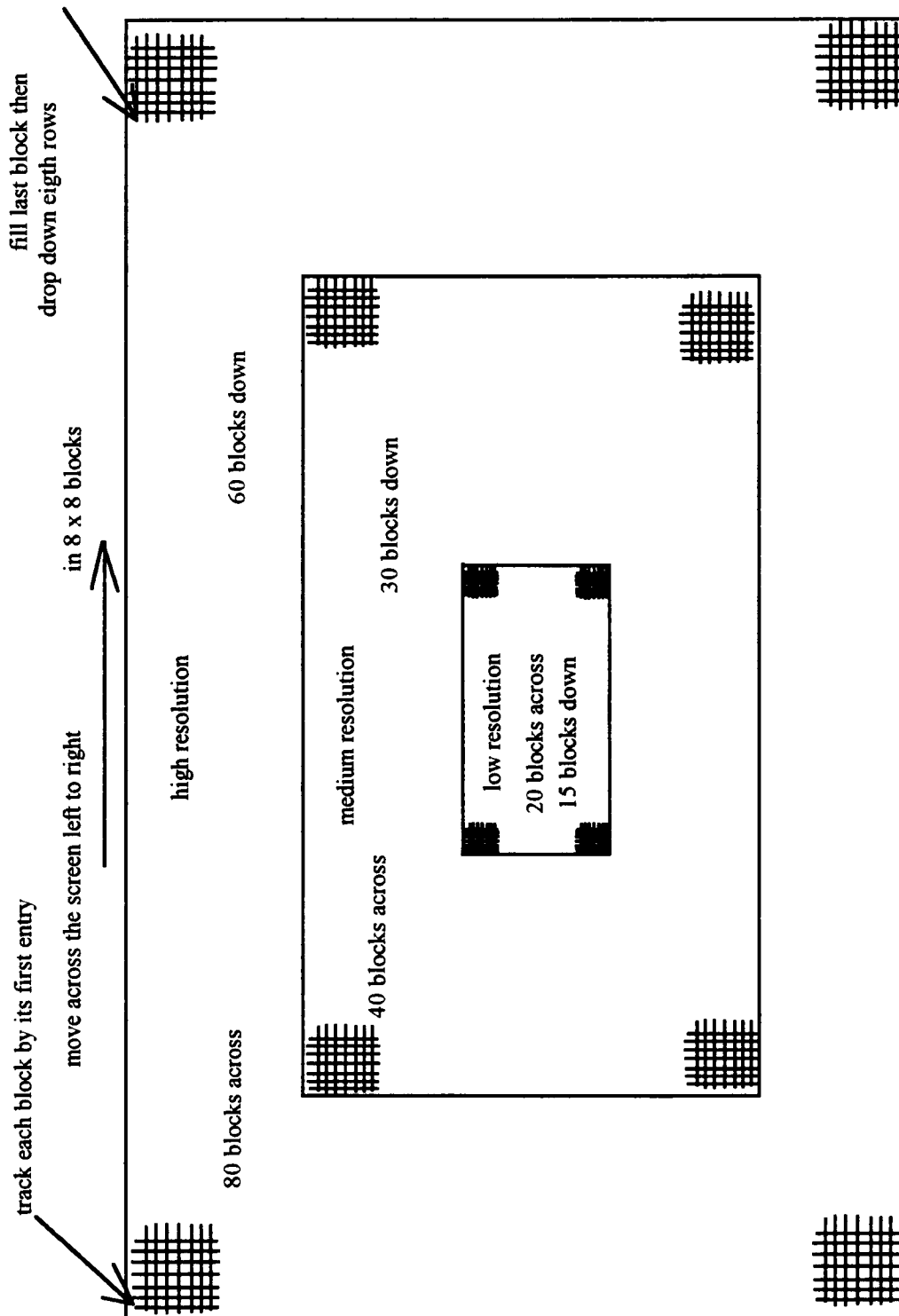


Figure 25 Image Display Characteristics.

6.0 THE DATA LINK

6.1 General

This chapter describes the setup and operation of the telephone data link. The modem operates as an interrupt driven input and output device using hardware flow control. When using hardware flow control, the modem interrupts the computer's processor (CPU) when received data is available and the CPU notifies the modem when it can no longer accept data. Both transmitted and received data is buffered to maintain the continuous operation of both the modem and CPU which operate at different speeds.

To meet the system's design specification of sending two high resolution, color images within one minute, a high speed V.32bis modem must be employed. Most off-the-shelf V.32bis telephone modems also have, as standard features, V.42bis error correction and data compression. While data compression and error correction are not specifically required to meet the performance specifications, they can improve data throughput and reliability when operating in a noisy environment. Although the system presented in this work uses a V.32bis and V.42bis modem, it will support all V series compatible modems. An excellent explanation of the initialization and operation of the modem using C++ object oriented programming can be found in Holmes and Flanders [7].

6.2 Initializing the Serial Data Port and Modem

Establishing a data link begins by initializing the data modem. Data modems fall into two categories: external and internal. External data modems interface to the CPU through a serial data port (comport), while an internally installed modem has an integrated comport. Whether internal or external, the serial port must be assigned a specific address: 2e8, 3e8, 2f8, or 3f8. These addresses are referred to as Comport 1, 2 3 or 4, respectively. Since the modem communications are interrupt driven, the comport must also be assigned a hardware interrupt. Most personal computers (PC) have interrupts 3 through 7 available for use by peripheral devices. All hardware interrupts are connected to a programmable interrupt controller (PIC). When an interrupt occurs, the PIC notifies the CPU which interrupt has occurred. The CPU

then executes a command residing at a designated memory location, the interrupt value times 8, which causes the CPU to execute the interrupt service function.

During system initialization the user is queried for the modem's comport, interrupt, desired serial port rate, and asynchronous data format. Default settings of comport 2 using interrupt 3, and a data rate of 38,400 bps are used to initialize the serial port. The default asynchronous data format is 8 data bits, 1 stop bit, and no parity bit. Next, using the provided comport address, the modem is identified, then configured to the requested settings. Lastly, the interrupt service function is installed and the assigned interrupt is activated. The program preserves the prior interrupt's service function location and resets it upon program termination.

In a V.42bis modem, to achieve a data throughput higher than the data transmission rate, the comport rate should be set higher than modem's highest data communications rate. The increased data throughput results from the modem performing data compression prior to data transmission. The data compression performed by the transmitting modem is inverted by the receiving modem. A V.42bis modem has internal transmit and receive data buffers, which store data prior to its compression and after its decompression.

The default comport rate (38,400 bps) is well above the highest V.32bis data transmission rate of 14,400 bps. Since the data being sent is already substantially compressed, the increased data throughput resulting from the V.42 textual substitution compression will not be great. However, increased data throughput will occur when the compressed image contains large areas of low detail information; the compressed data will then have some repetitive byte patterns.

The comport interfaces to the computer's CPU through a Universal Asynchronous Receiver Transmitter (UART). Two UARTs are commonly used, the 8250 (standard UART) and the 16550 (FIFO UART). The standard UART accepts only one character at a time, while the FIFO UART can store up to 14, first in first out (FIFO), characters at once. A complete description of the operation of these UARTs can be found in [7]. Today's internal V.42 modems do not contain a UART, but their internal processors emulate a FIFO UART.

A UART has internal registers that control or provide status information about some aspect of the data transmission process. The FIFO UART has eight addresses including the

comport address, which is referred to as the base address. Each addresses corresponds to at least one internal register. Some addresses correspond to different registers when writing to or reading from the location. The FIFO UART addresses one more write register than a standard UART which has seven. The additional register, the scratch pad register (SCR), can be used to distinguish these UARTs. Some of the registers in the FIFO UART also have expanded features, compared to the standard UART. These features handle considerations resulting from the receipt of multiple characters at a time. The FIFO UART's registers, their functions, and addresses are illustrated in Figure 26.

Once the user provides the comport address, interrupt value, data rate, and asynchronous data format, the modem can be activated and initialized. This process begins by determining which UART is present at the comport. To determine the UART present, the function `CheckTypePort`, shown in Figure 27, is called. It writes a known byte (01010101) to the SCR. The SCR is then read and if this value is not returned, a standard UART is assumed. Otherwise, the interrupt identification register (IIR) is checked for a proper FIFO response (c0), in which case a FIFO UART is declared.

6.2.1 Setting the Comport Rate.

Next, the comport rate is set using the `SetSpeed` function shown in Figure 27. To set the comport rate, the UART's divisor latch is enabled and its receive clock speed (115,200) is divided by a 16-bit number loaded into the UART's divisor. The divisor is set to 3 to achieve a data rate of 38,400 bps. Prior to loading the divisor, the divisor latch must be enabled. The divisor latch is enabled by setting bit 7 in the line control register (LCR) to 1. This action permits the UART to interpret the contents of the transmit holding register (THR) and the IER as the lower and higher order bits of the divisor, respectively. Once the data rate is set the divisor latch is turned off, thereby allowing access to the THR and IIR.

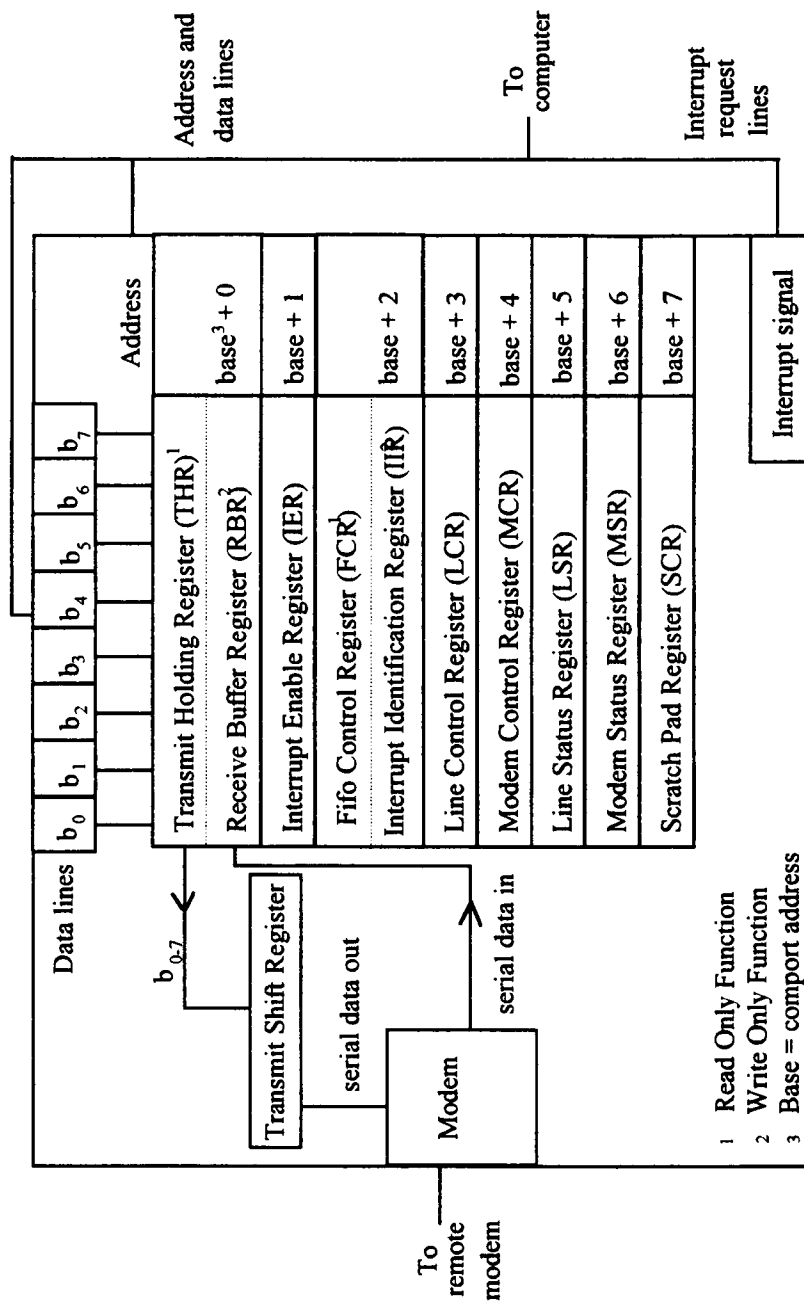


Figure 26. Illustration of an Internal Modem with a FIFO UART.

```

void CheckTypePort(void){

    outportb(base + SCR, 0x55);
    if (inportb(base + SCR) != 0x55)
        fifo = 0;
    else { outportb(base + FCR, 0xcf);
        if ((inportb(base + IIR) & 0xc0)
            != 0xc0) fifo = 0;
        else { // else must be FIFO UART
            outportb(base + FCR, 0);
            fifo = 1;
        }
    }
}

void SetSpeed(void) {

    outportb(base + LCR,
        inportb(base + LCR) | LCR_DLAB);
    outportb(base + IER, line_speed >> 8);
    outportb(base + THR, line_speed & 0xff);
    outportb(base + LCR,
        inportb(base + LCR) & ~LCR_DLAB);
}

```

// check for existence of scratch register
 // if not present
 // standard UART
 // set as 14 character FIFO,
 // if IIR show improper FIFO response
 // set as standard UART

 // clear FCR
 // set FIFO flag

// check for port types
 // line_speed holds divisor
 // set divisor latch
 // then output divisor
 // most significant bit
 // and least significant bit
 // reset divisor latch

Figure 27. Functions CheckTypePort and SetSpeed Source Code.

6.2.2 Setting the Serial Port Interrupt

After setting the comport rate, the function SetInterrupt, shown in Figure 28, is called. SetInterrupt first sets the data terminal ready (DTR) and request to send (RTS) signals, then instructs the modem that an asynchronous PC adapter is present. This is accomplished by sending the appropriate value, 13, to the modem control register (MCR). The CPU can now use the DTR setting to instruct the modem to hang up the telephone line (after a connection is made) by returning the DTR to zero. Also, after a connection is made, the RTS signal can now be used to implement hardware flow control. By resetting it to zero, the modem will stop

```

void SetInterrupts(void) {                                     // set interrupts

    outportb(base + MCR, MCR_DO);                             // set modem to acknowledge
                                                                // DTR, RTS and asynchronous
                                                                // adapter interface
    outportb(base + IER, 0);                                   // disable modem interrupt
    if (fifo) outportb(base + FCR, FCR_16550);                 // set fifo features
    inportb(base + MSR);                                       // clear modem status interrupt
    inportb(base + RBR);                                       // clear receive buffer
    inportb(base + IIR);                                       // show no interrupts
    outportb(base + IER,                                      // set modem interrupt
        IER_RBF | IER_TBE | IER_MSI);
}

```

Figure 28. Function SetInterrupts Source Code.

sending received data to the PC. Next, the modem's interrupts are disabled by writing a zero to the IER. The modem status register (MSR), IIR, and receive buffer register (RBR) are then read, to clear any previously initiated, and still pending, interrupts. Interrupts are then enabled for receive buffer full, transmit buffer empty and modem status interrupt conditions, by writing a 0b to the interrupt enable register (IER). These interrupts notify the processor, respectively, if data is available to be read, if the modem can accept more characters, or when a change to the clear to send (CTS) signal occurs. The CTS establishes the PC to modem data flow control; when CTS is not active, the PC stops sending data to the modem.

6.2.3 Setting the Serial Data Format

The last step in initializing the modem is to set the asynchronous data format. Setting the asynchronous data format is accomplished by setting the bits in the line control register (LCR) which represent the chosen data format. The byte representing the code word is prepared when the data format parameters are chosen. After all the parameters are chosen, this byte is output to the LCR. This code can be found in Setup.cpp in Appendix II.

6.3 Data Buffering

After the modem has been initialized, the FIFO input and output buffers are established. The operation of the input buffer is illustrated in Figure 29. The output buffer's operation parallels that of the input buffer. Figure 30 shows the source code which implements these buffers.

Data is removed from the input buffer using the character read (Read) function. When called, the read function returns -1 if no data is available. Otherwise it returns a zero. When a character is available in the input data buffer, it is retrieved using the buffer retrieval pointer, (*i_get*) which in turn is incremented. Next, the buffer counter (*count*) is decremented. When *i_get* reaches the bottom of the buffer, it is reset to the top of the buffer.

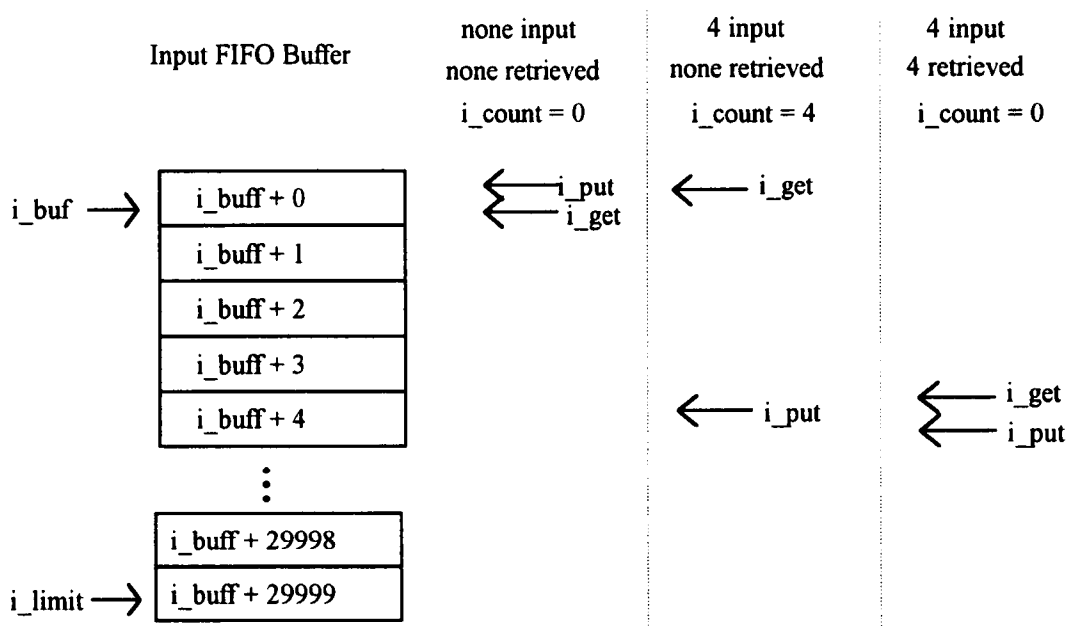


Figure 29. Input FIFO Buffer Operation. The buffer address (*i_buf*), buffer input pointer (*i_put*), and buffer output pointer (*i_get*) all point to the top of the buffer. As data is placed in the buffer, *i_put* and the buffer counter (*i_count*) are incremented. When data is retrieved from the buffer, *i_get* is incremented and *i_count* is decremented. When the buffer pointers reach the bottom of the buffer, they are reset to the top of the buffer. Flow control keeps *i_put* from looping past *i_get* by checking *i_count* to determine if the buffer is nearly full.

```

void SetBuffers(void) {
    o_size = 30000;           // set output buffer size
    i_size = 30000;           // set input buffer size
    i_buf = (unsigned char far *) farmalloc(30000L); // allocate input buffer
    if (i_buf == NULL) End(NoMemory); // end program if memory unavailable
    memset(i_buf, 0, i_size); // set input buffer to zero
    i_limit = i_buf + (i_size - 1); // set input buffer limit
    i_get = i_buf;             // set get from buffer pointer to first entry
    i_put = i_buf;             // set load buffer pointer to first entry
    o_buf = (unsigned char far *) farmalloc(30000L); // allocate output buffer
    if (o_buf == NULL) End(NoMemory); // end program if memory unavailable
    memset(o_buf, 0, o_size); // set output buffer limit
    o_limit = o_buf + (o_size - 1); // set output buffer limit
    o_get = o_buf;             // set get from buffer pointer to first entry
    o_put = o_buf;             // set load buffer pointer to first entry
    i_stop = i_size-100;       // initiate input flow control point
    i_start = i_stop/2;        // remove input flow control point
    o_stop = o_size-100;       // set output flow control point
    o_flow = i_flow = 0;       // flag when flow is controlled
    o_count = i_count = 0;     // initialized to top of data buffer
    empty_trans = 1;           // needed to prime output
    msr_changed = 1;           // needed to update display
}

```

Figure 30. SetBuffers Function Source Code and Operation. This function allocates memory for the input and output buffers and defines the pointers, counters and limits necessary for the buffer's operation. The basic operation of these buffers is illustrated in Figure 29.

Lastly, if input flow control was activated to restrict data input because the buffer was nearly full, and if the buffer has subsequently been sufficiently emptied, data input is resumed by setting the request to send (RTS) signal to one. When the modem receives this signal, it will again place data into the input data stream.

Data is placed into the output data buffer using the character and string write (Write) functions. When a character is ready to be output, the character write function is called. This function first determines if room is available in the output buffer. If not, program execution waits for room to become available. Next, the character is placed into the buffer, and the character counter is incremented. If necessary the empty transmitter flag is reset. Lastly, if a character is available, and the modem signals the path is clear to send (CTS), the character is output. The Read and Write functions are shown and explained in Figure 31.

```

unsigned int Read(char *c) {
    if (i_count == 0) return(0xffff);
    *c = *i_get++;
    i_count--;
    if (i_get == i_limit) i_get = i_buf;
    if (i_flow && (i_count <= i_start))
        RequestToSend(1);
    return(0);
}

//read character from input queue

// return queue empty flag (-1)
// else get character
// reduce queue count
// if bottom of buffer, loop to top
// if flow control no longer needed
// raise RTS, which resets flow control flag
// return character available flag (0)

void Write(int c) {
    if (o_count == o_stop) delay(1000);
    *o_put++ = c;
    o_count++;
    if (o_put == o_limit) o_put = o_buf;
    if (empty_trans) {
        empty_trans = 0;
        if ((inportb(base + MSR) & MSR_CTS)
            && (c = Deque()) != -1)
            outportb(base + THR, c);
    }
}

//write character to output queue

// wait for room in output queue
// put char in queue
// increase queue count
// if end of buffer, loop to beginning
// if transmitter empty
// reset flag
// if clear to send
// and character available
// output character

```

Figure 31. Functions Read and Write Source Code. The character string write function merely calls the Write function once for each character in the character string.

Functions Queue and Deque increment and decrement the pointers associated with the input and output data buffers, respectively. These functions are shown in Figure 32. In addition to handling the pointer arithmetic, Queue sets input (RTS) flow control.

6.4 Hardware Flow Control

This section describes data and flow control. When using an interrupt driven port, hardware flow control is the most suitable. The program code which implements the hardware flow control is integrated in the read, write, buffering, and interrupt functions and is not separable. Hardware flow control is a method a device uses to signal that a character may be received by the device. When a modem (comport) is ready to accept a character, it signals the CPU by setting the CTS signal active. If CTS is not active, and the system honors hardware flow control the CPU will not send a character to the modem, otherwise the character will be lost. When the CPU is ready to accept characters signals modem by setting the RTS signal active. In this situation, if the modem has an internal

```

void Queue(int c) {                                     // place character in input queue
    if ((i_count >= i_stop)&& NOT i_flow)                // flow control needed and not on
        RequestToSend(0);
    if ((i_size - i_count) != 0){                       // room in queue
        i_count++;                                     // increase character count
        *i_put++ = c;                                 // add character to queue
        if (i_put == i_limit) i_put = i_buf;           // if end of buffer, loop
    }
}

unsigned int Deque(void) {                               //get character from output queue
    unsigned char c;                                   // character to be removed
    if (o_count == 0) return(-1);                      // if empty return empty
    c = *o_get++;                                     // otherwise get char from queue
    o_count--;                                        // reduce queue count
    if (o_get == o_limit) o_get = o_buf;               // if end of buffer, loop
    return(c & 0xff);                                 // return char
}

```

Figure 32. Functions Queue and Deque Source Code.

buffer, future incoming characters will be stored. Otherwise, the characters will be lost, as no additional storage capacity is available. Loosing future incoming characters is preferable to the sending of characters to a filled buffer, as the buffer will overwrite itself and its entire contents will be corrupted.

6.5 The Modem Interrupt Service Function

This function is shown in Figure 33. When a modem interrupt is detected, control of the CPU is forfeited to the interrupt service function. This function first interrogates the modem to determine which interrupt is pending. Only the highest pending interrupt is reported at a time. The function first checks the reported interrupt, then checks those of lower priority. If the interrupt is due to a CTS signal change, the flag showing a change to the CTS state is set. This flag is used to update the modem status display. If output has been previously controlled, and the modem can now accept a character, output flow control is removed, and, if a character is available, it is sent to the modem. Next, the transmit buffer register empty interrupt is checked.

If the transmit register is empty one of three actions will occur. First the CTS signal is checked to determine if flow control is active. If clear to send is not active, flow control will be enabled and the next interrupt will be checked. If CTS is active, but no data is available to be sent, the empty transmitter flag is set and the next interrupt is checked. Lastly, if the path is clear and character(s) are available, they are loaded into the modem's transmit holding register (THR); up to 14 for a FIFO UART but only one, for a standard UART. The next interrupt, receiver buffer full (RBF) is now checked. If the receive buffer is full, the received characters are placed into the input buffer. The interrupt service function is then exited.

6.6 Establishing a Data Link Connection

After flow control is implemented, the data buffers are established, the modem is initialized, and the hardware interrupts activated, a communications link may be established with a remote device. The system present in this work functions as a terminal

```

void UartIntFunction(void) {                                     // modem interrupt function

    int    c, cnt, first_cycle;
    char   iir;
    while (((iir = inportb(base + IIR)) & IIR_PEND) == 0) {    // while interrupt is pending
        switch (iir & IIR_II) {                                  // switch on interrupt type
            case IIR_MSI:                                       // if CTS
                msr_changed = 1;                                // msr flag changed
                if ((int_msr = inportb(base + MSR))             // CTS interrupt?
                    & MSR_CTS)
                if (o_flow || inportb(base + LSR)               // and flow controlled?
                    & LSR_THRE) {
                    o_flow = 0;                                  // clear flow control flag
                    if ((c = Deque()) == -1) empty_trans = 1;    // if available
                    else outportb(base + THR, c);                // output character
                }
                if (int_msr & MSR_DDCD) {                         // if carrier lost 500 ms
                    delay(500);                                  // echo EOI command
                    if ((inportb(base + MSR) & MSR_CD) == 0) {
                        Write(0xff);
                        Write(0xd9);
                        StatusLine();
                    }
                }
            }

            case IIR_TBE:                                       // transmit buffer empty
                cnt = fifo ? 15 : 1;                             // FIFO accepts 15 characters
                if (NOT (inportb(base + MSR)                     // if not clear to send
                    & MSR_CTS)) o_flow = 1;                     // show flow controlled
                else for (; cnt--){                             // loop till buffer is full
                    if ((c = Deque()) == -1) {                   // if not char available
                        empty_trans = 1;                         // set empty trans flag
                        break;                                    // exit loop
                    }
                    outportb(base + THR, c);                     // otherwise output characters
                }

            case IIR_RBF:                                       // read data if available
                while ((inportb(base + LSR) & LSR_DR)
                    Queue(inportb(base + RBR));
                }
        }

        __asm mov    al, EOI                                    // load EOI to register al
        __asm out I8259, al                                     // send EOI to int controller
    }
}

```

Figure 33. Modem Interrupt Service Function Source Code.

program with interrupt controlled input and output. Therefore, to establish a phone connection the user must enter modem commands which perform these functions. (Modem commands are entered from the keyboard, written to the output buffer, then sent to the modem.) The modem commands are referred to as AT commands, meaning attention, as AT precedes each command. Although a sophisticated modem user can take advantage of many AT commands, only three common AT commands are needed to operate this system. These commands are A for answer, D for dial, and X1 to ignore the absence of a dial tone. Additional commands are available by striking the function keys shown as shown on the terminal screen's menu.

The terminal screen also shows the special function commands which compress and transmit an image (F3), receive and display a compressed image (F4), select the image resolution and color qualities (F6), clear the terminal screen (F5), recalls the setup utility (F1), hangs up a connection (F7), and terminate the program F(10). The system operation, terminal modes, image display and available function commands are explained in detail in Chapter 7, System Integration.

7.0 DISPLAYS, USER COMMANDS, AND SYSTEM INTEGRATION

7.1 The System Displays and Video Memory

The application described in this work uses three distinct displays: the setup display and the terminal display, the image display. The setup and terminal displays are text displays while the image display is a graphics display. In text mode the monitor is treated as an array of 25 rows and 80 columns. Each location may display a character. The text and background colors are selected with sixteen choices for a color monitor and two for a monochrome monitor. Each color may be presented in either high or low intensity, effectively doubling the number of available shades. To display a character, it is written to the computer's video memory storage area where the Video Graphics Adapter (VGA), displays the requested character in the desired color. Each location of the screen requires two bytes of memory for a total of 4000 bytes of memory which is stored by the VGA. The text displays are shown in Figures 34 and 35. The graphics display is shown in Figure 36.

In the image display screen the monitor is placed into graphics mode. In graphics mode the individual colors (red, green, and blue) comprising a single pixel must be placed into video memory for each display location. The amount of memory available in the VGA determines the number of colors displayed and the display's resolution. Various VGAs display graphics using a variety of formats to accommodate a variety of display sizes, resolutions and a sufficient range of colors. These various display formats use different memory mapping techniques to provide a larger resolution or greater color content. An explanation of memory mapping techniques (planar, packed pixel and others) can be found in [practical ray tracing]. The Video Electronics Standards Association (VESA), working with manufacturers, has promoted standardized display formats. A particular format can be identified by its VESA specified mode.

The system described in this work utilizes two video display formats, VESA modes 110 and 112. Each mode uses a planar memory model. In VESA mode 110, two bytes are used to represent each pixel. Five bits are allocated for the red and blue color

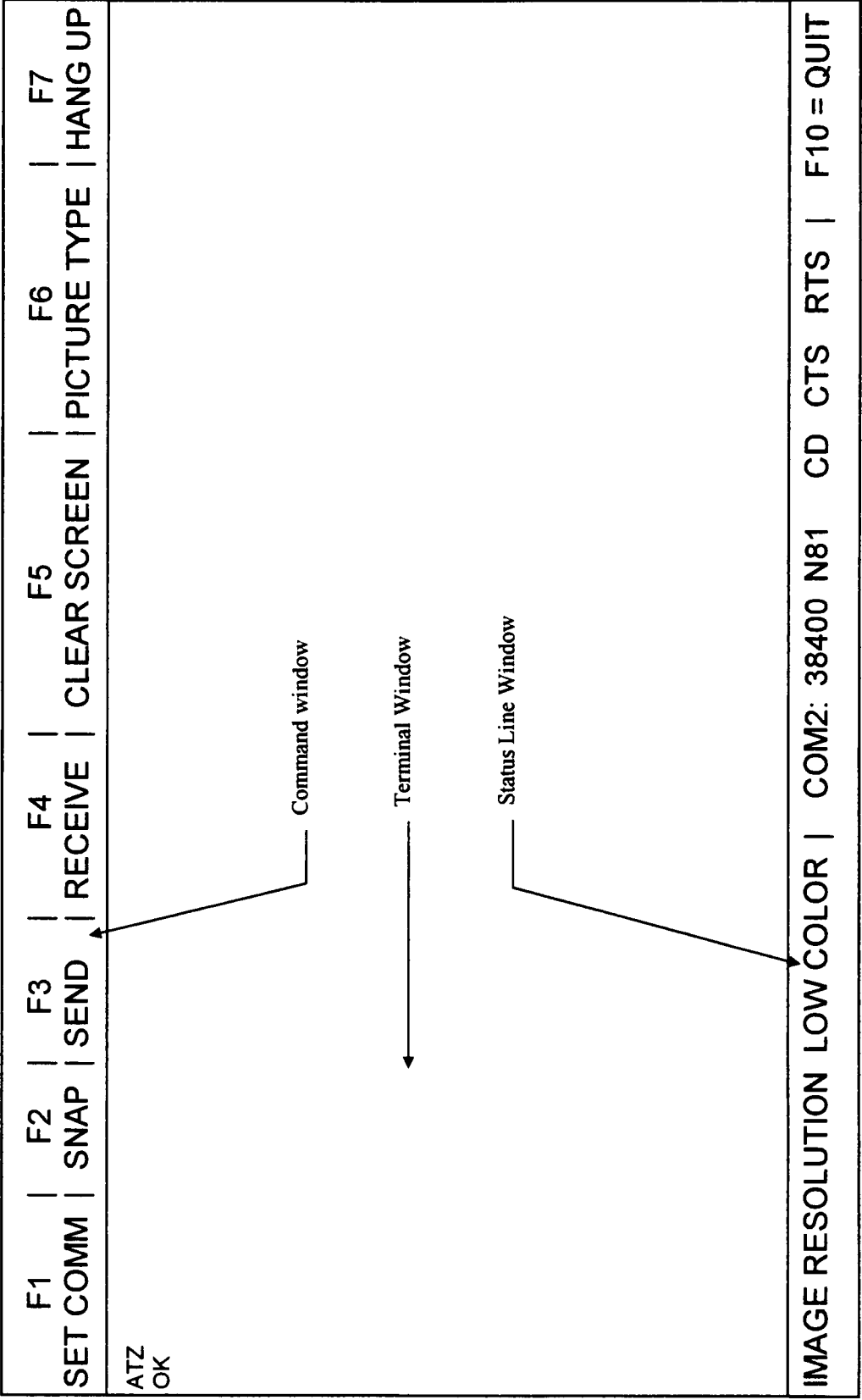


Figure 34 Terminal Display Screen

F1	F2	F3	F4	F5	F6	F7
SET COMM	SNAP	SEND	RECEIVE	CLEAR SCREEN	PICTURE TYPE	HANG UP

Set communications port (1), (2), (3), (4) ? 2

Using Interrupt (3) - (5) ? 3

Set speed (1)2400, (2)4800, (3)9600, (4)19200 (5)38400 ? 5

Set number of data bits (7) or (8) ? 8

Set parity (1)none, (2)even, (3)odd ? 1

Set stop bits (1) or (2) ? 1

Escape for default values.

Comport 2, irq 3, 38400 bps, 8N1 (8 data, No parity, 1 stop)

IMAGE RESOLUTION LOW COLOR | COM2: 38400 N81 CD CTS RTS | F10 = QUIT

Figure 35. Setup Display Screen

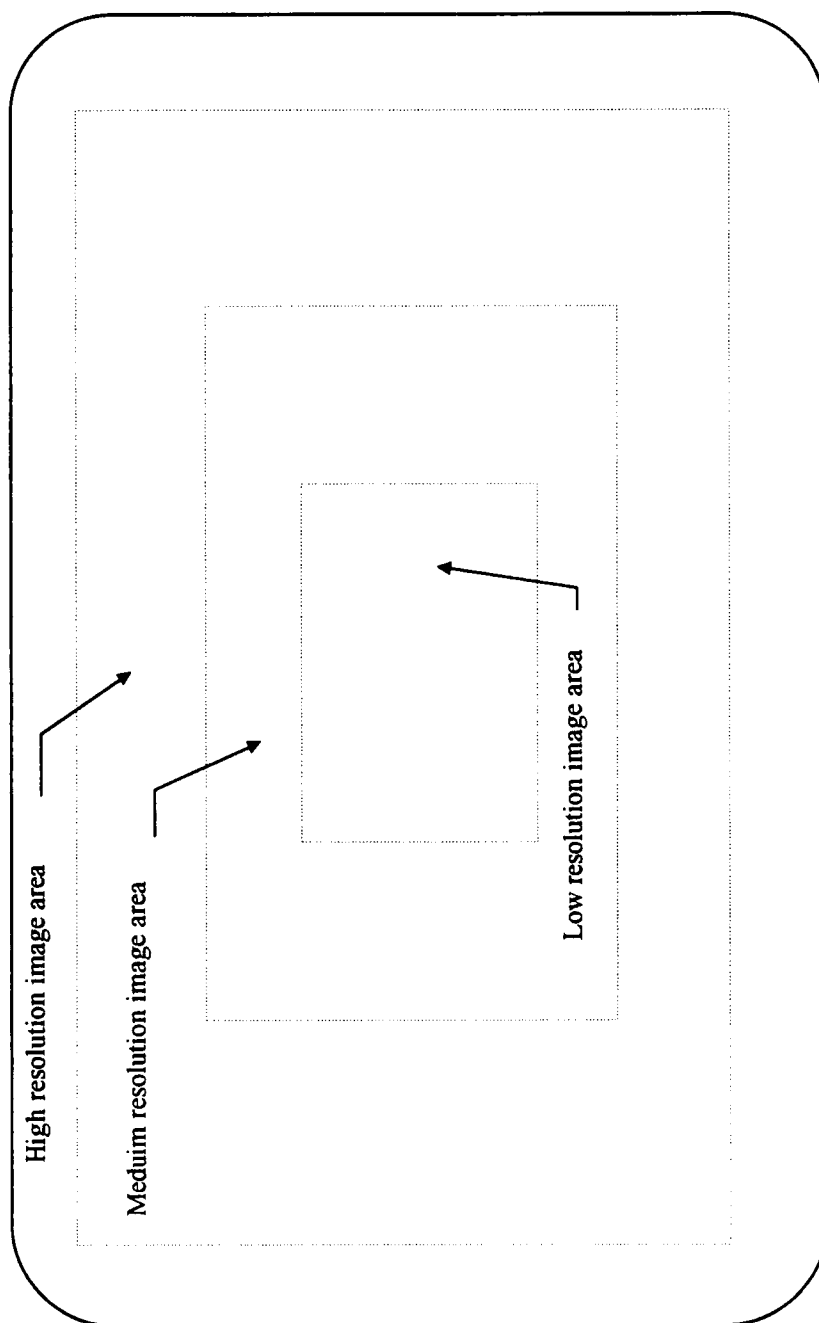


Figure 36. Image Display Screen. The dotted boxes show the area occupied by the image for the designated resolution.

components while six bits are allocated for green. This results in a total of 65,536 unique colors to represent the image. In VESA mode 112, three bytes are used to represent each pixel, using one byte for each color component. This allocation results in 1,677,216 unique colors to represent an image. When using VESA mode 110, colors may not appear to change shades continuously. A band of slightly different colors may be noticeable to the casual observer. In VESA mode 112, the transition between shades of colors will appear continuous to viewers.

VESA mode 110 assigns two bytes of memory for each pixel therefore a high resolution display (640x480) requires 614,400 bytes of video memory. VESA mode 112 assigns three bytes per pixel, therefore, the same image will need fifty percent more memory - nearly a megabyte.

When the PC first became available, it was assumed that only 128 Kbytes of video memory would ever be needed. To address memory beyond this video memory limit, the VGA uses bank switching. Each manufacturer of a VGA differs in how they implement their bank switching. The system described in this work is designed for a Cirrus Logic GL-GD5426, but will work with many other VGAs that address their bank switching similarly.

7.2 The Setup and Terminal Screens

The setup display is the first screen appearing to the system user. It is established using the function, *void QuerySetup(void)*, found in file *setup.cpp*, which can be found in Appendix II. To establish the setup display, an area of a monitor is defined, the color is set, and a boarder is drawn. The user is then queried for the needed comport parameters for initializing the comport and the asynchronous data format. Once these items are entered, the screen is removed and the terminal screen is established. The setup screen can be recalled by the user during program execution by using the command SET COM, F1. The setup screen shown in Figure 35 resulted from recalling setup after the program was first executed.

The terminal screen is established in much the same way as the display screen. It contains three sections: the command window, the terminal window, and the status line

window. The command window, which occupies the top two lines of the screen, displays the user commands needed to operate the system. The command window and user commands can be seen in Figure 34.

Just below the command window is the terminal window. The terminal screen occupies 21 lines and is used to display all modem commands and text entered through the keyboard, as well as any textual information received by the modem while the terminal window screen is active. The terminal window functions in scroll mode, meaning that when the terminal window is full, the cursor remains at the bottom and the text scrolls up one line. The terminal window is visible in Figure 34.

Beneath the terminal window is the status line. This window is divided into three sections. The left section presents the compressed image resolution and color settings. The center section shows the assigned comport settings and asynchronous data format settings and, if they are active, the modem's carrier detect (CD), RTS, and CTS signals. The right section displays the exit function command, QUIT, which causes the modem to hang up, resets all interrupts, deallocates memory used by the program, and returns control of the CPU to the operating system. The status line is updated each time a change occurs to an item which it displays.

7.3 The Image Display Screen.

The image display screen, when activated, stores the contents of the terminal screen, then clears the monitor and places it in the highest available and supported VESA video mode. The display screen is activated when either a SNAP, F2, or SEND, F3, command is entered through the terminal mode keyboard. The image display screen is activated also when a RECEIVE, F4, command is issued and compressed image data is received by the modem. Otherwise, the receive mode displays the message "waiting for an image, press any key to escape". Image data is displayed by the *put_pixel_xxx* function, found in file vid.cpp, which can be seen in Appendix II. Function *put_pixel_xxx* divides the screen into a 640x480 grid and writes each pixel by sending the appropriate data format to the proper VGA memory bank. Once an image is displayed, the monitor is returned to text mode, and the previous text display is replaced.

7.4 User Commands.

The user controls the system operation by using the commands shown in Figure 34. Each command requires only a single key stroke to execute. These commands are explained in Table 6.

Table 6. System Operating Commands

COMMAND	FUNCTION	DESCRIPTION
F1	SET COMM	Perform comport and data format initialization
F2	SNAP	Display the current image (from file)
F3	SEND	Compress and send the current image.
F4	RECEIVE	Receive and display a compressed image
F5	CLEAR SCREEN	Clear the Terminal Window
F6	PICTURE TYPE	Current image setting for SNAP and SEND
F7	HANG UP	Disconnect the modem and hang up telephone connection
F10	QUIT	Disconnect modem, hang up telephone, deallocate memory, reset interrupts and return to operation system

7.5 System Integration

System integration involves two elements: the hardware integration and software integration. The system described in this work, from a hardware viewpoint, is composed of two PCs, each equipped with an internal V.32bis, V.42bis modem and Cirrus Logic CL-GD5426 VGAs. They are interconnected via a telephone line. The system used by TN CAP for air to ground image transmission will be similar. It will use an external terminal network connector and packet radio as the data link, replacing the internal telephone modem and telephone line. Software controlling the X.25 interface will replace the modem controlling software. The prototype system described in this work, can be used between fixed ground stations.

The system's executable software is contained in thirteen files: comm.cpp, compress.cpp, dct.cpp, expand.cpp, global.cpp, image.cpp, initial.cpp, interpt.cpp, menu.cpp, setup.cpp, tables.cpp, take.cpp, and vid.cpp. Global.cpp contains all global variables for the program, function prototypes, and display messages. Image.cpp contains the function main which interprets all the user commands and directs system initialization. The functions contained in each file are shown in Table 7. A flowchart for main is shown in Figure 37.

The function main controls system operation. It first calls the function initialize(), which calls the subroutines necessary to initialize the system. Initialize() first calls VESAmode() to determine if VESA mode 110 or 112 is supported, then Initialize() sets the text screen mode. Next, it calls QuerySetup() to initialize the serial data ports and asynchronous data forms. Then it calls SetBuffers() to establish the input and output buffers, after which it initializes the modem. It then sets the modem initialization flag, and clears the modem status register. Next, the image segment storage area is allocated and the Huffman tables are loaded. Lastly, the screen's status line and menu are displayed. If the system is unable to complete the initialization, an appropriate error message is displayed. When Initialize() is finished, the terminal window is active and the system is operational.

System operation is now governed by the main program loop. This loop first checks the modem status change flag, and if the status has changed, the status line window is updated. Next, the program checks if the system operator has pressed a key. If a valid character has been pressed, it is written to the output screen. A character is then read from the input buffer and displayed in the terminal window. Lastly, the keystroke is checked against the valid commands. If a valid command has been issued, the appropriate function is executed. After the command is executed, the program loop begins again.

If the quit command is given, the function End() is called. It hangs up the modem, de-installs the modem interrupt, resets the display screen, frees allocated memory, displays

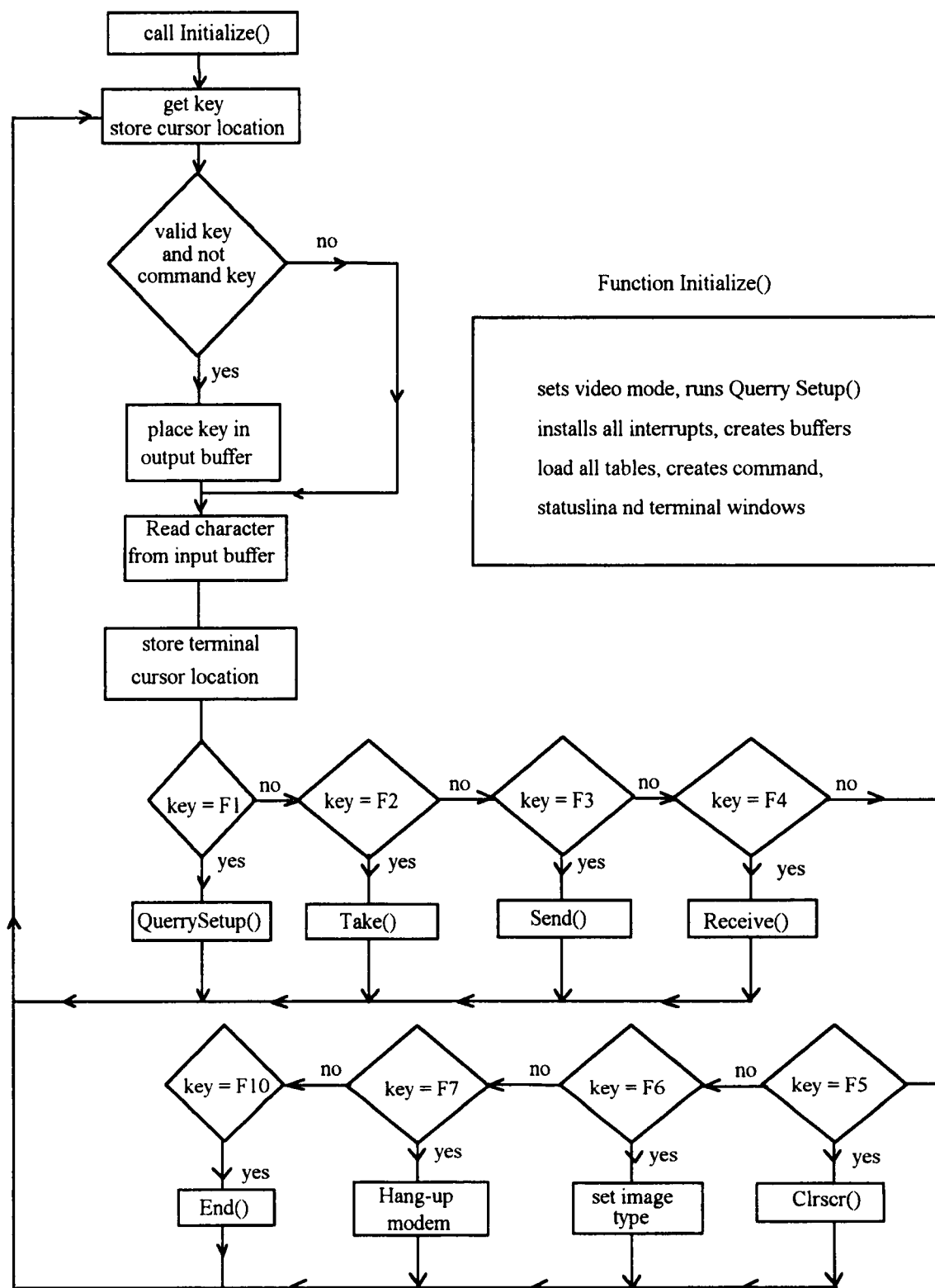


Figure 37. Flow Chart for Function void main(void).

Table 7. Source Code Files, Functions, and Function Descriptions.

File	Function	Function Description
Comm.cpp	void Queue(int c) unsigned int Dequeue(void) void ClearInputBuffer(void) void ClearOutputBuffer(void) void Write(int c) void Write(char *s) unsigned int Read(char *c) unsigned int Word_in(unsigned int &word_in) void RequestToSend(int f)	place character in input queue get character from output queue clear input buffer clear output buffer write character to output queue write string to output queue read character from input queue retrieve code word and marker flag set RTS signal
compress.cpp	void Compress(void)	compress image
dct.cpp	void dct(float *array) void idct(unsigned temp, float *a_p)	perform 2D DCT perform inverse 2-D DCT
expand.cpp	void Expand(void)	decompress image
image.cpp	void main(void) void End(char *msg)	main program quit program execution restore original conditions
initial.cpp	void Initialize(void)	initialization routine
interpt.cpp	void InstallModemInt(void) void DeInstallModemInt(void) void UartIntRoutine(void) void interrupt CriticalErrorHandler(...) void interrupt CommInt(...)	install modem interrupt deinstall modem interrupt modem service interrupt routine disable DOS error - abort, retry fail? load UartIntRoutine address
menu.cpp	int GetKey() void Menu(void)	get key stroke draw user command window
setup.cpp	void QuerySetup(void) void CheckTypePort(void) void SetSpeed(void) void SetInterrupt(void) void SetBuffers(void) void CreateTables(void)	draw setup window implement user settings Check for FIFO UART set comport data rate set modem interrupts set up input and output buffers load quantization and Huffman tables
take.cpp	void Take(void)	snap image

Table 7. (continued) Source Code Files, their Functions, and Function Descriptions.

File	Function	Function Description
vid.cpp	void SetTextMode(void)	Set text mode
	int SetVESAVideoMode (unsigned int Mode)	check and set VESA mode
	void put_pixel_110(int x,int y, UCHAR red, UCHAR green, UCHAR blue)	set pixel in VESA mode 110
	void put_pixel_112(int x,int y, UCHAR red, UCHAR green, UCHAR blue)	set pixel in VESA mode 112

an appropriate termination message, and returns system control to the disk operating system. The source code for main can be found in file image.cpp in Appendix II.

Next, Chapter 8 summarizes the system's performance, then presents conclusions based on the system's performance.

8.0 RESULTS AND CONCLUSIONS

8.1 System Performance Testing

The prototype system, in lieu of obtaining its image using a frame grabber circuit card, gets its image from an file stored on disk. The performance of the system described in this work was evaluated using a 640x480 high resolution color image where each pixel was represented by three bytes; one for each color component. Each data byte represents an unsigned number ranging from 0 to 255. Images are stored as unsigned values since they represent 256 discrete levels of the video signal which ranges from 0 to 1 volt. The values are converted to signed numbers for processing by subtracting 128 from each prior to compression. This conversion is needed since chrominance values may be negative and since the DCT uses signed numbers. Once the pixel values are recovered they are converted back to unsigned values and displayed. Using unsigned numbers reduces the magnitude of the numbers being represented thus reducing memory requirements and increasing operating speed.

The high resolution image is contained in the file high.rez. A medium resolution image is created from the high resolution image by replacing every 2x2 pixel block with a single pixel. This pixel's color components are the average values of the color components in the original image's 2x2 pixel block. A low resolution image is created from the medium resolution image using the same technique. Thus, the low and medium resolution images are a 4:1 and 16:1 compression of the high resolution image, and therefore have less detail and contain higher spatial frequencies. These images are stored in files medium.rez and low.rez, respectively.

System performance tests were conducted using a system comprised of an Intel 80486 microprocessor with a clock speed of 66 MHz, a Cirrus Logic CL-GD5426 VGA with 1 Mbyte of video RAM, and an internal V.32bis and V.42bis compatible Telephath II telephone modem. Using the default system settings a modem connection between two systems was established at 14,400 bps using V.42bis error correction and data compression. Each image was compressed, transmitted, received, and displayed using the

Table 8. Overall System Performance. The time to take, compress, transmit, receive decompress, and display an image is measured. The time is taken from when the F3 SEND key is hit until the image is fully displayed on the receiving station.

Image Type	Send to Receive and Display Time (seconds)	Image Quality
High color	23	Excellent
High monochrome	13	Excellent
Medium color	7	Good
Medium Monochrome	4	Good
Low Color	2.5	Adequate
Low Monochrome	1.5	Adequate

prototype system. Image quality was rated, and the overall time from start of image compression to end of image display was measured. These results are found in Table 8.

Execution time and the size of the compressed image data was also determined. To obtain time and size information, the prototype system was modified to save the compressed image data to a file and to provide execution time for obtaining the compressed image data file. The receive command was also modified to read the image from the compressed file and provide execution time to decompress the image data and display the recovered image. These results and the compressed file sizes are shown in Table 9.

The compressed image file was then transferred between systems using a telephone modem with file transfer protocols. The transfer times were obtained for x-modem, Kermit, file transfer protocols and for transfer using V.42 error correction with V.42bis compression. The transmission time was also obtained using no error correction or compression. These results are shown in Table 10.

8.2 Conclusions.

Based on these results, the system presented in this work provides good image quality in a reasonable time, thus meeting the first design goal. System operation needs only single key strokes to take, send, and receive an image, thus fulfilling the second design objective of ease of use. In addition, since the operator may view an image prior to transmission, the operator may select the most suitable image type and resolution before sending the image.

From Table 9 it can be seen that decompression takes about 57 percent of the time required for compression for a color image, and about 77 percent of the time for a monochrome image. Since the code word search is a little longer when decompressing (a search versus a look up table), the increased speed most likely results from using a 1-D DCT. In computing the first dimension (rows) of the 2-D DCT, only rows with non-zero entries are computed. Using these percentages and the number of multiplications needed to perform the transformation (Table 5), it can be concluded that, on average, for a color

Table 9. Prototype System Performance Results for Compression, Decompression and Image Quality. Overall compression includes the reduction obtained from transforming the high resolution image to the lower resolution prior to JPEG compression.

Image Resolution (Bytes)	Compressed Image Size	JPEG Compression Ratio	Overall Compression Ratio	Compression time (seconds)	Decompression time (seconds)	Image Quality
High color	36268	25.41:1	25.4:1	13.24	7.55	excellent
High monochrome	20863	14.72:1	14.7:1	6.53	4.99	excellent
Medium color	11693	19.70:1	78.8:1	3.35	1.92	good
Medium Monochrome	7573	10.14:1	40.6:1	1.65	1.27	good
Low Color	3602	16.00:1	256:1	0.88	0.50	poor
Low Monochrome	2290	8.38:1	134:1	0.44	0.33	poor

Table 10. Compressed Image Data Transmission Time Using Various Transfer Methods.

Image Type	Compressed Image Size (bytes)	Kermit (seconds)		X-modem (seconds)		V.42 and V.42bis (seconds)		Direct Transfer (seconds)	
High color	36268	54		44		23		28	
High monochrome	20863	33		28		13		17	
Medium color	11693	18		15		7		10	
Medium monochrome	7573	15		13		4		6	
Low Color	3602	10		10		2.5		4.5	
Low monochrome	2290	7		7		1.5		3.5	

image only 1 row of the inverse DCT is computed. Similarly, for a monochrome image, 4 rows on average must contain non-zero terms.

For the color image it can be concluded that the 1-D DCT will outperform a 2-D DCT (108 versus 144 multiplications). However, for monochrome images, they perform about equally (144 versus 145 multiplications). This result is image dependent, but it implies that for the inverse operation, a 1D DCT is a reasonable choice. This result is predictable since an image's chrominance information is primarily low frequency, while there are twice as many chrominance as luminance components in a color image.

Lastly, it can be seen that when using V.42 error correction and V.42bis data compression, compressed image transmission time is reduced. This result implies that the compressed image has repeated byte patterns which can be exploited using a textual data compression scheme. By viewing the compressed image data file, it can be seen that repeated patterns do occur. This result is also predictable.

In image areas where neighboring 8x8 image blocks have little detail, repeated image patterns occur. For example, if only a DC DCT coefficient is significant for three 8x8 blocks of a color image, the following symbol and corresponding data streams will result:

(0) (EOB) (0) (EOB) (0) (EOB) → 00 00 00 0101 00 0101

(0) (EOB) (0) (EOB) (0) (EOB) → 00 00 00 0101 00 0101

(0) (EOB) (0) (EOB) (0) (EOB) → 00 00 00 0101 00 0101.

The DC size symbol is zero since the DC term is differentially encoded. The resulting hexadecimal data stream 01 45 01 45 01 45 represents a repeated pattern whose redundancy can be exploited using a textual compression scheme.

8.3 Future Directions

The successful performance of this prototype system demonstrates that a low bandwidth still image transmission system using a JPEG image compression technique and low bandwidth data link can meet TN CAP's need for near real time image reconnaissance. In addition, the prototype system revealed that the image transmission

time can be reduced through the use of the V.42bis text compression scheme. Since the packer radio has nominal data rate of 9600 bps, a data compression technique should be employed to improve overall data throughput.

Lastly, the lower resolution image exhibits some undesirable attributes due to the loss of spectral frequency content during image compression. To improve the quality of these images some form of image processing may be desirable.

LIST OF REFERENCES

LIST OF REFERENCES

1. R. B. Arps, T. K. Truong, "Comparison of International Standards for Lossless Still Image Compression," Proceedings of the IEEE, pp. 889-899, June 1994.
2. T. P. Bizon, M. Shalkhauser, and W. A. Whyte, Jr, "Real-time transmission of digital video using variable-length coding," NASA Technical Memorandum 106092.
3. Borland C++ Version 4.0 Reference Manuals, Borland International, Inc, Scotts Valley, CA, 1993.
4. R. F. Haines, S. L. Chuang, "The effects of video compression on acceptability of images for monitoring life sciences experiments," NASA Technical Paper 3239, 1992.
5. M. A. Haque, "A two-dimensional fast cosine transform," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-33, pp. 1532-1539, December 1985.
6. S. A. Helmers, "Data Communications: A Beginner's Guide To Concepts And Technology," Prentice-Hall, Inc. NJ pp. 29-33, 125-139.
7. M. Holmes, B. Flanders, "C++ Communications Utilities," Ziff-Davis Press, CA, 1993.
8. B. W. Kernighan and D. M. Ritchie, "The C Programming Language, Second Edition," Prentice Hall, 1988.

9. B. G. Lee, "A new algorithm to compute the discrete cosine transform," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-32, pp. 1243-1245, December 1984.
10. M. Laurie, "Color and Resolution" Byte, pp. 171-176, July 1992.
11. B. Lindgren, "The X.25 Handbook," INFOTRANS, Sweden, 1990.
12. C. A. Lindsay, "Practical Ray Tracing," New York, NY, Wiley, 1992.
13. A. N. Nertravali, B. G. Haskell, "Digital Pictures: Representation and Compression," New York, NY, Plenum Press, 1988.
14. A. V. Oppenheim, R. W. Schaffer, "Discrete-Time Signal Processing," New Jersey, Prentice-Hall, 1989.
15. W. B. Pennebaker, J. L. Mitchell, "JPEG Still Image Data Compression Standard," New York, NY, Van Nostrand Reinhold, 1993.
16. K. R. Rao and P. Yip, "Discrete Cosine Transform Algorithms, Advantages, Applications," San Diego, CA, Academic Press, 1990.
17. H. Schildt, "C, The Pocket Reference, Second Edition," Osborne McGraw-Hill, CA, 1991.
18. B. Skylar, "Digital Communications," New Jersey, Prentice Hall, 1988 pp. 417-424.
19. C. Thomborson, "The V.42bis standard for data-compressing modems," IEEE Micro, pp. 41 - 53, October 1992.

20. G. Ungerboeck, "Channel coding with multilevel/phase signals," IEEE Transactions on Information Theory, vol. IT28, Jan. 1982, pp. 55-67.
21. G. K. Wallace, "The JPEG Still Picture Compression Standard," IEEE Transactions on Consumer Electronics, pp. xvii - xxxiv, February 1992.
22. Z Wang, "On computing the discrete Fourier and cosine transforms," IEEE Transactions on Acoustics, Speech, and Signal Processing, vol. ASSP-32, pp. 1341-1344, October 1985.
23. W. A. Whyte, Jr, K. Sayood, "Data compression for full motion video transmission," NASA Technical Memorandum 105239, 1991.
24. J. Ziv, A. Lempel, "Compression of individual sequences via variable-rate coding," IEEE Transactions on Information Theory, May 1978, pp. 530-536.

APPENDICES

APPENDIX I

LUMINANCE DC DIFFERENCE COEFFICIENTS

SIZE	Code Word		Code Length
	Decimal	Hexadecimal	
0	0	0000	2
1	16384	4000	3
2	24576	6000	3
3	32768	8000	3
4	20480	5000	3
5	49152	C000	3
6	57344	E000	4
7	61440	F000	5
8	63488	F800	6
9	64512	FC00	7
10	65024	FE00	8
11	65280	FF00	9

CHROMINANCE DC DIFFERENCE COEFFICIENTS

SIZE	Code Word		Code Length
	Decimal	Hexadecimal	
0	0	0000	2
1	16384	4000	2
2	32768	8000	3
3	49152	C000	3
4	57344	E000	4
5	61440	F000	5
6	63488	F800	6
7	64512	FC00	7
8	64512	FC00	8
9	65280	FF00	9
10	65408	FF80	10
11	65472	FFC0	11

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
40960	A000	0	0	4	1
0	0	0	1	2	0
16384	4000	0	2	2	0
32768	8000	0	3	3	1
45056	B000	0	4	4	1
53248	D000	0	5	5	2
61440	F000	0	6	7	4
63488	F800	0	7	8	5
64896	FD80	0	8	10	6
65410	FF82	0	9	16	9
65411	FF83	0	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
49152	C000	1	1	4	2
55296	D800	1	2	5	2
61952	F200	1	3	7	4
64256	FB00	1	4	9	5
65216	FEC0	1	5	11	7
65412	FF84	1	6	16	9
65413	FF85	1	7	16	9
65414	FF86	1	8	16	9
65415	FF87	1	9	16	9
65416	FF88	1	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
57344	E000	2	1	5	3
63744	F900	2	2	8	5
64960	FDC0	2	3	10	6
65344	FF40	2	4	12	8
65417	FF89	2	5	16	9
65418	FF8A	2	6	16	9
65419	FF8B	2	7	16	9
65420	FF8C	2	8	16	9
65421	FF8D	2	9	16	9
65422	FF8E	2	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
59392	E800	3	1	6	3
64384	FB80	3	2	9	5
65360	FF50	3	3	12	8
65423	FF8F	3	4	16	9
65424	FF90	3	5	16	9
65425	FF91	3	6	16	9
65426	FF92	3	7	16	9
65427	FF93	3	8	16	9
65428	FF94	3	9	16	9
65429	FF95	3	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
60416	EC00	4	1	6	3
65024	FE00	4	2	10	7
65430	FF96	4	3	16	9
65431	FF97	4	4	16	9
65432	FF98	4	5	16	9
65433	FF99	4	6	16	9
65434	FF9A	4	7	16	9
65435	FF9B	4	8	16	9
65436	FF9C	4	9	16	9
65437	FF9D	4	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
62464	F400	5	1	7	4
65248	FEE0	5	2	11	7
65438	FF9E	5	3	16	9
65439	FF9F	5	4	16	9
65440	FFA0	5	5	16	9
65441	FFA1	5	6	16	9
65442	FFA2	5	7	16	9
65443	FFA3	5	8	16	9
65444	FFA4	5	9	16	9
65445	FFA5	5	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
62976	F600	6	1	7	4
65376	FF60	6	2	12	8
65446	FFA6	6	3	16	9
65447	FFA7	6	4	16	9
65448	FFA8	6	5	16	9
65449	FFA9	6	6	16	9
65450	FFAA	6	7	16	9
65451	FFAB	6	8	16	9
65452	FFAC	6	9	16	9
65453	FFAD	6	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64000	FA00	7	1	8	5
65392	FF70	7	2	12	8
65454	FFAE	7	3	16	9
65455	FFAF	7	4	16	9
65456	FFB0	7	5	16	9
65457	FFB1	7	6	16	9
65458	FFB2	7	7	16	9
65459	FFB3	7	8	16	9
65460	FFB4	7	9	16	9
65461	FFB5	7	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64512	FC00	8	1	9	6
65408	FF80	8	2	15	9
65462	FFB6	8	3	16	9
65463	FFB7	8	4	16	9
65464	FFB8	8	5	16	9
65465	FFB9	8	6	16	9
65466	FFBA	8	7	16	9
65467	FFBB	8	8	16	9
65468	FFBC	8	9	16	9
65469	FFBD	8	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64640	FC80	9	1	9	6
65470	FFBE	9	2	16	9
65471	FFBF	9	3	16	9
65472	FFC0	9	4	16	10
65473	FFC1	9	5	16	10
65474	FFC2	9	6	16	10
65475	FFC3	9	7	16	10
65476	FFC4	9	8	16	10
65477	FFC5	9	9	16	10
65478	FFC6	9	A	16	10
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64768	FD00	A	1	9	6
65479	FFC7	A	2	16	10
65480	FFC8	A	3	16	10
65481	FFC9	A	4	16	10
65482	FFCA	A	5	16	10
65483	FFDB	A	6	16	10
65484	FFCC	A	7	16	10
65485	FFCD	A	8	16	10
65486	FFCE	A	9	16	10
65487	FFCF	A	A	16	10
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65088	FE40	B	1	10	7
65488	FFD0	B	2	16	10
65489	FFD1	B	3	16	10
65490	FFD2	B	4	16	10
65491	FFD3	B	5	16	10
65492	FFD4	B	6	16	10
65493	FFD5	B	7	16	10
65494	FFD6	B	8	16	10
65495	FFD7	B	9	16	10
65496	FFD8	B	A	16	10
0	0	n/a	n/a	n/a	n/a

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65152	FE80	C	1	10	7
65497	FFD9	C	2	16	10
65498	FFDA	C	3	16	10
65499	FFDB	C	4	16	10
65500	FFDC	C	5	16	10
65501	FFDD	C	6	16	10
65502	FFDE	C	7	16	10
65503	FFDF	C	8	16	10
65504	FFE0	C	9	16	11
65505	FFE1	C	A	16	11
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65280	FF00	D	1	11	8
65506	FFE2	D	2	16	11
65507	FFE3	D	3	16	11
65508	FFE4	D	4	16	11
65509	FFE5	D	5	16	11
65510	FFE6	D	6	16	11
65511	FFE7	D	7	16	11
65512	FFE8	D	8	16	11
65513	FFE9	D	9	16	11
65514	FFEA	D	A	16	11
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65515	FFEB	E	1	16	11
65516	FFEC	E	2	16	11
65517	FFED	E	3	16	11
65518	FFEE	E	4	16	11
65519	FFEF	E	5	16	11
65520	FFF0	E	6	16	12
65521	FFF1	E	7	16	12
65522	FFF2	E	8	16	12
65523	FFF3	E	9	16	12
65524	FFF4	E	A	16	12

AC LUMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65312	FF20	F	0	11	8
65525	FFF5	F	1	16	12
65526	FFF6	F	2	16	12
65527	FFF7	F	3	16	12
65528	FFF8	F	4	16	13
65529	FFF9	F	5	16	13
65530	FFFA	F	6	16	13
65531	FFFB	F	7	16	13
65532	FFFC	F	8	16	14
65533	FFFD	F	9	16	14
65534	FFFE	F	A	16	15

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	0	0	2	0
16384	4000	0	1	2	0
32768	8000	0	2	3	1
40960	A000	0	3	4	1
49152	C000	0	4	5	2
51200	C800	0	5	5	2
57344	E000	0	6	6	3
61440	F000	0	7	7	4
64000	FA00	0	8	9	5
64896	FD80	0	9	10	6
65344	FF40	0	A	12	8
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
45056	B000	1	1	4	1
58368	E400	1	2	6	3
62976	F600	1	3	8	4
64128	FA80	1	4	9	5
65216	FE66	1	5	11	7
65360	FF50	1	6	12	8
65416	FF89	1	7	16	9
65417	FF8A	1	8	16	9
65418	FF8B	1	9	16	9
65419	FF8C	1	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
53248	D000	2	1	5	2
63232	F700	2	2	8	4
64960	FDC0	2	3	10	6
65376	FF60	2	4	12	8
65412	FF84	2	5	15	9
65420	FF8D	2	6	16	9
65421	FF8E	2	7	16	9
65422	FF8F	2	8	16	9
65423	FF90	2	9	16	9
65424	FF91	2	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
55296	D800	3	1	5	2
63488	F800	3	2	8	5
65024	FE00	3	3	10	7
65392	FF70	3	4	12	8
65425	FF91	3	5	16	9
65426	FF92	3	6	16	9
65427	FF93	3	7	16	9
65428	FF94	3	8	16	9
65429	FF95	3	9	16	9
65430	FF96	3	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
59392	E800	4	1	6	3
64256	FB00	4	2	9	5
65431	FF97	4	3	16	9
65432	FF98	4	4	16	9
65433	FF99	4	5	16	9
65434	FF9A	4	6	16	9
65435	FF9B	4	7	16	9
65436	FF9C	4	8	16	9
65437	FF9D	4	9	16	9
65438	FF9E	4	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
60416	EC00	5	1	6	3
65088	FE40	5	2	10	7
65439	FF9F	5	3	16	9
65440	FFA0	5	4	16	9
65441	FFA1	5	5	16	9
65442	FFA2	5	6	16	9
65443	FFA3	5	7	16	9
65444	FFA4	5	8	16	9
65445	FFA5	5	9	16	9
65446	FFA6	5	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
61952	F200	6	1	7	4
65248	FEE0	6	2	11	7
65447	FFA7	6	3	16	9
65448	FFA8	6	4	16	9
65449	FFA9	6	5	16	9
65450	FFAA	6	6	16	9
65451	FFAB	6	7	16	9
65452	FFAC	6	8	16	9
65453	FFAD	6	9	16	9
65454	FFAE	6	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
62464	F400	7	1	7	4
65280	FF00	7	2	11	8
65455	FFAF	7	3	16	9
65456	FFB0	7	4	16	9
65457	FFB1	7	5	16	9
65458	FFB2	7	6	16	9
65459	FFB3	7	7	16	9
65460	FFB4	7	8	16	9
65461	FFB5	7	9	16	9
65462	FFB6	7	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
63744	F900	8	1	8	5
65463	FFB7	8	2	16	9
65464	FFB8	8	3	16	9
65465	FFB9	8	4	16	9
65466	FFBA	8	5	16	9
65467	FFBB	8	6	16	9
65468	FFBC	8	7	16	9
65469	FFBD	8	8	16	9
65470	FFBE	8	9	16	9
65471	FFBF	8	A	16	9
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64384	FB80	9	1	9	5
65472	FFC0	9	2	16	10
65473	FFC1	9	3	16	10
65474	FFC2	9	4	16	10
65475	FFC3	9	5	16	10
65476	FFC4	9	6	16	10
65477	FFC5	9	7	16	10
65478	FFC6	9	8	16	10
65479	FFC7	9	9	16	10
65480	FFC8	9	A	16	10
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64512	FA98	A	1	9	6
65481	FFC9	A	2	16	10
65482	FFCA	A	3	16	10
65483	FFCB	A	4	16	10
65484	FFCC	A	5	16	10
65485	FFCD	A	6	16	10
65486	FFCE	A	7	16	10
65487	FFCF	A	8	16	10
65488	FFD0	A	9	16	10
65489	FFD1	A	A	16	10
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64640	FC80	B	1	9	6
65490	FFD2	B	2	16	10
65491	FFD3	B	3	16	10
65492	FFD4	B	4	16	10
65493	FFD5	B	5	16	10
65494	FFD6	B	6	16	10
65495	FFD7	B	7	16	10
65496	FFD8	B	8	16	10
65497	FFD9	B	9	16	10
65498	FFDA	B	A	16	10
0	0	n/a	n/a	n/a	n/a

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
64768	FD00	C	1	9	6
65499	FFDB	C	2	16	10
65500	FFDC	C	3	16	10
65501	FFDD	C	4	16	10
65502	FFDE	C	5	16	10
65503	FFDF	C	6	16	10
65504	FFE0	C	7	16	11
65505	FFE1	C	8	16	11
65506	FFE2	C	9	16	11
65507	FFE3	C	A	16	11
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65312	FF20	D	1	11	8
65508	FFE4	D	2	16	11
65509	FFE5	D	3	16	11
65510	FFE6	D	4	16	11
65511	FFE7	D	5	16	11
65512	FFE8	D	6	16	11
65513	FFE9	D	7	16	11
65514	FFEA	D	8	16	11
65515	FFEB	D	9	16	11
65516	FFEC	D	A	16	11
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65408	FF80	E	1	14	9
65517	FFED	E	2	16	11
65518	FFEE	E	3	16	11
65519	FFEF	E	4	16	11
65520	FFF0	E	5	16	12
65521	FFF1	E	6	16	12
65522	FFF2	E	7	16	12
65523	FFF3	E	8	16	12
65524	FFF4	E	9	16	12
65525	FFF5	E	A	16	12

AC CHROMINANCE HUFFMAN CODE

Code Decimal	Code Hexadecimal	RUNLENGTH Symbol	SIZE Symbol	CodeLength bits	Number of Leading ones
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
0	0	n/a	n/a	n/a	n/a
65152	FE80	F	0	10	7
65414	FF86	F	1	15	9
65526	FFF6	F	2	16	12
65527	FFF7	F	3	16	12
65528	FFF8	F	4	16	13
65529	FFF9	F	5	16	13
65530	FFFA	F	6	16	13
65531	FFFB	F	7	16	13
65532	FFFC	F	8	16	14
65533	FFFD	F	9	16	14
65534	FFFE	F	A	16	15

INVERSE AC LUMINANCE HUFFMAN CODE

Code Word	RUN LENGTH	Code SIZE	Code Length	Number of Leading 1s	Code Word	RUN LENGTH	Code SIZE	Code Length	Number of Leading 1s
0	0	1	2	0	FF8B	2	7	16	9
4000	0	2	2	0	FF8C	2	8	16	9
8000	0	3	3	1	FF8D	2	9	16	9
A000	0	0	4	1	FF8E	2	A	16	9
B000	0	4	4	1	FF8F	3	4	16	9
C000	1	1	4	2	FF90	3	5	16	9
D000	0	5	5	2	FF91	3	6	16	9
D800	1	2	5	2	FF92	3	7	16	9
E000	2	1	5	3	FF93	3	8	16	9
E800	3	1	6	3	FF94	3	9	16	9
EC00	4	1	6	3	FF95	3	A	16	9
F000	0	6	7	4	FF96	4	3	16	9
F200	1	3	7	4	FF97	4	4	16	9
F400	5	1	7	4	FF98	4	5	16	9
F600	6	1	7	4	FF99	4	6	16	9
F800	0	7	8	5	FF9A	4	7	16	9
F900	2	2	8	5	FF9B	4	8	16	9
FA00	7	1	8	5	FF9C	4	9	16	9
FB00	1	4	9	5	FF9D	4	A	16	9
FB80	3	2	9	5	FF9E	5	3	16	9
FC00	8	1	9	6	FF9F	5	4	16	9
FC80	9	1	9	6	FFA0	5	5	16	9
FD00	A	1	9	6	FFA1	5	6	16	9
FD80	0	8	10	6	FFA2	5	7	16	9
FDC0	2	3	10	6	FFA3	5	8	16	9
FE00	4	2	10	7	FFA4	5	9	16	9
FE40	B	1	10	7	FFA5	5	A	16	9
FE80	C	1	10	7	FFA6	6	3	16	9
FEC0	1	5	11	7	FFA7	6	4	16	9
FEE0	5	2	11	7	FFA8	6	5	16	9
FF00	D	1	11	8	FFA9	6	6	16	9
FF20	F	0	11	8	FFAA	6	7	16	9
FF40	2	4	12	8	FFAB	6	8	16	9
FF50	3	3	12	8	FFAC	6	9	16	9
FF60	6	2	12	8	FFAD	6	A	16	9
FF70	7	2	12	8	FFAE	7	3	16	9
FF80	8	2	15	9	FFAF	7	4	16	9
FF82	0	9	16	9	FFB0	7	5	16	9
FF83	0	A	16	9	FFB1	7	6	16	9
FF84	1	6	16	9	FFB2	7	7	16	9
FF85	1	7	16	9	FFB3	7	8	16	9
FF86	1	8	16	9	FFB4	7	9	16	9
FF87	1	9	16	9	FFB5	7	A	16	9
FF88	1	A	16	9	FFB6	8	3	16	9
FF89	2	5	16	9	FFB7	8	4	16	9
FF8A	2	6	16	9	FFB8	8	5	16	9

INVERSE AC LUMINANCE HUFFMAN CODE

Code Word	RUN LENGTH	Code SIZE	Code Length	Number of Leading 1s	Code Word	RUN LENGTH	Code SIZE	Code Length	Number of Leading 1s
FFB9	8	6	16	9	FFDC	C	5	16	10
FFBA	8	7	16	9	FFDD	C	6	16	10
FFBB	8	8	16	9	FFDE	C	7	16	10
FFBC	8	9	16	9	FFDF	C	8	16	10
FFBD	8	A	16	9	FFE0	C	9	16	11
FFBE	9	2	16	9	FFE1	C	A	16	11
FFBF	9	3	16	9	FFE2	D	2	16	11
FFC0	9	4	16	10	FFE3	D	3	16	11
FFC1	9	5	16	10	FFE4	D	4	16	11
FFC2	9	6	16	10	FFE5	D	5	16	11
FFC3	9	7	16	10	FFE6	D	6	16	11
FFC4	9	8	16	10	FFE7	D	7	16	11
FFC5	9	9	16	10	FFE8	D	8	16	11
FFC6	9	A	16	10	FFE9	D	9	16	11
FFC7	A	2	16	10	FFEA	D	A	16	11
FFC8	A	3	16	10	FFEB	E	1	16	11
FFC9	A	4	16	10	FFEC	E	2	16	11
FFCA	A	5	16	10	FFED	E	3	16	11
FFDB	A	6	16	10	FFEE	E	4	16	11
FFCC	A	7	16	10	FFEF	E	5	16	11
FFCD	A	8	16	10	FFF0	E	6	16	12
FFCE	A	9	16	10	FFF1	E	7	16	12
FFCF	A	A	16	10	FFF2	E	8	16	12
FFD0	B	2	16	10	FFF3	E	9	16	12
FFD1	B	3	16	10	FFF4	E	A	16	12
FFD2	B	4	16	10	FFF5	F	1	16	12
FFD3	B	5	16	10	FFF6	F	2	16	12
FFD4	B	6	16	10	FFF7	F	3	16	12
FFD5	B	7	16	10	FFF8	F	4	16	13
FFD6	B	8	16	10	FFF9	F	5	16	13
FFD7	B	9	16	10	FFFA	F	6	16	13
FFD8	B	A	16	10	FFFB	F	7	16	13
FFD9	C	2	16	10	FFFC	F	8	16	14
FFDA	C	3	16	10	FFFD	F	9	16	14
FFDB	C	4	16	10	FFFE	F	A	16	15

INVERSE AC CHROMINANCE CODES

Code Word	RUN LENGTH	SIZE	Code Length	Number of Leading 1s	Code Word	RUN LENGTH	SIZE	Code Length	Number of Leading 1s
0	0	0	2	0	FF8D	2	6	16	9
4000	0	1	2	0	FF8E	2	7	16	9
8000	0	2	3	1	FF8F	2	8	16	9
A000	0	3	4	1	FF90	2	9	16	9
B000	1	1	4	1	FF91	2	A	16	9
C000	0	4	5	2	FF91	3	5	16	9
C800	0	5	5	2	FF92	3	6	16	9
D000	2	1	5	2	FF93	3	7	16	9
D800	3	1	5	2	FF94	3	8	16	9
E000	0	6	6	3	FF95	3	9	16	9
E400	1	2	6	3	FF96	3	A	16	9
E800	4	1	6	3	FF97	4	3	16	9
EC00	5	1	6	3	FF98	4	4	16	9
F000	0	7	7	4	FF99	4	5	16	9
F200	6	1	7	4	FF9A	4	6	16	9
F400	7	1	7	4	FF9B	4	7	16	9
F600	1	3	8	4	FF9C	4	8	16	9
F700	2	2	8	4	FF9D	4	9	16	9
F800	3	2	8	5	FF9E	4	A	16	9
F900	8	1	8	5	FF9F	5	3	16	9
FA00	0	8	9	5	FFA0	5	4	16	9
FA80	1	4	9	5	FFA1	5	5	16	9
FB00	4	2	9	5	FFA2	5	6	16	9
FB80	9	1	9	5	FFA3	5	7	16	9
FA98	A	1	9	6	FFA4	5	8	16	9
FC80	B	1	9	6	FFA5	5	9	16	9
FD00	C	1	9	6	FFA6	5	A	16	9
FD80	0	9	10	6	FFA7	6	3	16	9
FDC0	2	3	10	6	FFA8	6	4	16	9
FE00	3	3	10	7	FFA9	6	5	16	9
FE40	5	2	10	7	FFAA	6	6	16	9
FE80	F	0	10	7	FFAB	6	7	16	9
FE66	1	5	11	7	FFAC	6	8	16	9
FEE0	6	2	11	7	FFAD	6	9	16	9
FF00	7	2	11	8	FFAE	6	A	16	9
FF20	D	1	11	8	FFAF	7	3	16	9
FF40	0	A	12	8	FFB0	7	4	16	9
FF50	1	6	12	8	FFB1	7	5	16	9
FF60	2	4	12	8	FFB2	7	6	16	9
FF70	3	4	12	8	FFB3	7	7	16	9
FF80	E	1	14	9	FFB4	7	8	16	9
FF84	2	5	15	9	FFB5	7	9	16	9
FF86	F	1	15	9	FFB6	7	A	16	9
FF89	1	7	16	9	FFB7	8	2	16	9
FF8A	1	8	16	9	FFB8	8	3	16	9
FF8B	1	9	16	9	FFB9	8	4	16	9
FF8C	1	A	16	9	FFBA	8	5	16	9

INVERSE AC CHROMINANCE CODES

Code Word	RUN LENGTH	SIZE	Code Length	Number of Leading 1s	Code Word	RUN LENGTH	SIZE	Code Length	Number of Leading 1s
FFBB	8	6	16	9	FFDD	C	4	16	10
FFBC	8	7	16	9	FFDE	C	5	16	10
FFBD	8	8	16	9	FFDF	C	6	16	10
FFBE	8	9	16	9	FFE0	C	7	16	11
FFBF	8	A	16	9	FFE1	C	8	16	11
FFC0	9	2	16	10	FFE2	C	9	16	11
FFC1	9	3	16	10	FFE3	C	A	16	11
FFC2	9	4	16	10	FFE4	D	2	16	11
FFC3	9	5	16	10	FFE5	D	3	16	11
FFC4	9	6	16	10	FFE6	D	4	16	11
FFC5	9	7	16	10	FFE7	D	5	16	11
FFC6	9	8	16	10	FFE8	D	6	16	11
FFC7	9	9	16	10	FFE9	D	7	16	11
FFC8	9	A	16	10	FFEA	D	8	16	11
FFC9	A	2	16	10	FFEB	D	9	16	11
FFCA	A	3	16	10	FFEC	D	A	16	11
FFCB	A	4	16	10	FFED	E	2	16	11
FFCC	A	5	16	10	FFEE	E	3	16	11
FFCD	A	6	16	10	FFEF	E	4	16	11
FFCE	A	7	16	10	FFF0	E	5	16	12
FFCF	A	8	16	10	FFF1	E	6	16	12
FFD0	A	9	16	10	FFF2	E	7	16	12
FFD1	A	A	16	10	FFF3	E	8	16	12
FFD2	B	2	16	10	FFF4	E	9	16	12
FFD3	B	3	16	10	FFF5	E	A	16	12
FFD4	B	4	16	10	FFF6	F	2	16	12
FFD5	B	5	16	10	FFF7	F	3	16	12
FFD6	B	6	16	10	FFF8	F	4	16	13
FFD7	B	7	16	10	FFF9	F	5	16	13
FFD8	B	8	16	10	FFFA	F	6	16	13
FFD9	B	9	16	10	FFFB	F	7	16	13
FFDA	B	A	16	10	FFFC	F	8	16	14
FFDB	C	2	16	10	FFFD	F	9	16	14
FFDC	C	3	16	10	FFFE	F	A	16	15

LUMINANCE QUANTIZATION TABLE

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

CHROMINANCE QUANTIZATION TABLE

17	18	24	47	99	99	99	99
18	21	26	66	99	99	99	99
24	26	56	99	99	99	99	99
47	66	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99
99	99	99	99	99	99	99	99

APPENDIX II

File : IMAGE.CPP

```

#include <stdio.h>           // standard I/O library
#include <conio.h>           // console I/O function library
#include <string.h>          // string handling routine library
#include <stdlib.h>          // std conversion routine library
#include <dos.h>             // dos function library
#include <bios.h>            // BIOS function library
#include <io.h>              // I/O function library
#include <malloc.h>          // memory allocation library
#include <math.h>            // math function library
#include <float.h>           // floating point library

extern unsigned _stklen = 8192;

// set stack length

#include "global.cpp"        // include source codes
#include "interpt.cpp"        // variables and prototypes
#include "setup.cpp"          // interrupt routines
#include "comm.cpp"           // set up and initialization routines
#include "dct.cpp"            // handles buffers for comm ports
#include "compress.cpp"        // performs dct and idct
#include "expand.cpp"          // video compression algorithm
#include "initial.cpp"         // video reconstruction routine
#include "menu.cpp"            // initialization routines
#include "tables.cpp"          // menu display
#include "take.cpp"            // loads quantization and Huffman code
#include "vid.cpp"             // picture read and display
                               // video initialization and display routines

void main(void) {
    unsigned int  key, read, cursor_x, cursor_y;
    unsigned char c;
    Initialize();              // initialize all parameters
    Write(INITIAL_MODEM);      // reset modem
    while (1){                 // program loop
        gotoxy(cursor_x, cursor_y); // set cursor location
        if (msr_changed){      // check MSR and update status line
            msr_changed = 0;
            StatusLine();
        }
        key = GetKey();         // get user request
        if ((key != 0) && (key < 0x100)) Write(key); // if not alt key display
        if (key == 0xd) Write(0xa); // if carriage return add linefeed
        read = Read(&c);        // read any input from modem
        if (read == 0) printf("%c", c); // if data present, print it
        cursor_x = wherex();    // get x cursor position
        cursor_y = wherey();    // get y cursor position
        if (key == 0x13b) {     // setup queues, modem, drivers
            outportb(base + MCR, MCR_DO & ~MCR_DTR); // hang-up modem
            ClearInputBuffer(); // clear input buffer
            ClearOutputBuffer(); // and output buffer
            DeInstallModemInt(); // remove port interrupt
            QuerySetup();        // query setup
        }
    }
}

```

File : IMAGE.CPP

```

InstallModemInt();           // install new interrupt
Write(INITIAL_MODEM);       // reset modem
msr_changed = 1;           // flag port change
StatusLine();               // update status line
}
if (key == 0x13c) {          // get image?
    gettext(1,1,80,25,OldWindowData); // save present window
    Take();                  // take picture
    puttext(1,1,80,25,OldWindowData); // replace window
    window(1,3,80,23);      // activate terminal screen
    textcolor(TerminalForegroundColor); // set terminal colors
    textbackground(TerminalBackgroundColor);
    _wscroll = 1;          // scroll screen
    StatusLine();          // update status line
}
if (key == 0x13d) {          // Send Image
    gettext(1,1,80,25,OldWindowData); // save window
    Compress();             // call compress
    puttext(1,1,80,25,OldWindowData); // replace window
    window(1,3,80,23);      // activate terminal window
    textcolor(TerminalForegroundColor); // set colors
    textbackground(TerminalBackgroundColor);
    _wscroll = 1;          // set scroll
    StatusLine();          // update status line
}
if (key == 0x13e) {          // receive image
    gettext(1,1,80,25,OldWindowData); // save window
    Expand();               // call expand
    puttext(1,1,80,25,OldWindowData); // replace window
    window(1,3,80,23);      // activate terminal window
    textcolor(TerminalForegroundColor); // set colors
    textbackground(TerminalBackgroundColor);
    _wscroll = 1;          // set scroll
    StatusLine();          // update status line
}
if (key == 0x13f) {          //clear screen
    clrscr();
    cursor_x = 1;           // set cursor to
    cursor_y = 1;           // top left of screen
    StatusLine();           // update status line
    window(1,3,80,23);      // activate terminal window
    textcolor(TerminalForegroundColor); // set colors
    textbackground(TerminalBackgroundColor);
    _wscroll = 1;          // set scroll
    StatusLine();          // update status line
}
if (key == 0x140) {          //select image attributes
    pict--;
    if (pict < 0) pict = 5;
    StatusLine();

```

File : IMAGE.CPP

```

    }
    if (key == 0X141) {
        outportb(base + MCR, MCR_DO & ~MCR_DTR); // hang-up modem
        delay(1550); // disconnect modem
        outportb(base + MCR, MCR_DO); // wait a bit
        StatusLine(); // raise DTR
    }
    if (key == 0x144) End(Done); // quit program
}

void End(char *msg) { // end program routine
    if (InitComplete) { // if initialization was complete
        outportb(base + MCR, MCR_DO & ~MCR_DTR); //hang up modem
        delay(1550); // wait
        outportb(base + MCR, MCR_DO); // raise DTR
        DeInstallModemInt(); // deinstall modem interrupt
        window(1,1,80,max_lines); // reset window size to full screen
        textcolor(WHITE); // set white on black screen
        textbackground(BLACK);
        clrscr();
        _setcursortype(_NORMALCURSOR); // replace cursor type
        farfree(picture_data); // free all allocated memory
        farfree(o_buf); // free all allocated memory
        farfree(i_buf); // free all allocated memory
    }
    printf(msg); // print exit message
    delay(2000); // wait
    clrscr(); // clear screen
    exit(EXIT_SUCCESS); // return exit value
}

```

File: GLOBAL.CPP

```

// Global Defines for UART and Modem
#define NOT ! // shorthand logical
#define UINT unsigned int // unsigned integer
#define UCHAR unsigned char // unsigned character
// UART regs (base address +)
#define RBR 0 // receive buffer register
#define THR 0 // transmit holding register
#define DLL 0 // divisor latch LSB
#define DLM 1 // divisor latch MSB
#define IER 1 // interrupt enable register
#define IIR 2 // interrupt id register
#define FCR 2 // FIFO control register
#define LCR 3 // line control register
#define MCR 4 // modem control register
#define LSR 5 // line status register
#define MSR 6 // modem status register
#define SCR 7 // scratch register
// interrupt enable register
#define IER_RBF 0x01 // receive buffer full
#define IER_TBE 0x02 // transmit buffer empty
#define IER_MSI 0x08 // modem status interrupt
// interrupt id register
#define IIR_PEND 0x01 // interrupt pending = 0
#define IIR_II 0x06 // determine pending interrupt
#define IIR_MSI 0x00 // modem status interrupt
#define IIR_TBE 0x02 // transmit buffer empty
#define IIR_RBF 0x04 // receive buffer full
#define IIR_RFF 0x0c // receive fifo threshold
// modem status register
#define MSR_CTS 0x10 // clear to send
#define MSR_DSR 0x20 // data set ready (modem ready)
#define MSR_DDCD 0x08 // delta carrier detect
#define MSR_CD 0x80 // carrier detected
#define FCR_16550 0xc1 // 16550 fifo enable/reset
#define LCR_DLAB 0x80 // divisor latch access bit
#define MCR_DTR 0x01 // DTR control
#define MCR_RTS 0x02 // RTS control
#define MCR_DO 0x0b // dtr, rts & out2 enabled
#define LSR_DR 0x01 // data ready
#define LSR_THRE 0x20 // transmit holding reg empty
#define LSR_TSRE 0x40 // transmit shift reg empty
// 8259 Programmable Interrupt Controller Definitions
#define I8259 0x20 // control register address
#define EOI 0x20 // end of interrupt command
#define I8259M 0x21 // mask register
#define VIDEO 0X10
#define VESAFUNCTIONID 0x4f
#define Mode640x480P24 0x112
#define Mode640x480P16 0x110
// define function keys

```

File: GLOBAL.CPP

```

#define F1          0x100 + '\x3b'          // F1
#define F2          0x100 + '\x3c'          // F2
#define F3          0x100 + '\x3d'          // F3
#define F4          0x100 + '\x3e'          // F4
#define F5          0x100 + '\x3f'          // F5
#define F6          0x100 + '\x40'          // F6
#define F7          0x100 + '\x41'          // F7
#define F10         0x100 + '\x44'          // F10
#define SWITCH(x)   ((x >> 8) | (x << 8))   // integer storage

//Globals for dct and idct

int pre  entropy[64],
        value[64],
        run_length[64];
float   lumin_table[64],                // quantization tables
        cromin_table[64],              // and pointer
        inv_lumin_table[64],           // quantization tables
        inv_cromin_table[64];          // and pointer
float   pi = 4 * atan(1),
        c1p16 = 1 / (cos(pi/16)*2),    // for dct
        c3p16 = 1 / (cos(3*pi/16)*2),
        c5p16 = 1 / (cos(5*pi/16)*2),
        c7p16 = 1 / (cos(7*pi/16)*2),  // constants
        c1p8 = 1 / (cos(pi/8)*2),
        c3p8 = 1 / (cos(3*pi/8)*2),
        c1p4 = 1 / (cos(pi/4)*2);
UINT    word_in,
        diag_index[64],
        ac_crom_huf_code[256],
        ac_crom_code_length[256],
        ac_lum_huf_code[256],
        ac_lum_code_length[256],
        dc_cro_code[16] = {0,16384,32768,49152,57344,61440,63488,
                           64512,65024,65280,65408,65472},
        dc_cro_index_length[10] = {0,3,4,5,6,7,8,9,10,11},
        dc_cro_code_length[12] = {2,2,2,3,4,5,6,7,8,9,10,11},
        dc_lum_code[16] = {0,16384,24576,32768,40960,49152,57344,
                           61440,63488,64512,65024,65280},
        dc_lum_code_length[12] = {2,3,3,3,3,3,4,5,6,7,8,9},
        dc_lum_index_length[8] = {0,1,6,7,8,9,10,11},
        inv_dc_cro_code_length[12] = {2,2,2,3,4,5,6,7,8,9,10,11},
        inv_dc_lum_code_length[8] = {0,1,6,7,8,9,10,11},
        inv_ac_cro_code[256],
        inv_ac_lum_code[256],
        inv_ac_cro_code_length[16],
        inv_ac_lum_code_length[16],
        inv_ac_cro_run_length[256],
        inv_ac_lum_run_length[256],
        inv_ac_cro_value_length[256],

```

File: GLOBAL.CPP

```

inv_ac_lum_value_length[256],
un_mask[16] = {0x8000, 0xc000, 0xe000, 0xf000, 0xf800, 0xfc00,
               0xfe00, 0xff00, 0xff80, 0xffc0, 0xffe0, 0xffff,
               0xffff8, 0xffffc, 0xffffe, 0xffff };

struct picture_parameters {
    char      *parm;
    UINT      total_segments, color, mcu_reload, h_rez, v_rez,
               file_size, offset, column_start, column_end;
    } pict_parm[6]=
        {{ "High color", 15, 3, 320, 640, 480, 0xf000, 0x5000, 0, 80},
        { "High B & W", 15, 1, 320, 640, 480, 0xf000, 0x5000, 0, 80},
        { "Med color", 6, 3, 200, 320, 240, 0x9600, 0x1900, 20, 60},
        { "Med B & W", 6, 1, 200, 320, 240, 0x9600, 0x1900, 20, 60},
        { "Low color", 1, 3, 300, 160, 120, 0xe100, 0x4600, 30, 50},
        { "Low B & W", 1, 1, 300, 160, 120, 0xe100, 0x4600, 30, 50}};

// Globals for communications routines

UINT port, base, irq, line_speed, line, i_start, i_stop, o_stop,
i_size, o_size, i_count, o_count;

UCHAR *c, lsr, io_f = 0, msr_changed, int_msr, int_lsr,
fif, flow, o_flow, i_flow, empty_trans;

UCHAR far
    *i_buf, *i_get, *i_put, *i_limit, *i_last, *o_get, *o_put, *o_buf, *o_limit;

char    *INITIAL_MODEM = "ATZ\r";

enum port_type {no_port, std_uart, fifo_uart};

struct {
    int      port;                // port base address
    char      irq;                // interrupt number
    port_type available;          // port available flag
    char      *name;              // port name
    CommPorts[4] = { { 0x3f8, 4, no_port, "COM1:" }, { 0x2f8, 3, no_port, "COM2:" },
                    { 0x3e8, 4, no_port, "COM3:" }, { 0x2e8, 3, no_port, "COM4:" }, };

    char      *speed, *parity, *data, *stop;

// Globals for video routines

union REGS regs;
UCHAR far    *picture_data;
UINT VESAmode;

// Globals for windows

```

File: GLOBAL.CPP

```

char    OldWindowData[5200],
        MenuForegroundColor = BLUE,
        MenuBackgroundColor = WHITE,
        StatusForegroundColor = WHITE,
        StatusBackgroundColor = BLUE,
        SetupForegroundColor= WHITE,
        SetupBackgroundColor= GREEN,
        TerminalForegroundColor = WHITE,
        TerminalBackgroundColor = BLUE;
int      InitComplete;

        // Globals for Menu

int pict = 5;
char      stat_format[] = " IMAGE RESOLUTION: " " %11s\x3" "%s %5s %s%1.1s%s%s %s %s \x3" "
        F10 = QUIT",
        *Resolution[6] = {{ "HIGH COLOR"}, {"HIGH B&W"}, {"MED COLOR"},
        {"MED B&W"}, {"LOW COLOR"}, {"LOW B&W"}},
        menuline1[] = "F1 \x3 F2 \x3F3\x3F4\x3F5\x3F6\x3 F7",
        menuline2[] = " SET COMM\x3SNAP\x3 SEND \x3RECEIVE \x3CLEAR SCREEN\x3
        PICTURE TYPE \x3 HANG UP ";

        // Globals for Setup.cpp

char      *SetPort = "Set communications port (1), (2), (3), (4) ? ",
        *DataType = "Set number of data bits (7) or (8) ? ",
        *ParityType = "Set parity (1)none, (2)even, (3)odd ? ",
        *SpeedType = "Set speed (1)2400, (2)4800, (3)9600, (4)19200 (5)38400 ?",
        *StopType = "Set stop bits (1) or (2) ?",
        *Notice = "Escape for default values.",
        *Defaults = " Comport 2, irq 3, 38400 bps, 8N1 (8 data, No parity, 1 stop)";

        // Globals for Initial

int        max_lines;
char      *DefaultValues " Comport 2, irq 3, 38400 bps, 8N1 (8 data, No parity, 1 stop)";

        // Globals for end

char      BadScreenWidth[] = "Screen must be 80 characters wide.",
        Done[] = "Program termination normal.",
        Mono[] = "Must have a color monitor",
        Notables[] = "Configuration tables not found",
        NoMemory[] = "Insufficient memory for allocation.",
        NoVESA[] = "Video card does not support required video modes.";

        // Global prototypes

void      CreateTables(void),

```

File: GLOBAL.CPP

```

CheckTypePort(void),
ClearInputBuffer(void),
ClearOutputBuffer(void),
Compress(void),
Dct(float *),
DeInstallModemInt(void),
End(char *),
Expand(void),
Idct(float *),
Initailize(void),
InstallModemInt(void),
Menu(void),
put_pixel_110(int, int, UCHAR, UCHAR, UCHAR),
put_pixel_112(int, int, UCHAR, UCHAR, UCHAR),
Queue(int),
QuerySetUp (void),
RequestToSend(int f),
SetBuffers (void),
SetSpeed (void),
SetInterrupt (void),
SetTextMode(void),
StatusLine(void),
Take(void),
UartIntRoutine(void),
Write(int c),
Write(char *s),
int    GetKey(void),
        SetVESAVideoMode (unsigned int);
UINT   Deque(void),
        Read(char *c),
        Word_in(unsigned int &word_in);
void interrupt CommInt(...);
void interrupt ControlBreak(...);
void interrupt CriticalRoutine(...);
void interrupt far (*OldComm)(...);
void interrupt far (*OldControlBreak)(...);

```

File: INTERPT.CPP

```

void    UartIntRoutine(void) {                                // modem interrupt routine
    int    c, cnt, first_cycle;
    char    iir;
    while (((iir = inportb(base + IIR)) & IIR_PEND) == 0) {    // interrupt is pending?
        switch (iir & IIR_IL) {                                  // switch on interrupt type
            case IIR_MSI:                                        // if CTS
                msr_changed = 1;                                // flag msr changed
                if ((int_msr = inportb(base + MSR)) & MSR_CTS) { // CTS interrupt?
                    if (o_flow || inportb(base + LSR) & LSR_THRE) { // and flow controlled?
                        o_flow = 0;                                // clear flow controll flag
                        if ((c = Dequeue()) == -1) empty_trans = 1; // if available
                        else outportb(base + THR, c);              // output character
                    }
                }
                if (int_msr & MSR_DDCD) {                        // if carrier lost
                    delay(500);                                  // for more than 1/2 second
                    if ((inportb(base + MSR) & MSR_CD) == 0) {   // echo EOI command
                        Write(0xff);
                        Write(0xd9);
                        StatusLine();
                    }
                }
            }
            continue;

            case IIR_TBE:                                        // transmit buffer empty
                cnt = fifo ? 15 : 1;                             // FIFO accepts 15 characters
                for (; cnt--;) {                                  // loop till buffer full
                    if (NOT (inportb(base + MSR) & MSR_CTS)){    // if not clear to send
                        o_flow = 1;                                // show flow controlled
                        break;                                     // exit loop
                    }
                    if ((c = Dequeue()) == -1) {                 // if not char available
                        empty_trans = 1;                           // set empty trans flag
                        break;                                     // exit loop
                    }
                    outportb(base + THR, c);                     // otherwise output characters
                }
            }
            continue;

            case IIR_RBF:                                        // receive buffer full
                while ((inportb(base + LSR)) & LSR_DR)          // read data if available
                    Queue(inportb(base + RBR));
                continue;
        }
    }
    __asm mov    al, EOI                                        // load EOI to register al
    __asm out I8259, al                                         // send EOI to int controller
}

```

File: INTERPT.CPP

```

void      InstallModemInt(void) {
    int mask;                // interrupt mask
    OldComm = _dos_getvect(irq + 8); // save current interrupt vector
    _dos_setvect(irq + 8, CommInt); // set new interrupt vector
    mask = inportb(I8259M);      // get interrupt settings
    mask &= ~(1 << irq);        // activate modem interrupt
    outportb(I8259M, mask);     // set new interrupt
}

void      DeInstallModemInt(void){
    int mask;                // interrupt mask
    outportb(base + IER, 0);  // disable UART interrupt
    outportb(base + FCR, 0);  // disable FIFO
    _dos_setvect(irq + 8, OldComm); // set to old interrupt vector
    mask = inportb(I8259M);    // retrieve interrupt values
    mask |= (1 << irq);       // remove modem interrupt
    outportb(I8259M, mask);    // reset to old interrupt
}

void      interrupt CommInt(...)
{
    __asm sti                // re-enable interrupts
    UartIntRoutine();        // use object's interrupt rtn
}

#pragma option -O2-b-e      // no global reg allocation
                          // ..or dead code elimination

void      interrupt CriticalErrorHandler(...) {
    if (_AX & 0x800) _AX = (_AX & 0xff00) | 3; // when fail, abort, retry
    else _AX = (_AX & 0xff00) | 2;             // show failed
}                                              // else abort

```

File: SETUP.CPP

```

void QuerySetup(void) {
    unsigned char choice;
    unsigned int i, ch;
    gettext(1,3,80,24,OldWindowData);           // save old window data
    window(5,7,75,22);                           // draw setup window
    _wscroll = 0;                                // disable scroll
    textcolor(SetupForegroundColor);             // set colors
    textbackground(SetupBackgroundColor);
    clrscr();                                     // clear area
    gotoxy(1,1);                                 // goto first position
    cprintf("%c", '\xda');                       // draw block borders
    for (i = 2; i < 71; i++) cprintf("%c", '\xc4');
    gotoxy(1,71);
    cprintf("%c", '\xbf');
    gotoxy(1,16);
    cprintf("%c", '\xc0');
    for (i = 2; i < 71; i++) cprintf("%c", '\xc4');
    gotoxy(71,16);
    cprintf("%c", '\xd9');
    for (i = 2; i < 16; i++){
        gotoxy(1, i);
        cprintf("%c", '\xb3');
        gotoxy(71, i);
        cprintf("%c", '\xb3');
    }
    gotoxy(3,14);                               // print notice
    cprintf( "%s", Notice);
    gotoxy(3,15);                               // print default values
    cprintf("%s", Defaults);
    _setcursortype(_NOCURS);                   // remove cursor
    while (1){                                  // set port address
        gotoxy(5,3);
        cprintf("%s", SetPort);
        choice = GetKey();
        ch = choice - 48;
        if (choice != 27) cprintf("%c", choice);
        else cprintf("%i",2);
        if ((ch < 5) && (ch > 0)) {
            base = CommPorts[ch - 1].port;
            irq = CommPorts[ch - 1].irq; port = ch - 1;
            port = ch - 1;
            break;
        }
        if (choice == 27){                      // default values
            base = CommPorts[1].port;           //comm port 2
            irq = CommPorts[1].irq;             // interrupt 3
            port = 1;                           // pointer to commports[]
            break; }
        if (choice != 0) cprintf("%c",0x07);    // beep on bad input
    }
}

```

File: SETUP.CPP

```

while(1){
    gotoxy(5,4);
    cprintf("%s", "Using Interrupt (3) - (5)? ");
    choice = GetKey();
    if (choice != 27) cprintf("%c", choice);
    else cprintf("%i", CommPorts[port].irq);
    ch = choice - 48;
    if (choice == 27) break;
    if ( (ch < 6) && (ch > 2)) {
        irq = ch;
        break;
    }
    if (choice != 0) cprintf("%c", 0x07);
}
while (1){
    gotoxy(5,5);
    cprintf("%s", SpeedType);
    choice = GetKey();
    ch = choice - 48;
    if (choice != 27) cprintf("%c", choice);
    else cprintf("%i", 5);
    if (ch == 1){line_speed = 48; speed = "2400"; break; }
    if (ch == 2){line_speed = 24; speed = "4800"; break; }
    if (ch == 3){line_speed = 12; speed = "9600"; break; }
    if (ch == 4){line_speed = 6; speed = "19200"; break; }
    if (ch == 5){line_speed = 3; speed = "38400"; break; }
    if (choice == 27) {line_speed = 3; speed = "38400"; break; }
    if (choice != 0) cprintf("%c", 0x07);
}
CheckTypePort();
SetSpeed();
SetInterrupt();
while (1){
    gotoxy(5,6);
    cprintf("%s", DataType);
    choice = GetKey();
    ch = choice - 48;
    if (choice != 27) cprintf("%c", choice);
    else cprintf("%i", 8);
    if (ch == 7) {line = 2; data = "7"; break; }
    if (ch == 8) {line = 3; data = "8"; break; }
    if (choice == 27) {line = 3; data = "8"; break; }
    if (choice != 0) cprintf("%c", 0x07);
}
while (1){
    gotoxy(5,7);
    cprintf("%s", ParityType);
    choice = GetKey();
    ch = choice - 48;
    if (choice != 27) cprintf("%c", choice);

```

File: SETUP.CPP

```

else cprintf("%i", 1),
if (ch == 1){line |= 0; parity = "N"; break; }
if (ch == 2){line |= 0x18; parity = "E"; break;}
if (ch == 3){line |= 0x08; parity = "O"; break;}
if (choice == 27) {parity = "N"; break; }
if (choice != 0) cprintf("%c", 0x07);
}
while (1){
    gotoxy(5,8);
    cprintf("%s", StopType);
    choice = GetKey();
    ch = choice - 48;
    if (choice != 27) cprintf("%c", choice);
    else cprintf("%i", 1);
    if(ch == 1) {stop = "1"; break;}
    if (ch == 2){line |= 4; stop = "2"; break; }
    if (choice == 27) {stop = "1";break; }
    if (choice !=0) {cprintf("%c", 0x07); }
}
delay(1000);
outportb(base, line & ~LCR_DLAB);
delay(10);
window(1,3,80,23);
puttext(1,3,80,24,OldWindowData);
StatusLine();
textcolor(TerminalForegroundColor);
textbackground(TerminalBackgroundColor);
_setcursortype(_NORMALCURSOR);
_wscroll = 1;
}

void CheckTypePort(void){
    outportb(base + SCR, 0x55);
    if (inportb(base + SCR) != 0x55)
        fifo = 0;
    else {
        outportb(base + FCR, 0xcf);
        if ((inportb(base + IIR) & 0xc0) != 0xc0)
            fifo = 0;
        else {
            outportb(base + FCR, 0);
            fifo = 1;
        }
    }
}
}

```

File: SETUP.CPP

```

void SetSpeed(void) {
    // check for port types
    // line_speed holds divisor
    outputb(base + LCR, inportb(base + LCR) | LCR_DLAB); // set divisor latch
    // output divisor
    outputb(base + IER, line_speed >> 8); // most significant bit
    outputb(base + THR, line_speed & 0xff); // and least significant bit
    outputb(base + LCR, inportb(base + LCR) & ~LCR_DLAB); // reset divisor latch
}

void SetInterrupt(void) {
    // set interrupts
    outputb(base + MCR, MCR_DO); // set modem to acknowledge DTR, RTS
    // and asynchronous adapter interface
    outputb(base + IER, 0); // disable modem interrupt
    if (fifo) outputb(base + FCR, FCR_16550); // set fifo features
    inportb(base + MSR); // clear modem status interrupt
    inportb(base + RBR); // clear receive buffer
    inportb(base + IIR); // show no interrupts
    outputb(base + IER, IER_RBF | IER_TBE | IER_MSI); // set modem interrupt
}

void SetBuffers(void) {
    // output buffer size
    o_size = 30000; // input buffer size
    i_size = 30000; // assign input buffer
    i_buf = (unsigned char far *) farmalloc(30000L); // quit if no memory
    if (i_buf == NULL) End(NoMemory); // zero memory locations
    memset(i_buf, 0, i_size); // set size limit
    i_limit = i_buf + i_size - 1; // set get pointer
    i_get = i_buf; // set retrieve pointer
    i_put = i_buf; // assign output buffer
    o_buf = (unsigned char far *) farmalloc(30000L); // quit if no memory
    if (o_buf == NULL) End(NoMemory); // zero memory locations
    memset(o_buf, 0, o_size); // set end limit
    o_limit = o_buf + o_size - 1; // set pointers
    o_get = o_buf; // set pointers
    o_put = o_buf;
    // set FIFO buffer controls
    i_stop = i_size-100; // set stop limits
    i_start = i_stop/2; // and start limits
    o_stop = o_size-100;
    o_flow = i_flow = 0; // remove flow control
    o_count = i_count = 0; // zero buffer counts
    // set flags used to signal output complete and to update modem status display
    empty_trans = 1; // flag empty transmitter
    msr_changed = 1; // and msr flag
}

```

File: COMM.CPP

```

unsigned int Word_in(unsigned int &word_in) { // retrieve code word
    unsigned char    c;
    unsigned int     test = 0, byte[2] = {0,0}; // set marker flag and clear output word
    int count = 1; // set word count
    while (count >=0){ // while output word has room
        loop: if(Read(&c) == 0xffff) goto loop; // read until character ready
        if (test == 0){ // if marker not found
            if (c== 0xff) test = 1; // if marker set marker flag
            else {
                byte[count] = c; // else keep character
                count--; // and increase count
            }
        }
        else {
            switch(c){ // switch based on character
                case (0xff):break; // another marker
                case (0x00):{ // remove pad and continue
                    test = 0; // reset flag
                    byte[count] = 0xff; // add 0xff byte
                    count--; // and increase count
                    break;
                }
                case (0xd9):{ // end of image marker
                    word_in = byte[0] | (byte[1]<< 8); // load word_in
                    return(0xffd9); // return with marker
                }
            }
        }
    }
    word_in = byte[0] | (byte[1]<< 8); // load word_in
    return(0); // return with no marker
}

void Queue(int c) { // place character in output queue
    if ((i_count >= i_stop)&& NOT i_flow) // flow control needed and not on
        RequestToSend(0);
    if ((i_size - i_count) != 0){ // room in queue
        i_count++; // increase character count
        *i_put++ = c; // add character to queue
        if (i_put == i_limit) i_put = i_buf; // if end of buffer, loop
    }
}

unsigned int Deque(void) { //get character from output queue
    unsigned char c; // character to be removed
    if (o_count == 0) return(-1); // if empty return empty
    c = *o_get++; // otherwise get char from queue
    o_count--; // reduce queue count
    if (o_get == o_limit) o_get = o_buf; // if end of buffer, loop
    return(c & 0xff); // return char
}

```

File: COMM.CPP

```

}

void Write(int c) {
    if (o_count == o_stop) delay(1000);
    *o_put++ = c;
    o_count++;
    if (o_put == o_limit) o_put = o_buf;
    if (empty_trans) {
        empty_trans = 0;
        if ((inportb(base + MSR) & MSR_CTS)
            && (c = Deque()) != -1)
            outportb(base + THR, c);
    }
}

void Write(char *s) {
    for (; *s;) Write(*s++);
}

unsigned int Read(char *c) {
    if (i_count == 0) return(0xffff);
    *c = *i_get++;
    i_count--;
    if (i_get == i_limit) i_get = i_buf;
    if (i_flow && (i_count <= i_start))
        RequestToSend(1);
    return(0);
}

void RequestToSend(int f) {
    outportb(base + MCR, MCR_DO -
              (f ? 0 : MCR_RTS));
    i_flow = NOT f;
    return;
}

void ClearInputBuffer(void) {
    _asm cli
    i_get = i_put = i_buf;
    i_count = 0;
    _asm sti
}

void ClearOutputBuffer(void) {
    _asm cli
    o_get = o_put = o_buf;
    o_count = 0;
    _asm sti
}

```

File: DCT.CPP

```

void dct(float *array) {
    int j, j8;
    float a[8], b[8], c[8];
    for(j=0; j<8; j++) {          //open loop j, row counter
        j8 = j*8;
        *(a)   = *(array + j8)    + *(array + j8 + 7);    //permuted input butterflies
        *(a+1) = *(array + j8 + 1) + *(array + j8 + 6);
        *(a+2) = *(array + j8 + 3) + *(array + j8 + 4);
        *(a+3) = *(array + j8 + 2) + *(array + j8 + 5);
        *(a+4) = *(array + j8)    - *(array + j8 + 7);
        *(a+5) = *(array + j8 + 1) - *(array + j8 + 6);
        *(a+6) = *(array + j8 + 3) - *(array + j8 + 4);
        *(a+7) = *(array + j8 + 2) - *(array + j8 + 5);
        *(a+4) *= c1p16;          //fast factor multiplications
        *(a+5) *= c3p16;
        *(a+6) *= c7p16;
        *(a+7) *= c5p16;
        *(b)   = *(a)    + *(a+2);    //slow butterflies
        *(b+1) = *(a+1)  + *(a+3);
        *(b+2) = *(a)    - *(a+2);
        *(b+3) = *(a+1)  - *(a+3);
        *(b+4) = *(a+4)  + *(a+6);
        *(b+5) = *(a+5)  + *(a+7);
        *(b+6) = *(a+4)  - *(a+6);
        *(b+7) = *(a+5)  - *(a+7);
        *(b+2) *= c1p8;          //fast factor multiplications
        *(b+3) *= c3p8;
        *(b+6) *= c1p8;
        *(b+7) *= c3p8;
        *(c)   = *(b)    + *(b+1);    //slow butterflies
        *(c+1) = *(b)    - *(b+1);
        *(c+2) = *(b+2)  + *(b+3);
        *(c+3) = *(b+2)  - *(b+3);
        *(c+4) = *(b+4)  + *(b+5);
        *(c+5) = *(b+4)  - *(b+5);
        *(c+6) = *(b+6)  + *(b+7);
        *(c+7) = *(b+6)  - *(b+7);
        *(c+1) *= c1p4;          // fast factor multiplication code
        *(c+3) *= c1p4;
        *(c+5) *= c1p4;
        *(c+7) *= c1p4;
        *(c+2) += *(c+3);          // last butterflies faster executing code
        *(c+6) += *(c+7);          // usable since original value is not reused
        *(c+4) += *(c+6);
        *(c+6) += *(c+5);
        *(c+5) += *(c+7);
        *(array + j8)   = *(c) ;      //store output values
        *(array + j8 + 4) = *(c+1) ;
        *(array + j8 + 2) = *(c+2) ;
        *(array + j8 + 6) = *(c+3) ;
    }
}

```

File: DCT.CPP

```

*(array + j8 + 1) = *(c+4) ;
*(array + j8 + 5) = *(c+5) ;
*(array + j8 + 3) = *(c+6) ;
*(array + j8 + 7) = *(c+7) ;
}
for(j=0;j<8;j++) {          //SECOND DIMENSION open loop j, column counter
    *(a)    = *(array + j)    + *(array + 56 + j) ;          //perform butterflies
    *(a + 1) = *(array + j + 8) + *(array + 48 + j);          //with permuted input order
    *(a + 2) = *(array + j + 24) + *(array + 32 + j);
    *(a + 3) = *(array + j + 16) + *(array + 40 + j);
    *(a + 4) = *(array + j)    - *(array + 56 + j);
    *(a + 5) = *(array + j + 8) - *(array + 48 + j);
    *(a + 6) = *(array + j + 24) - *(array + 32 + j);
    *(a + 7) = *(array + j + 16) - *(array + 40 + j);
    *(a + 4) *= c1p16;                                          //fast factor multiplications
    *(a + 5) *= c3p16;
    *(a + 6) *= c7p16;
    *(a + 7) *= c5p16;
    *(b)    = *(a)    + *(a + 2);                              //slow butterflies
    *(b + 1) = *(a + 1) + *(a + 3);
    *(b + 2) = *(a)    - *(a + 2);
    *(b + 3) = *(a + 1) - *(a + 3);
    *(b + 4) = *(a + 4) + *(a + 6);
    *(b + 5) = *(a + 5) + *(a + 7);
    *(b + 6) = *(a + 4) - *(a + 6);
    *(b + 7) = *(a + 5) - *(a + 7);
    *(b + 2) *= c1p8;                                          // fast factor multiplications
    *(b + 3) *= c3p8;
    *(b + 6) *= c1p8;
    *(b + 7) *= c3p8;
    *(c)    = *(b)    + *(b + 1);                              //slow butterflies
    *(c + 1) = *(b)    - *(b + 1);
    *(c + 2) = *(b + 2) + *(b + 3);
    *(c + 3) = *(b + 2) - *(b + 3);
    *(c + 4) = *(b + 4) + *(b + 5);
    *(c + 5) = *(b + 4) - *(b + 5);
    *(c + 6) = *(b + 6) + *(b + 7);
    *(c + 7) = *(b + 6) - *(b + 7);
    *(c + 1) *= c1p4;                                          // fast multiplication code
    *(c + 3) *= c1p4;
    *(c + 5) *= c1p4;
    *(c + 7) *= c1p4;
    *(c + 2) += *(c + 3);
    *(c + 6) += *(c + 7);
    *(c + 4) += *(c + 6);
    *(c + 6) += *(c + 5);
    *(c + 5) += *(c + 7);
    *(array + j)    = *(c) ;
    *(array + j + 32) = *(c + 1);
    *(array + j + 16) = *(c + 2);

```

File: DCT.CPP

```

        *(array + j + 48) = *(c + 3);
        *(array + j + 8)  = *(c + 4);
        *(array + j + 40) = *(c + 5);
        *(array + j + 24) = *(c + 6);
        *(array + j + 56) = *(c + 7);
    }
    return;
}

void idct(unsigned temp, float *a_p){
    int i, i8;
    float d[8], c[8], b[8];
    for (i=0; i<temp;i++){
        i8 = i * 8;                                // first dimension
        *(d)      = *(a_p + i8) ;
        *(d + 1)  = *(a_p + 4 + i8) ;
        *(d + 2)  = *(a_p + 2 + i8) ;
        *(d + 3)  = *(a_p + 6 + i8) ;
        *(d + 4)  = *(a_p + 1 + i8) ;
        *(d + 5)  = *(a_p + 5 + i8) ;
        *(d + 6)  = *(a_p + 3 + i8) ;
        *(d + 7)  = *(a_p + 7 + i8) ;

        *(d + 7) += *(d + 5);                        //Fast Butterflies
        *(d + 5) += *(d + 6);
        *(d + 6) += *(d + 4);
        *(d + 3) += *(d + 2);
        *(d + 7) += *(d + 6);

        *(d + 1) *= c1p4;                            //multiplications
        *(d + 3) *= c1p4;
        *(d + 5) *= c1p4;
        *(d + 7) *= c1p4;

        *(c)      = *(d)      + *(d + 1);            // butterflies
        *(c + 1)  = *(d)      - *(d + 1);
        *(c + 2)  = *(d + 2)  + *(d + 3);
        *(c + 3)  = *(d + 2)  - *(d + 3);
        *(c + 4)  = *(d + 4)  + *(d + 5);
        *(c + 5)  = *(d + 4)  - *(d + 5);
        *(c + 6)  = *(d + 6)  + *(d + 7);
        *(c + 7)  = *(d + 6)  - *(d + 7);

        *(c + 2) *= c1p8;                            //multiplications
        *(c + 3) *= c3p8;
        *(c + 6) *= c1p8;
        *(c + 7) *= c3p8;

        *(b)      = *(c)      + *(c + 2);            // butterflies
        *(b + 1)  = *(c + 1)  + *(c + 3);
        *(b + 2)  = *(c + 0)  - *(c + 2);
        *(b + 3)  = *(c + 1)  - *(c + 3);
        *(b + 4)  = *(c + 4)  + *(c + 6);
        *(b + 5)  = *(c + 5)  + *(c + 7);
        *(b + 6)  = *(c + 4)  - *(c + 6);
    }
}

```

File: DCT.CPP

```

*(b + 7) = *(c + 5) - *(c + 7);
*(b + 4) *= c1p16; //multiplications
*(b + 5) *= c3p16;
*(b + 6) *= c7p16;
*(b + 7) *= c5p16;
*(a_p + 0 + i8) = *(b) + *(b + 4);
*(a_p + 1 + i8) = *(b + 1) + *(b + 5);
*(a_p + 3 + i8) = *(b + 2) + *(b + 6);
*(a_p + 2 + i8) = *(b + 3) + *(b + 7);
*(a_p + 7 + i8) = *(b) - *(b + 4);
*(a_p + 6 + i8) = *(b + 1) - *(b + 5);
*(a_p + 4 + i8) = *(b + 2) - *(b + 6);
*(a_p + 5 + i8) = *(b + 3) - *(b + 7);
}
for (i=0; i<8;i++) { //second dimension
*(d) = *(a_p + i);
*(d + 1) = *(a_p + 32 + i);
*(d + 2) = *(a_p + 16 + i);
*(d + 3) = *(a_p + 48 + i);
*(d + 4) = *(a_p + 8 + i);
*(d + 5) = *(a_p + 40 + i);
*(d + 6) = *(a_p + 24 + i);
*(d + 7) = *(a_p + 56 + i);
*(d + 7) += *(d + 5); //Fast Butterflies
*(d + 5) += *(d + 6);
*(d + 6) += *(d + 4);
*(d + 3) += *(d + 2);
*(d + 7) += *(d + 6);
*(d + 1) *= c1p4; // multiplications
*(d + 3) *= c1p4;
*(d + 5) *= c1p4;
*(d + 7) *= c1p4;
*(c) = *(d) + *(d + 1); // butterflies
*(c + 1) = *(d) - *(d + 1);
*(c + 2) = *(d + 2) + *(d + 3);
*(c + 3) = *(d + 2) - *(d + 3);
*(c + 4) = *(d + 4) + *(d + 5);
*(c + 5) = *(d + 4) - *(d + 5);
*(c + 6) = *(d + 6) + *(d + 7);
*(c + 7) = *(d + 6) - *(d + 7);
*(c + 2) *= c1p8; // multiplications
*(c + 3) *= c3p8;
*(c + 6) *= c1p8;
*(c + 7) *= c3p8;
*(b) = *(c) + *(c + 2); // butterflies
*(b + 1) = *(c + 1) + *(c + 3);
*(b + 2) = *(c + 0) - *(c + 2);
*(b + 3) = *(c + 1) - *(c + 3);
*(b + 4) = *(c + 4) + *(c + 6);
*(b + 5) = *(c + 5) + *(c + 7);

```

File: DCT.CPP

```

*(b + 6) = *(c + 4) - *(c + 6);
*(b + 7) = *(c + 5) - *(c + 7);
*(b + 4) *= c1p16;                                     // multiplications
*(b + 5) *= c3p16;
*(b + 6) *= c7p16;
*(b + 7) *= c5p16;
*(a_p + 0 + i) = *(b + 0) + *(b + 4);
*(a_p + 8 + i) = *(b + 1) + *(b + 5);
*(a_p + 24 + i) = *(b + 2) + *(b + 6);
*(a_p + 16 + i) = *(b + 3) + *(b + 7);
*(a_p + 56 + i) = *(b + 0) - *(b + 4);
*(a_p + 48 + i) = *(b + 1) - *(b + 5);
*(a_p + 32 + i) = *(b + 2) - *(b + 6);
*(a_p + 40 + i) = *(b + 3) - *(b + 7);
}
}

```

File : COMPRESS.CPP

```

void Compress(void)
{
float redf,greenf,bluef,ArrayY[64],ArrayI[64],ArrayQ[64],*array_pointer,*quant_table;
int LastDC[3] = {0,0,0};

unsigned int PicturePointer, x_offset, y_offset, color, start, i8, j3, j, i, index, mcu_count,
current_word, block_row, count, segment, actual_row, pointer, temp, temp1,
bits_empty, bits_filled, code_count, code_length,
*dc_code, *dc_code_length, *ac_code, *ac_code_length,
// negates unsigned values
negate[16] = {{0x0},{0x1},{0x3},{0x7},{0xf},{0x1f},{0x3f},{0x7f},{0xff},
{0x1ff},{0x3ff},{0x7ff},{0xff},{0x1ff},{0x3ff},{0x7ff}};

unsigned char red, green, blue;
FILE *file_ptr;
SetVESA VideoMode(VESAmode); // set video mode
Write(0xff); // send marker;
Write(0xd8); // send SOI;
Write(0xff); // send marker;
Write(0xe0); // send application segment;
Write(0x00); // send integer length of segment
Write(0x04); // send length of application segment
x_offset = 0; // high rez offset
y_offset = 0;
BW = 0; // set color

switch (pict) { // set application segment, display location
case (0):{
Write(0xcc); // high resolution
Write(0xf0); // color
file_ptr = fopen("high.rez","rb");
break;
}

case (1):{
Write(0xcc); // high resolution
Write(0xf0); // b & w
BW = 1;
file_ptr = fopen("high.rez","rb");
break;
}

case (2):{
Write(0x66); // medium resolution
Write(0xf0); // color
x_offset = 160;
y_offset = 120;
file_ptr = fopen("medium.rez","rb");
break;
}

case (3):{

```

File : COMPRESS.CPP

```

        Write(0x66);                                // medium resolution
        Write(0x0f);                                // b & w
        BW = 1;
        x_offset = 160;
        y_offset = 120;
        file_ptr = fopen("medium.rez", "rb");
        break;
    }
    case (4):{
        Write(0x33);                                // low resolution
        Write(0xf0);                                // color
        x_offset = 240;
        y_offset = 180;
        file_ptr = fopen("low.rez", "rb");
        break;
    }
    case (5):{
        Write(0x33);                                // low resolution
        Write(0x0f);                                // b & w
        BW = 1;
        x_offset = 240;
        y_offset = 180;
        file_ptr = fopen("low.rez", "rb");
        break;
    }
}

index = 0; bits_empty = 16; bits_filled = 0; current_word = 0; PicturePointer = 0;
value[1] = 0; run_length[1] = 0; block_row = 0; actual_row = 0;
for (segment = 0; segment < pict_parm[pict].total_segments;
    segment++){
    temp = fread(picture_data, sizeof(unsigned char),
        pict_parm[pict].file_size, file_ptr);
    if(PicturePointer == pict_parm[pict].h_rez * 3){
        actual_row += 8;
        PicturePointer = 0;
    }
    block_row = 0;
    for (mcu_count = 0; mcu_count < pict_parm[pict].mcu_reload; mcu_count++){
        if(PicturePointer == pict_parm[pict].h_rez * 3){
            actual_row += 8;
            block_row += pict_parm[pict].h_rez * 24;
            PicturePointer = 0;
        }
        for (i=0; i<8; i++){
            start = i*pict_parm[pict].h_rez * 3 + block_row + PicturePointer;
            i8 = i * 8;
            for(j=0; j<8; j++){
                j3 = j * 3 + start;
                red = *( picture_data + j3);
            }
        }
    }
}

```

File : COMPRESS.CPP

```

green = *( picture_data + j3 + 1);
blue = *( picture_data + j3 + 2);
bluef = blue - 128;
greenf = green - 128;
redf = red - 128;
ArrayY[i8 + j] = .299 * redf + .587 * greenf + .114 * bluef;
ArrayI[i8 + j] = .596 * redf - .275 * greenf - .321 * bluef;
ArrayQ[i8 + j] = .212 * redf - .533 * greenf + .311 * bluef;
if (VESAmode==0x112) // video mode 70h
    put_pixel_112((PicturePointer/3) + j + x_offset,
        actual_row + i + y_offset, red,green,blue);
else if (VESAmode==0x110) // video mode 68h
    put_pixel_110((PicturePointer/3) + j + x_offset,
        actual_row + i + y_offset, red,green,blue);
    }
}
color = 0;
PicturePointer += 24;
while (color < pict_parm[pict].color){
    switch (color){ // set proper code pointer
        case(0):{
            array_pointer = ArrayY;
            quant_table = lumin_table;
            ac_code = ac_lum_huf_code;
            ac_code_length = ac_lum_code_length;
            dc_code = dc_lum_code;
            dc_code_length = dc_lum_code_length;
            break;
        }
        case(1):{
            array_pointer = ArrayI;
            quant_table = cromin_table;
            ac_code = ac_crom_huf_code;
            ac_code_length = ac_crom_code_length;
            dc_code = dc_cro_code;
            dc_code_length = dc_cro_code_length;
            break;
        }
        case(2):{
            array_pointer = ArrayQ;
            break;
        }
    }
}
dct(array_pointer);
for (i=0;i<64;i++){
    if (array_pointer[i] >= 0)
        pre_entropy[i] = floor((array_pointer[i] * quant_table[i] + .5);
    else pre_entropy[i] = ceil((array_pointer[i] * quant_table[i] ) - .5);
}
value[0] = pre_entropy[0] - LastDC[color]; //set dc value

```

File : COMPRESS.CPP

```

count = 1;
LastDC[color] += value[0];
color++;
i = 0;
for (j=1;j<64;j++) {
    if (pre_entropy[diag_index[j]] == 0) i++;
    else {
        while (i > 15) {
            run_length[count] = 240;
            value[count] = 0;
            count++;
            i -= 16;
        }
        i = i < 4;
        value[count] = pre_entropy[diag_index[j]];

        temp1 = 0;
        temp = abs(pre_entropy[diag_index[j]]);
        while (temp > 0) {
            temp >>= 1;
            temp1++;
        }
        run_length[count] = i | temp1;
        count++; // increment array counter
        i = 0;
    }
}
run_length[count] = 0; // append end of block character
value[count] = 0;
count++;

temp = abs(value[0]);
index = 0;
if (temp != 0) {
    while (temp > 0) {
        temp >>= 1;
        index++;
    }
}
temp = dc_code[index];
code_length = dc_code_length[index];
if (bits_empty < code_length) {
    temp1 = temp;
    temp = temp >> bits_filled;
    current_word |= temp;
    current_word = SWITCH(current_word);
    Write(current_word);
    if ((current_word << 8) == 0xff00) Write(0x00);

    current_word = SWITCH(current_word);
}

// RUNLENGTH counter
// loop counter
// If value = 0 incr. RUNLENGTH
// If current value > 0 then
// Is RUNLENGTH greater than 15?
// yes, store as run_length of 15 zeros
// with a value of zero
// point to next code entry
// reduce RUNLENGTH by 16 zeros
// repeat until RUNLENGTH < 16.
// put RUNLENGTH in upper 4 bits
// store number in 'value'
// determine SIZE = log2(|value|)
// counter for SIZE

// temp = absolute value of 'value'
// divide temp by 2
// increment SIZE
// when temp = 0 then temp1 = SIZE
// byte = RUNLENGTH/SIZE

// Get next entry
// end of loop, last entry checked

// run_length = value = 0 [EOB]
// counts number of entries in value
// create dc bit stream
// find dc code word

// find symbol
// index = SIZE

// temp holds code word.
// code_length=word length code.
// Enough space in current word?
// No. Create second codeword
// align bits
// add bits to current word
// exchange first & second bytes
// output first byte
// shift out first byte
// if byte equals marker flag, pad
// reverse byte order

```

File : COMPRESS.CPP

```

Write(current_word);
if ((current_word << 8) == 0xff00) Write(0x00);

current_word = (temp1 << bits_empty);
bits_filled = code_length - bits_empty;
bits_empty = 16 - bits_filled;
}
else {
    if (bits_empty > code_length){
        current_word |= (temp >> bits_filled);
        bits_filled += code_length;
        bits_empty -= code_length;
    }
    else {
        current_word |= (temp >> bits_filled);
        current_word = SWITCH(current_word);
        Write(current_word);
        if ((current_word << 8) == 0xff00) Write(0x00);
        current_word = SWITCH(current_word);
        Write(current_word);
        if ((current_word << 8) == 0xff00) Write(0x00);
        bits_empty = 16;
        bits_filled = 0;
        current_word = 0;
    }
}

if (value[0] != 0){
    code_length = index;
    if (value[0] < 0) temp = negate[index] + value[0];
    else temp = value[0];
}
if (bits_empty < code_length){
    temp1 = temp;
    temp = temp >> bits_filled;
    current_word |= temp;
    current_word = SWITCH(current_word);
    Write(current_word);
    if ((current_word << 8) == 0xff00) Write(0x00);

    current_word = SWITCH(current_word);
    Write(current_word);
    if ((current_word << 8) == 0xff00) Write(0x00);

    current_word = (temp1 << bits_empty);
    bits_filled = code_length - bits_empty;
    bits_empty = 16 - bits_filled;
}
else {
    if (bits_empty > code_length){
        current_word |= (temp >> bits_filled);
        bits_filled += code_length;
    }
}

```

```

// output next byte
// current word now shifted to zero
// if byte equals marker flag, pad
// place remaining bits in new word
// calculate bits empty
// calculate bits filled

// Sufficient space in current word
// Too much space?
// Yes, then align and add value
// calculate bits filled
// and bits empty

// exact amount of space
// Yes, then align and add value
// reverse byte order
// output first byte
// shift & pad, if necessary
// reverse byte order
// output second byte
// shift & pad, if necessary
// set bits empty
// set bits filled
// current_word now empty

// calculate additional bits

// Enough space in current word?
// No. Create second codeword
// align bits
// add bits to current word
// exchange first & second bytes
// output first byte
// shift out first byte
// if byte equals marker flag, pad
// reverse byte order
// output next byte
// current word now shifted to zero
// if byte equals marker flag, pad
// place remaining bits in new word
// calculate bits empty
// calculate bits filled

// Sufficient space in current word
// Too much space?
// Yes, then align and add value
// calculate bits filled

```

File : COMPRESS.CPP

```

    bits_empty -= code_length;                                // and bits empty
}
else {                                                        // exact amount of space
    current_word |= (temp >> bits_filled);                  // Yes, then align and add value
    current_word = SWITCH(current_word);                    // reverse byte order
    Write(current_word); // output first byte
    if ((current_word << 8) == 0xff00) Write(0x00);          // shift & pad, if necessary
    current_word = SWITCH(current_word);                    // reverse byte order
    Write(current_word); // output second byte
    if ((current_word << 8) == 0xff00) Write(0x00);          // shift & pad, if necessary
    bits_empty = 16;                                         // set bits empty
    bits_filled = 0;                                         // set bits filled
    current_word = 0;                                        // current_word now empty
}
}
}

for(code_count = 1; code_count < count; code_count++){ // create ac bit stream
    temp = ac_code[run_length[code_count]];                // get ac_code
    code_length = ac_code_length[run_length[code_count]]; // get ac_code length
if (bits_empty < code_length){ // Enough space in current word?
    temp1 = temp; // No. Create second codeword
    temp = temp >> bits_filled; // align bits
    current_word |= temp; // add bits to current word
    current_word = SWITCH(current_word); // exchange first & second bytes
    Write(current_word); // output first byte
    if ((current_word << 8) == 0xff00) Write(0x00); // shift out first byte
    // if byte equals marker flag, pad
    current_word = SWITCH(current_word); // reverse byte order
    Write(current_word); // output next byte
    if ((current_word << 8) == 0xff00) Write(0x00); // current word now shifted to zero
    // if byte equals marker flag, pad
    current_word = (temp1 << bits_empty); // place remaining bits in new word
    bits_filled = code_length - bits_empty; // calculate bits empty
    bits_empty = 16 - bits_filled; // calculate bits filled
}
else { // Sufficient space in current word
    if (bits_empty > code_length){ // Too much space?
        current_word |= (temp >> bits_filled); // Yes, then align and add value
        bits_filled += code_length; // calculate bits filled
        bits_empty -= code_length; // and bits empty
    }
    else { // exact amount of space
        current_word |= (temp >> bits_filled); // Yes, then align and add value
        current_word = SWITCH(current_word); // reverse byte order
        Write(current_word); // output first byte
        if ((current_word << 8) == 0xff00) Write(0x00); // shift & pad, if necessary
        current_word = SWITCH(current_word); // reverse byte order
        Write(current_word); // output second byte
        if ((current_word << 8) == 0xff00) Write(0x00); // shift & pad, if necessary
        bits_empty = 16; // set bits empty
    }
}
}

```

File : COMPRESS.CPP

```

bits_filled = 0; // set bits filled
current_word = 0; // current_word now empty
}
}

// add additional bits
if(value[code_count] == 0 && run_length[code_count] == 0) break;
if (value[code_count] != 0){
    temp = abs(value[code_count]);
    code_length = 0;
    while (temp != 0){
        temp >>= 1;
        code_length++;
    }
    if (value[code_count] < 0) temp = negate[code_length] + value[code_count];
    else temp = value[code_count];
if (bits_empty < code_length){ // Enough space in current word?
    temp1 = temp; // No. Create second codeword
    temp = temp >> bits_filled; // align bits
    current_word |= temp; // add bits to current word
    current_word = SWITCH(current_word); // exchange first & second bytes
    Write(current_word); // output first byte
    if ((current_word << 8) == 0xff00) Write(0x00); // shift out first byte
    // if byte equals marker flag, pad
    current_word = SWITCH(current_word); // reverse byte order
    Write(current_word); // output next byte
    if ((current_word << 8) == 0xff00) Write(0x00); // current word now shifted to zero
    // if byte equals marker flag, pad
    current_word = (temp1 << bits_empty); // place remaining bits in new word
    bits_filled = code_length - bits_empty; // calculate bits empty
    bits_empty = 16 - bits_filled; // calculate bits filled
}
else { // Sufficient space in current word
    if (bits_empty > code_length){ // Too much space?
        current_word |= (temp >> bits_filled); // Yes, then align and add value
        bits_filled += code_length; // calculate bits filled
        bits_empty -= code_length; // and bits empty
    }
    else { // exact amount of space
        current_word |= (temp >> bits_filled); // Yes, then align and add value
        current_word = SWITCH(current_word); // reverse byte order
        Write(current_word); // output first byte
        if ((current_word << 8) == 0xff00) Write(0x00); // shift & pad, if necessary
        current_word = SWITCH(current_word); // reverse byte order
        Write(current_word); // output second byte
        if ((current_word << 8) == 0xff00) Write(0x00); // shift & pad, if necessary
        bits_empty = 16; // set bits empty
        bits_filled = 0; // set bits filled
        current_word = 0; // current_word now empty
    }
}
}

```

File : COMPRESS.CPP

```

    }
}
count = 0; // zero counter
temp = GetKey(); // check for cancelling transmission
if (temp !=0 ) {
    ClearOutputBuffer(); // clear buffer
    Write(0xff); // send EOI markder flag
    Write(0xd9); //0xFFD9
    fclose(file_ptr);
    SetTextMode();
    return;
}
} // end of color loop
} // end of mcu_count loop
} // end of segment loop
if(bits_empty != 16) { // pad to end of word if not empty
    temp1 = 0xffff; // with '1' bits
    temp1 >>=bits_empty ; // determine # of 1s bits
    current_word |= temp1;
    current_word = SWITCH(current_word);
    Write(current_word);
    current_word = SWITCH(current_word);
    Write(current_word);
}
Write(0xff); // send FF byte_out;
Write(0xd9); // send EOI byte_out;
delay(2500); // wait, if desired
SetTextMode(); // set monitor to text mode
fclose(file_ptr); // close file
return; // end of program
}

```

File: EXPAND.CPP

```

void expand(void){
float   ArrayY[64],ArrayI[64],ArrayQ[64],*a_p,*quant_table,tempfloat;
int     bits_shifted, LastDC[3] = {0,0,0}, add_bits_length, color=0, mcu_count, bits_available,
        total, k, j, i, i8, pict, rowoffset,
        check[11] = { {0x0},{0x1},{0x2},{0x4},{0x8},{0x10},
                      {0x20},{0x40},{0x80},{0x100},{0x200},};
UINT    *dc_index,*ac_index,*run_index,*value_index,*ac_code,*dc_code,current_word,
        word_in,marker_flag, temp,column,row,key,
        negate[16] = { {0x0},{0x1},{0x3},{0x7},{0xf},{0x1f},{0x3f},{0x7f},{0xff},{0x1ff},
                      {0x3ff},{0x7ff},{0xff},{0x1ff},{0x3ff},{0x7ff}};

UCHAR red,green,blue,c;
memset(ArrayY,0,256);           // initialize arrays
memset(ArrayI,0,256);
memset(ArrayQ,0,256);
memset(LastDC,0,6);
window(1,1,80,max_lines);
clrscr();
cprintf("%s","Waiting for picture. ... strike any key to abort");
begin:                           // check for marker flags or user cancel
if (key != 0){                   // if user cancels then return
    SetTextMode();
    return;
}
temp = 0;                       // initialize check values
marker_flag = 0;
key = Read(&c);                 // read incoming character
if (key == 0xffff) goto begin;  // if none available check for user exit
if (c == 0xff) temp = 1;       // if marker flag set flag and continue
                                // else look for user cancel.
                                // look for start of image flag
while (1) {
    Read(&c);
    if (temp == 1) {
        temp = 0;              // if flag continue
        if (c == 0xd8) break;   // if SOI continue
        else if (c != 0xff) goto begin; // else if false start, look for user cancel
    }
    if (c == 0xff) temp = 1;    // set flag
}
while (1) {                     // get application marker flag or goto begin
    Read(&c);
    if (temp == 1) {
        temp = 0;
        if (c == 0xe0) break;   // found application marker, continue
    }
    if (c == 0xff) temp = 1;    // found flag
}
loop :if (Read(&c) == 0xffff) goto loop; // get length of marker segment
temp = c;
temp <<=8;
loop1 :if (Read(&c) == 0xffff) goto loop1;

```

File: EXPAND.CPP

```

temp += c - 2;

while(temp > 0){
    loop2 :if (Read(&c) == 0xffff) goto loop2;
    switch(c) {
        case (0xcc):{ pict = 0; break;} // high resolution marker
        case (0x66):{ pict = 2; break;} // medium resolution marker
        case (0x33):{ pict = 4; break;} // low resolution marker
        case (0xf0):{ break;} // color attribute
        case (0x0f):{ pict++; break;} // b & w attribute
    }
    temp--; // count parameters
}
SetVESAVideoMode(VESAmode); // set graphics display mode
if (pict >= 4){ // center low resolution image
    row = 160; rowoffset = 160;
    column = pict_parm[pict].column_start;
}
else {
    if (pict >= 2){
        row = 120; rowoffset = 120; // center medium resolution image
        column = pict_parm[pict].column_start;
    }
    else{
        if (pict >= 0){ // center high resolution image
            row = 0; rowoffset = 0;
            column = pict_parm[pict].column_start;
        }
    }
}
marker_flag = Word_in(current_word); // input current word
marker_flag = Word_in(word_in); // and next word
temp = 0; // initialize values
bits_available = 16;
while(1){ // start main loop
    switch (color){ // mcu-count repeats at 100 for debugging
        case(0):{ // set pointers to luminance codes and tables
            ac_code = inv_ac_lum_code;
            dc_code = dc_lum_code;
            quant_table = inv_lumin_table;
            dc_index = dc_lum_index_length;
            ac_index = inv_ac_lum_code_length;
            value_index = inv_ac_lum_value_length;
            run_index = inv_ac_lum_run_length;
            a_p = ArrayY;
            break;
        }
        case(1):{ // set pointers to Chrominancecodes and tables
            ac_code = inv_ac_cro_code;
            dc_code = dc_cro_code;

```

File: EXPAND.CPP

```

    quant_table = inv_cromin_table;
    dc_index = dc_cro_index_length;
    ac_index = inv_ac_cro_code_length;
    value_index = inv_ac_cro_value_length;
    run_index = inv_ac_cro_run_length;
    a_p = ArrayI;
    break;
}
case(2):{                                     // point to 2nd chrominance array
    a_p = ArrayQ;
    break;
}
}
j = 1;                                     // check for dc value, j is code pointer
while (j<10) {                             // since only a few dc values, seach in order
    temp = current_word & un_mask[j];      // unmask data bits
    for (k = dc_index[j-1]; k < dc_index[j]; k++){ // point to code of proper length
        if (temp == dc_code[k]){           // equal
            add_bits_length = k;           // yes, read additional bits length
            bits_shifted = j+1;            // and set bits to be shifted
            j=12;                          // flag loop to break while loop
            break;                         // break for loop
        }
    }
    j++;                                    // increment code length pointer
}

    // remove bits from current word then bring in next bits
current_word <<= bits_shifted;              // shift out code from current word
temp = word_in;                            // retrieve bits in next word to fill gap
if (bits_shifted < bits_available) {        // more than enough bits in next word
    temp >>= 16 - bits_shifted;             // yes, allign words
    current_word |= temp;                   // refill current word
    word_in <<= bits_shifted;              // shift bits out next word
    bits_available -= bits_shifted;         // reduce number of bits available in next word
}
else {                                       // not enough bits in next word
    if (bits_shifted > bits_available) {    // too few bits
        temp >>= 16 - bits_shifted;        // yes, allign bits
        current_word |= temp;              // add bits to word
        if (marker_flag == 0)              //get next word, if not end flag
            marker_flag = Word_in(word_in);
        temp = word_in;                   //copy next word to temp
        bits_shifted -= bits_available;    // set number of bits remaining
        bits_available = 16;               // and bits available
        temp >>= 16 - bits_shifted;        // allign bits
        current_word |= temp;              // refill current word
        word_in <<= bits_shifted;          // shift bits out next word
        bits_available -= bits_shifted;    // reduce number of bits available in next word
    }
}

```

File: EXPAND.CPP

```

        else {
            temp >>= 16 - bits_shifted;           // just the right number
            current_word |= temp;                 // align words
            if (marker_flag == 0)                 // add bits
                marker_flag = Word_in(word_in);   // get next word, if not end flag
            bits_available = 16;                  // reset bits available for new word
        }
    }
    if (add_bits_length != 0){                   // if additional bits are present
        bits_shifted = add_bits_length;          // how many are needed
        value[0] = current_word >> (16-bits_shifted); // set value = additional bits
        if (value[0] < check[add_bits_length])    // is value less than 2^(n-1) [check]
            value[0] -= negate[add_bits_length];  // if so, must represent a negative number
                                                // remove used bits routine

        current_word <<= bits_shifted;
        temp = word_in;
        // bits shift routine
        if (bits_shifted < bits_available) {
            temp >>= 16 - bits_shifted;
            current_word |= temp;
            bits_available -= bits_shifted;
            word_in <<= bits_shifted;
        }
        else {
            if(bits_shifted > bits_available) {
                temp >>= 16 - bits_shifted;
                current_word |= temp;
                if (marker_flag == 0) marker_flag = Word_in(word_in);
                temp = word_in;
                bits_shifted -= bits_available;
                bits_available = 16;
                temp >>= 16 - bits_shifted;
                current_word |= temp;
                bits_available -= bits_shifted;
                word_in <<= bits_shifted;
            }
            else {
                temp >>= 16 - bits_shifted;
                current_word |= temp;
                if (marker_flag == 0) marker_flag = Word_in(word_in);
                bits_available = 16;
            }
        }
    }
}
else value[0] = 0;                             // if additional bits not needed value equals zero
i = 1;                                           // set count for runlength and values arrays
total = 0;                                       // set counter for total number of ac values
while (1){                                       // loop for all values break on EOB marker
    total++;
    j = 1;                                       // j holds number of leading '1' plus 1

```

File: EXPAND.CPP

```

temp = current_word; // temp holds variable length code
while ((temp & 0x8000) == 0x8000){ // while current bit is '1'
    temp <<= 1; // shift bit out
    j++; // let pointer skip to next code length
} // code length s
while (j<16) { // while code length is valid
    temp = current_word & un_mask[j]; // unmask first code with correct number leading 1s'
    for (k = ac_index[j-1]; k < ac_index[j]; k++){ // check all codes of this length
        if (j == 15) k += (temp - ac_code[k]); // if code length is 16 point to proper entry
        if (temp == ac_code[k]){ // code match
            run_length[i] = run_index[k]; // yes, set run_length
            add_bits_length = value_index[k]; // set number of additional bits get value
            bits_shifted = j + 1; // set number of bits to throw away
            j = 16; // flag to end checking
            break; // break leave for loop
        }
    }
    j++; // increase code length
} // end while loop, check next length word
current_word <<= bits_shifted; // perform bit shifting routine
temp = word_in;
if (bits_shifted < bits_available){
    temp >>= 16 - bits_shifted;
    current_word |= temp;
    bits_available -= bits_shifted;
    word_in <<= bits_shifted;
    bits_shifted = 0;
}
else {
    if (bits_shifted > bits_available){
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        if (marker_flag == 0) marker_flag = Word_in(word_in);
        temp = word_in;
        bits_shifted -= bits_available;
        bits_available = 16;
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        bits_available -= bits_shifted;
        word_in <<= bits_shifted;
    }
    else {
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        if (marker_flag == 0) marker_flag = Word_in(word_in);
        bits_available = 16;
    }
}

// get_add_bits just like for dc term
bits_shifted = add_bits_length;

```

File: EXPAND.CPP

```

value[i] = current_word >> (16-bits_shifted);
if (value[i] < check[add_bits_length]) value[i] -= negate[add_bits_length];
current_word <<= bits_shifted;
temp = word_in;

// bits shift routine
if (bits_shifted < bits_available){
    temp >>= 16 - bits_shifted;
    current_word |= temp;
    bits_available -= bits_shifted;
    word_in <<= bits_shifted;
}
else{
    if(bits_shifted > bits_available) {
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        if (marker_flag == 0) marker_flag = Word_in(word_in);
        temp = word_in;
        bits_shifted -= bits_available;
        bits_available = 16;
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        bits_available -= bits_shifted;
        word_in <<= bits_shifted;
    }
    else {
        temp >>= 16 - bits_shifted;
        current_word |= temp;
        if (marker_flag == 0) marker_flag = Word_in(word_in);
        bits_available = 16;
    }
}
if ( (run_length[i] == 0) && (value[i] == 0) ) break; // if last value is end of block, break loop
total += run_length[i]; // else set total number of entries determined
i++; // increase holding pointer
if (total > 65) { // too many entries in array? Terminate
    memset(ArrayY,0,256); // due to comm or decoding error
    memset(ArrayI,0,256); // reset all values and return
    memset(ArrayQ,0,256);
    memset(LastDC,0,6);
    color = 0;
    goto begin;
}
}
memset(a_p, 0, 256); // initial array to zero
a_p[0] = (value[0] + LastDC[color]) * quant_table[0]; // unquantize values
LastDC[color] += value[0]; // add previous DC value to recover DC difference
k=1; // k points to diagonal array position
temp = 1; // temp holds highest normal ordered array entry
for (j=1;j < i;j++){
    k += run_length[j];

```

File: EXPAND.CPP

```

a_p[diag_index[k] ] = value[j] * quant_table[diag_index[k]];
if(temp < diag_index[k]) temp = diag_index[k];
k++;
}
temp = temp >> 3; // divide maximum entry by eight
temp++; // leaving number of non-zero rows
idct(8, a_p); // IDCT operates on all columns and non zero rows
color++; // count this color component complete
if (color == pict_parm[pict].color) { // do we have all needed color component
    color = 0; // reset color
    for (i=0; i < 8; i++) { // display array
        i8 = i * 8;
        for (j=0; j<8; j++) {
            k = (*(ArrayY + i8 + j) - *(ArrayI + i8 + j) * 1.11 + *(ArrayQ + i8 + j) * 1.73);
            k += 128; // compute blue color
            if (k < 0) k = 0; // eliminate noticable roundoff
            else if (k > 255) k = 255; // eliminate noticable roundoff
            blue = k;
            k = (*(ArrayY + i8 + j) - *(ArrayI + i8 + j) * .28 - *(ArrayQ + i8 + j) * .64);
            k += 128; // compute green color
            if (k < 0) k = 0; // eliminate noticable roundoff
            else if (k > 255) k = 255; // eliminate noticable roundoff
            green = k;
            k = (*(ArrayY + i8 + j) + *(ArrayI + i8 + j) * .95 + *(ArrayQ + i8 + j) * .62);
            k += 128; // compute red color
            if (k < 0) k = 0; // eliminate noticable roundoff
            else if (k > 255) k = 255; // eliminate noticable roundoff
            red = k;
            if (VESAmode==0x112) put_pixel_112(j+column*8,i+row,red,green,blue);
            else if(VESAmode==0x110) put_pixel_110(j+column*8,i+row,red,green,blue);
        }
    }
    column++; // point to next column
    if (column == pict_parm[pict].column_end){ // if end of screen reset
        column = pict_parm[pict].column_start; // to beginning row
        row+=8; // increase row count
    }
    if ( marker_flag == 0xffd9 ) { // image complete?
        memset(ArrayY,0,256); // ready for next image
        memset(ArrayI,0,256);
        memset(ArrayQ,0,256);
        memset(LastDC,0,6);
        color =0;
        goto begin;
    }
}
} // end of main loop
} // end of program

```

File: INITIAL.CPP

```

void Initialize(void) {
    VESAmode = SetVESAVideoMode(Mode640x480P24);
    if (VESAmode == 0xffff)
        VESAmode = SetVESAVideoMode(Mode640x480P16);
    if (VESAmode == 0xffff)
        cprintf("video modes not supported");
    struct text_info screen;
    _dos_setvect(0x24, CriticalErrorHandler);
    gettextinfo(&screen);
    if (screen.screenwidth < 80) End(BadScreenWidth);
    max_lines = screen.screenheight;
    SetTextMode();
    textcolor(TerminalForegroundColor);
    textbackground(TerminalBackgroundColor);
    window(1, 1, 80, 24);
    _wscroll = 1;
    textcolor(TerminalForegroundColor);
    textbackground(TerminalBackgroundColor);
    clrscr();
    QuerySetup();
    SetBuffers();
    InstallModemInt();
    _setcursortype(_NORMALCURSOR);
    _wscroll = 1;
    InitComplete = 1;
    int_msr = inportb(base + MSR);
    picture_data = (unsigned char far *) farmalloc(0xffff3);
    if (picture_data == NULL) End(NoMemory);
    CreateTables();
    StatusLine();
    Menu();
    return;
}

```

```

// set 8 bit video mode
// if mode not available
// set planar 5 bit video mode
// VESA mode not supported
// End(NoVESA);
// storage for gettextinfo
// install DOS critical error handler
// place textinfo in screen
// if screen too small quit
// read max screen lines
// place into text mode
// set foreground for terminal
// set terminal background color
// set terminal window
// enable scroll
// set terminal foreground color
// set terminal background color
// clear screen
// set up modem routine
// set I/O buffers
// install modem interrupts
// set cursor for terminal
// set screen scroll
// flag initialization completed
// read MSR
// allocate memory for picture_data
// end if memory not available
// set up tables and codes
// set modem status line
// install menu window

```

File: INITIAL.CPP

```
void StatusLine(void){
char msr;
gettext(1,1,80,24,OldWindowData);           // save screen
window(1, 25, 80, 25);                       // open status line window
_wscroll=0;                                  // remove scroll
textcolor(StatusBackgroundColor);             // set colors for status line
textbackground(StatusForegroundColor);
msr = int_msr;                                // check msr to display
cprintf(stat_format,                          // write current status line
    Resolution[pict], CommPorts[port].name, speed,
    parity, data, stop,
    msr & MSR_CD? "CD" : "",                  // show carrier detect
    msr & MSR_CTS ? "CTS" : " ",             // show clear to send
    i_flow ? " " : "RTS");                   // show flow control
puttext(1,1,80,24,OldWindowData);            // replace screen
window(1, 3, 80, 24);                        // go to terminal window
_wscroll = 1;// set scroll
textcolor(TerminalForegroundColor);          // set colors
textbackground(TerminalBackgroundColor);
}
```

File: MENU.CPP

```

int  GetKey() {                                //get a key or function key
    static
    int  k;
    if ((k = bioskey(1)) != 0){                // key available
        if (k == -1) k = 0;                   // clear control break key
        else{                                  // else
            k = bioskey(0);                   //get key available
            if (NOT (k & 0xff))                // set function key
                k = 0x100 + (k >> 8);
            else k &= 0xff;                   // else set normal key
            return(k);                        // return with key
        }
    }
    return(k);                                // return with key
}

void Menu(void) {
    window(1,1,80,2);                        // draw menu window
    _wscroll=0;                              // remove scroll
    textcolor(MenuForegroundColor);          //set menu text color
    textbackground(MenuBackgroundColor);     // set background color
    gotoxy(1,1);                             // goto line 1
    cprintf("%s",menuline1);                 // display menu line
    gotoxy(1,2);                             // goto line 2
    cprintf("%s",menuline2);                 // display second line
    window(1,3,80,24);                      // size terminal window
    gotoxy(1,3);                             // goto beginning of terminal window
    textcolor(TerminalForegroundColor);     // set terminal window colors
    textbackground(TerminalBackgroundColor);
    StatusLine();                            // set status line
    _wscroll= 1;                             // replace scroll
}

```

File: TABLES.CPP

```

void CreateTables(void) {
    int i,j;
    float sqrt2=sqrt(2), rt2by8=sqrt(2)/8;
    FILE *lumin_pointer;                                // set lumin table
    if((lumin_pointer = fopen("lumin.bin","rb")) != NULL) {
        fread(lumin_table,sizeof(float),64,lumin_pointer);}
    else End(Notables);
    fclose(lumin_pointer);
    lumin_table[0] /= 8;
    for(i=1;i<8;i++) {
        lumin_table[i] *=(rt2by8);
        lumin_table[i*8] *=(rt2by8);
    }
    for(j=8;j<64;j+=8) for(i=1;i<8;i++) lumin_table[i+j] /= 4;

    FILE *cromin_pointer;                                //set chromin table
    if((cromin_pointer = fopen("cromin.bin","rb")) != NULL) {
        fread(cromin_table,sizeof(float),64,cromin_pointer);}
    else End(Notables);
    fclose(cromin_pointer);
    cromin_table[0] /= 8;
    for(i=1;i<8;i++) {
        cromin_table[i] *=(rt2by8);
        cromin_table[i*8] *=(rt2by8);
    }
    for(j=8;j<64;j+=8) for(i=1;i<8;i++) cromin_table[i+j] /= 4;

    FILE *inv_lumin_pointer;
    if((inv_lumin_pointer = fopen("inv_lumn.bin","rb")) != NULL) {
        fread(inv_lumin_table,sizeof(float),64,lumin_pointer);}
    else End(Notables);
    fclose(inv_lumin_pointer);
    inv_lumin_table[0] /= 8;
    for(i=1;i<8;i++) {
        inv_lumin_table[i] /= (4*sqrt2);
        inv_lumin_table[i*8] /= (4*sqrt2);
    }
    for(j=8;j<64;j+=8) {for(i=1;i<8;i++) inv_lumin_table[i+j] /= 4;}

    FILE *inv_cromin_pointer;
    if((inv_cromin_pointer = fopen("inv_crmn.bin","rb")) != NULL){
        fread(inv_cromin_table,sizeof(float),64,inv_cromin_pointer);}
    else End(Notables);
    fclose(inv_cromin_pointer);
    inv_cromin_table[0] /= 8;
    for(i=1;i<8;i++) {
        inv_cromin_table[i] /= (4*sqrt2);
        inv_cromin_table[i*8] /= (4*sqrt2);
    }
    for(j=8;j<64;j+=8) {for(i=1;i<8;i++) inv_cromin_table[i+j] /= 4;}

```

File: TABLES.CPP

```
FILE *diag_index_pointer;
if((diag_index_pointer = fopen("diag_inx.bin","rb")) != NULL) {
    fread(diag_index,sizeof(UINT),64,diag_index_pointer);}
else End(Notables);
fclose(diag_index_pointer);

FILE *ac_crom_huf_code_pointer;
if((ac_crom_huf_code_pointer = fopen("cro_ac.bin","rb")) != NULL){
    fread(ac_crom_huf_code,sizeof(int),256,ac_crom_huf_code_pointer);}
else End(Notables);
fclose(ac_crom_huf_code_pointer);

FILE *ac_lum_huf_code_pointer;
if((ac_lum_huf_code_pointer = fopen("lum_ac.bin","rb")) != NULL) {
    fread(ac_lum_huf_code,sizeof(int),256,ac_lum_huf_code_pointer);}
else End(Notables);
fclose(ac_lum_huf_code_pointer);

FILE *ac_crom_code_length_pointer;
if((ac_crom_code_length_pointer = fopen("crac_len.bin","rb")) != NULL) {
    fread(ac_crom_code_length,sizeof(int),256,ac_crom_code_length_pointer);}
else End(Notables);
fclose(ac_crom_code_length_pointer);

FILE *ac_lum_code_length_pointer;
if((ac_lum_code_length_pointer = fopen("lmac_len.bin","rb")) != NULL) {
    fread(ac_lum_code_length,sizeof(int),256,ac_lum_code_length_pointer);}
else End(Notables);
fclose(ac_lum_code_length_pointer);

// inv ac cro pointers
FILE *inv_ac_cro_code_pointer;
if((inv_ac_cro_code_pointer = fopen("crac_inv.bin","rb")) != NULL) {
    fread(inv_ac_cro_code,sizeof(int),256,inv_ac_cro_code_pointer);}
else End(Notables);
fclose(inv_ac_cro_code_pointer);

FILE *inv_ac_cro_run_length_pointer;
if((inv_ac_cro_run_length_pointer = fopen("crrl_inv.bin","rb")) != NULL) {
    fread(inv_ac_cro_run_length,sizeof(int),256,inv_ac_cro_run_length_pointer);}
else End(Notables);
fclose(inv_ac_cro_run_length_pointer);

FILE *inv_ac_cro_value_length_pointer;
if((inv_ac_cro_value_length_pointer = fopen("crvl_inv.bin","rb")) != NULL) {
    fread(inv_ac_cro_value_length,sizeof(int),256,inv_ac_cro_value_length_pointer);}
else End(Notables);
fclose(inv_ac_cro_value_length_pointer);
```

File: TABLES.CPP

```
//ac lum pointers
FILE *inv_ac_lum_code_pointer;
if((inv_ac_lum_code_pointer = fopen("lmac_inv.bin","rb")) != NULL)
    {fread(inv_ac_lum_code,sizeof(int),256,inv_ac_lum_code_pointer);}
else End(Notables) ;
fclose(inv_ac_lum_code_pointer);

FILE *inv_ac_lum_run_length_pointer;
if((inv_ac_lum_run_length_pointer = fopen("lmrl_inv.bin","rb")) != NULL) {
    fread(inv_ac_lum_run_length,sizeof(int),256,inv_ac_lum_run_length_pointer);}
else End(Notables);
fclose(inv_ac_lum_run_length_pointer);

FILE *inv_ac_lum_value_length_pointer;
if((inv_ac_lum_value_length_pointer = fopen("lmvl_inv.bin","rb")) != NULL){
    fread(inv_ac_lum_value_length,sizeof(int),256,inv_ac_lum_value_length_pointer);}
else End(Notables) ;
fclose(inv_ac_lum_value_length_pointer);

// code length pointers

FILE *inv_ac_cro_code_length_pointer;
if((inv_ac_cro_code_length_pointer = fopen("craclinv.bin","rb")) != NULL){
    fread(inv_ac_cro_code_length,sizeof(int),16,inv_ac_cro_code_length_pointer);}
else End(Notables) ;
fclose(inv_ac_cro_code_length_pointer);

FILE *inv_ac_lum_code_length_pointer;
if((inv_ac_lum_code_length_pointer = fopen("lmacinv.bin","rb")) != NULL){
    fread(inv_ac_lum_code_length,sizeof(int),16,inv_ac_lum_code_length_pointer);}
else End(Notables) ;
fclose(inv_ac_lum_code_length_pointer);

}
```

```

void Take(void) {

float redf, greenf, bluef, Y;
unsigned char red, green, blue;
UINT row, column_start, row_start, i, BW = 0, j, x, y, z;
FILE *file;
SetVESAVideoMode(VESAmode);
if (pict > 3){
    file = fopen("low.rez", "rb");
    row = 120; row_start = 180; column_start = 240;
    if (pict == 5) (BW = 1);
}
else{
    if (pict > 1){
        file = fopen("medium.rez", "rb");
        row = 40; row_start = 120; column_start = 160;
        if (pict == 3) (BW = 1);
    }
    else{
        file = fopen("high.rez", "rb");
        row = 32; row_start = 0; column_start = 0;
        if (pict == 1) (BW = 1);
    }
}
for (z=0; z<pict_parm[pict].total_segments; z++){
    fread(picture_data, sizeof(unsigned char), pict_parm[pict].file_size, file);
    i = row * z;
    for (y=0; y<row; y++){
        j = y * pict_parm[pict].h_rez * 3;
        for (x=0; x<pict_parm[pict].h_rez; x++){
            red = picture_data[j + 3 * x];
            green = picture_data[j + 3 * x + 1];
            blue = picture_data[j + 3 * x + 2];
            if (BW == 1) {
                redf = red; greenf = green; bluef = blue;
                Y = .299 * redf + .587 * greenf + .114 * bluef;
                if (Y >= 255) Y = 255;
                red = Y; green = red; blue = red;
            }
            if (VESAmode == 0x112) put_pixel_112(x + column_start, i + y + row_start, red, green, blue);
            else if (VESAmode == 0x110) put_pixel_110(x + column_start, i + y +
                row_start, red, green, blue);
        }
    }
}
fclose(file);
while (GetKey() == 0);
SetTextMode();
return;
}

```

File: VID.CPP

```

void SetTextMode(void) {                                     // set text mode using bios interrupt
    regs.h.ah = 0;
    regs.h.al = 3;
    int86(VIDEO, &regs, &regs);
}

int SetVESAVideoMode (unsigned int Mode) {
    Mode &= 0xffff;                                         // set bit 15 to clear memory
    regs.h.ah = VESAFUNCTIONID;;
    regs.h.al = 0x02;                                       // set video mode cmd
    regs.x.bx = Mode;
    int86 (VIDEO, &regs, &regs);
    if ((regs.h.al != VESAFUNCTIONID) || (regs.h.ah != 0))return(-1);
    else return(Mode);
}

void put_pixel_110(int x,int y,UCHAR red, UCHAR green, UCHAR blue) {
    unsigned int far *ptr;
    unsigned long offset;
    unsigned char bank;
    unsigned long y1;
    unsigned int redi,greeni,bluei,aplace;

    y1=y;
    offset = y1*640*2+x*2;                                  // find array location
    bank = offset/0x1000;                                    // find bank
    offset = offset%0x1000l;                                // find offset
    outportb(0x3ce,9);                                     // set bank write
    outportb(0x3cf,bank);                                   // write bank
    ptr = (unsigned int *) (0xa0000000 + offset);           // calculate video pointer location
    redi=red>>3;greeni=green>>3;bluei=blue>>3;            // place 5, 6, 5 bits into word
    *ptr = (redi<<10)+(greeni<<5)+bluei;                  // ouput pixel
}

void put_pixel_112(int x,int y, UCHAR red, UCHAR green, UCHAR blue) {
    unsigned char far *ptr;
    unsigned long offset;
    unsigned char bank;
    offset = (unsigned long)y*(640*3+128)+x*3;              // calculate offset
    bank = offset/0x1000l;    // calculate bank
    offset = offset%0x1000l;    // calculate offset
    outportb(0x3ce,9);    // set bank wite
    outportb(0x3cf,bank);    // wrtie bank
    ptr = (unsigned char *) (0xa0000000 + offset);          // point to pixel location
    *ptr++=blue;    // out each byte
    *ptr++=green;
    *ptr++=red;
}

```

VITA

Constantine Peter Bandy was born in Glendale, California on November 21, 1957. He attended elementary schools in Philadelphia, Pennsylvania and graduated from Philadelphia's Central High School in June 1975. Mr. Bandy received a BA (Chemistry) from Temple University, Philadelphia, Pennsylvania in 1979. Mr. Bandy was commissioned in the US Air Force in August 1981, and in 1983, received a BEE from Auburn University, Auburn, Alabama. He then served as a Developmental Engineer and Program Manager with the US Air Force's Electronics Systems Division at Hanscom Air Force Base, Bedford, Massachusetts. In 1987, Mr. Bandy joined the Tennessee Valley Authority, in Knoxville, where he held positions as Engineering Specialist and Supervisor. In 1991, Mr. Bandy joined the US Department of Energy as an electrical engineer.

Mr. Bandy is currently a Captain (Major selectee) in the US Air Force Reserve and is pursuing advanced degrees in electrical engineering at the University of Tennessee.