ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY

CODING AND TRANSMISSION OF NON UNIFORM ALPHABETS

BY

YOHANNES KASSAHUN

DECEMBER 1998

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY
ELECTRICAL ENGINEERING DEPARTMENT

Coding and Transmission of Non Uniform Alphabets

A Thesis Submitted to School of Graduate Studies of Addis Ababa University in Partial Fulfilment for the Degree of Masters of Science in Electrical Engineering

By

Yohannes Kassahun

Advisor

Dr.-Ing Hailu Ayele

Addis Ababa University

December 1998

ADDIS ABABA UNIVERSITY
SCHOOL OF GRADUATE STUDIES
FACULTY OF TECHNOLOGY
ELECTRICAL ENGINEERING DEPARTMENT

CODING AND TRANSMISSION OF NON UNIFORM ALPHABETS

By
Yohannes Kassahun

Approved by Board of Examiners:

Dr. Eneyew Adugna _____
Chairman, Department Graduate Committee (DGC)

Dr.Ing Hailu Ayele _____
Advisor

Fekadu Shewarega _____
Examiner

Prof. Lal Kishore.K _____
External Examiner

18/12/98

To My Family . . .

Contents

Acknowledgments
Abstract
List of Figures

Chapter 1 Introduction 1

1.1 Coding 2
1.2 Fixed Length Encoding 3
1.3 Variable Length Encoding 4

Chapter 2 Background and Literature Review 7

2.1 Information Theory 7
2.2 Entropy 8
2.3 Entropy and Coding 10
2.4 The Noiseless Coding Theorem 11
2.5 Variable Length Code Words 11
2.6 Kraft Inequality 15
2.7 A Source Coding Theorem (Part I) 18
2.8 A Source Coding Theorem (Part II) 20
2.9 Variable Length Encoding Procedure (Huffman Coding) 21
2.10 Compression 23
2.11 Compression Efficiency 23
2.12 Compression Methods 24
2.13 Data Compression and Information Transfer 26
2.14 Compression Effect on Information Transfer 28
2.15 Encoder and Decoder Cost for Huffman Codes 30

Chapter 3 Description of Algorithms and Their Implementation Procedures 32

3.1 General Flow Charts for Huffman Encoder and Decoder 33
3.1.1 Determination of Frequency Distribution of Symbols 34
3.1.2 Code Tree Generation 34
3.1.3 Perform Compression 34
3.1.4 Reading Decoding Table 35

3.1.5 Performing Decompression 36
3.2 Algorithms 36
3.2.1 Algorithm One 36
3.2.2 Algorithm Two 42
3.3 Real Time Text Transmission Over Telephone Line 45
3.3.1 Implementation of Communication Program 45
3.3 Implementation of File Compressor Programs 48
3.4 Implementation of Bitmap Compressor Program 48

Chapter 4 Results and Conclusions 49

4.1 File Compressor and Decompressor Programs 49
4.1.1 Text Files 49
4.1.1.1 Algorithm One 49
4.1.1.2 Algorithm Two 51
4.1.2 Word Processor Documents 53
4.2 Real Time Text Transmission Program 54
4.3 Bitmap Compressor Program 56
4.4 Conclusions 57

Appendix 1 **Windows Communication Functions Used in the Development of the Text Transmission Program** 61

Appendix 2 **The Bitmap Graphics File Format** 73

Appendix 3 **Standard Frequency Distribution of English Alphabets** 82

Appendix 4 **Text Compression and Decompression Programs using Algorithm One** 83

Appendix 5 **File Compressor and Decompressor Program Using Algorithm Two** 95

Appendix 6 **Real Time Text Transmission Program With and Without Huffman Encoding Scheme** 107

Appendix 7 **Bitmap Graphics Compressor Program** 152

REFERENCES 166

List of Figures

Page

Figure 1.1 Block diagram of a typical data transmission or storage system 1
Figure 2.1 Plot of the function $plog_2(1/p)$ 9
Figure 2.2 Tree representing code words of code III in Table 1. 14
Figure 2.3 Binary code tree(solid) imbedded in full tree(dashed) of order 3. 16
Figure 2.4 Basic data compression block diagram. 23
Figure 2.5 The Doni Madonna by Michelangelo printed from GIF file format. 25
Figure 2.6 The Doni Madonna by Michelangelo printed from a JPEG file created using a quality scale of 5. 25
Figure 2.7 Data compression affects the information transfer rate (ITR); through the use of data compression, the methodology and structure of data communication facility may be changed. 27
Figure 2.8 ITR and error rate. 29
Figure 3.1 General Flow Chart for Huffman Encoder 33
Figure 3.2 General Flow Chart for Huffman Decoder 35
Figure 3.3 Model for real time text transmission over telephone line. 45
Figure 4.1 Bar graph for Algorithm One. 50
Figure 4.2 Bar graph for Algorithm Two. 52
Figure 4.3 Bar graph obtained for word processor documents. 53
Figure 4.4 Attained information transfer rate for modem speed of 9600bps. 55
Figure 4.5 The basic JPEG compression algorithm 58
Figure A Transmission flow between two ports 67

ACKNOWLEDGMENTS

I am pleased to express my indebtedness to my advisor Dr.-Ing Hailu Ayele for proposing this project, giving guidance, and constructive suggestions while this project is in progress. I would like to express my gratitude to Dr. Eneyew Adugna for his valuable suggestions. I would also like to express my gratitude to Ato Frehiwot W/Hanna and Ato Tamrat Bayle for giving me software which I used for developing my programs.

Special thanks goes to Faculty of Technology of Addis Ababa University for letting me to pursue my study in Graduate School . Special thanks also goes to DAAD, German Academic Exchange Service, which sponsored me and covered the cost of this project and other expenses, without which the project wouldn't have this final shape.

Thanks also goes to my colleagues for their encouragement and suggestions which helped me in finalizing this work.

Finally, I would like to give special thanks to my family for their love and affection.

Above all I praise Lord who is always with me.

Yohannes Kassahun
December 1998

ABSTRACT

Two algorithms are developed in such a way that the decoding and encoding times for Huffman encoding scheme are minimized. Algorithm One is tested for compression and decompression of text files, while Algorithm Two is tested for compression and decompression of any form of file, real time text transmission, and compression and decompression of bitmap graphics files.

Programs used in the implementation of the developed algorithms in the above mentioned applications are presented. Results of the implementations are also included.

1

Introduction

There are three reasons for encoding data that is about to be transmitted (through space, for instance) or stored (on a computer disk). One reason is for *efficiency*. It clearly makes sense to compress data as much as possible in order to save transmission time or storage space. In fact, data compression is very big business in the computer world. The second reason to encode data is for *error detection and /or correction.* The third reason is for *Secrecy*, that is, so that unauthorized persons cannot read the data.

Encoding for efficiency, error correction, and secrecy are distinct, the first two are related. That is, if some data is to be transmitted or stored, then one would want to both compress it as much as possible and protect it from errors.

Encoding for efficiency lies under the well-known information theory and encoding for error correction falls under coding theory. Encoding for secrecy is the subject of cryptology. Information theory provides a powerful tool for application of discrete probability theory to the problem of encoding for efficiency, and elementary coding theory is an application of algebra. The subject of information theory and coding theory began in 1948 with a famous paper by Claude Shannon, of Bell Labs, entitled *A Mathematical Theory of Communication.* In his theory Shannon stated that by proper encoding of the information, errors induced by a noisy channel or storage medium can be reduced to any desired level without sacrificing the rate of information transmission or storage.

1.1 Coding

The block diagram shown in Figure 1.1 may represent a typical transmission (or storage) system. The *information source* can be either a person or a machine (digital computer). The source output, which is to be transmitted to a given destination, can be either a continuous waveform or a sequence of discrete symbols. The source encoder transforms the source output into a sequence of binary digits (bits) called the information sequence **u**. If the information source is continuos, analog-to-digital (A/D) conversion is necessary. The source encoder is designed such that

1. the number of bits per unit time required to represent the source output is minimized, and
2. the source output can be reconstructed from the information sequence **u** without ambiguity.

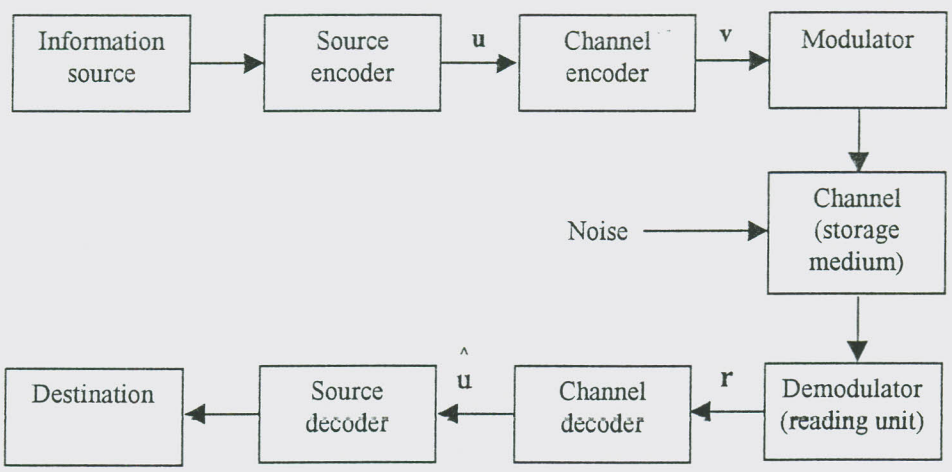


Figure 1.1 Block diagram of a typical data transmission or storage system

The *Channel encoder* transforms the information sequence **u** into a discrete *encoded sequence* **v** called a *code word*. Usually, **v** is also a binary sequence, although in some applications non-binary codes have been used.

Since discrete symbols are not suitable for transmission over a physical channel or recording on a digital storage medium, a *modulator* is necessary that transforms each output symbol of

the channel encoder into a waveform of duration T seconds which is suitable for transmission (or recording). These waveforms enter the *channel* (or *storage medium*) and may be corrupted by noise. Typical transmission channels include telephone lines, high-frequency radio links, microwave links, satellite links, and so on. Typical storage media include core and semiconductor memories, magnetic tapes, drums, disk files, and optical memory units. Each of these examples is subject to various types of noise disturbances. For example on a telephone line, the disturbance may come from switching impulse noise, thermal noise, crosstalk from other lines, or lightening. On magnetic tape, surface defects are regarded as a noise disturbance.

The *demodulator* processes each received waveform of duration T and produces an output that may be discrete (quantized) or continuos (unquantized). The sequence of demodulator output corresponding to the encoded sequence **v** is called the *received sequence* **r**.

The *channel decoder* transforms the received sequence **r** into a binary sequence $\hat{\mathbf{u}}$ called the *estimated sequence*. The decoding starategy is based on the rules of channel encoding, the noise characterstics of the channel (or storage medium). Ideally $\hat{\mathbf{u}}$ should be a replica of the information sequence **u.** However, the noise may cause some decoding errors.

The *source decoder* transforms the *estimated sequence* $\hat{\mathbf{u}}$ into an *estimate* of the source output and delivers this estimate to the *destination*. When the source is continous, it involves digital-to-analog (D/A) conversion. In a well-designed system, the estimate will be a faithful repoduction of the source output except when the channel (or storage medium) is very noisy.

1.2 Fixed Length Encoding

Most of the work done on encoding for error detection/ correction involves the use of *fixed-length* encoding schemes, where all code words have the same lengths. There are two types of *fixed-length* block codes in common use today, block codes and convolutional codes.

In block codes the encoder divides the information sequence into message blocks of k information bits each. A message block is represented by the binary k-tuple $\mathbf{u} = (u_1, u_2, \ldots u_k)$

called *message*. There are a total of 2^k different possible messages. The encoder transforms each message **u** independently into n-tupple $\mathbf{v} = (v_1, v_2, \ldots v_n)$ of discrete symbols called a *code word*. It is possible to see that, corresponding to the 2^k different possible messages, there are 2^k different possible code words at the encoder output. The set of 2^k code words of length n is called an (n, k) *block code*. The ratio $R = k/n$ is called the *code rate*, and is the information bits entering the encoder per transmitted symbol.

The encoder for a convolutional code also accepts blocks of k-bits of the information sequence **u** and produces an encoded sequence (code word) **v** of n-symbol blocks. (In convolutional coding, the symbols **u** and **v** are used to denote sequences of blocks rather than a single block.) Each encoded block depends not only on the corresponding k-bit message block in the same time unit, but also on m previous message blocks. Hence the encoder has a *memory order* of m. The set of encoded sequences produced by a k-input, n-output encoder of memory order m is called an (n, k, m) *convolutional code*. Again here the ratio $R = k/n$ is called the *code rate*.

1.3 Variable Length Encoding

Data compression is best accomplished by *variable-length encoding* schemes (this is a statistical encoding method), where the most frequently occurring source symbols are encoded with the shortest code words. This method can be used to obtain a minimization of the average code length of the encoded data, in a manner similar to that in which Morse selected the dot and dash representation of characters so that a single dot was used to represent the letter E, which is the most frequently encountered character in the English language, while longer strings or dots and dashes were used to represent characters that appear less frequently. The source data may be encoded as either a single symbol at a time or in predefined blocks of symbols. In such a situation, it is the (finite) probability distribution of the source that is important, and not the actual symbols themselves. To each probability distribution there is associated a quantity, called *entropy*, which is the measure of the total amount of "information" in the source. The goal of encoding is to encode the source data in such a way as to add as little additional information beyond the entropy as possible.

The famous information theorem known as *The Noiseless Coding Theorem* **[2]** provides a means for encoding, such that the total information in the encoded message is as close to the entropy of the original message as desired.

In many situations such as the archival data on a computer disk, the entire message is at our disposal, and so it is possible to scan the entire message and compile the actual frequency for the source symbols. This removes the uncertainty associated with the probability distribution. However, it is necessary to store additional frequency along with the message, thus partially defeating the purpose of the encoding.

If the source data is not at our disposal, the problem of uncertainty associated with the probability distribution of the source prevails. Here the form of encoding remains the same but the substance of the encoding is constantly changing. This type of encoding is *adaptive encoding*.

In this thesis, among the *variable-length encoding* schemes, the Huffman Encoding is selected for study and implementation. This encoding scheme is a highly efficient instantaneous encoding scheme. Even though, this coding is highly efficient, the coding and decoding process is not as such an easy process. Two algorithms are developed that will implement Huffman Encoding. The first algorithm is not as efficient as the second algorithm. (Here the word efficient is used to denote speed of coding and decoding not ratio of compression.) The second algorithm is efficient having a much superior performance over the first algorithm. The first algorithm can only be applied to text documents while the second algorithm can be applied to any type of data.

Chapter 2 presents the basic Information Theory and Coding. It also discusses the underlying principles of variable length coding with respect to Huffman coding.

Chapter 3 is dedicated to the discussion of the implementation of the two algorithms developed in this Thesis. It also summarizes the objective that is to be achieved using predefined measures.

The Results obtained in applying the Huffman encoding to storage media and communication over telephone line are presented in chapter 4. The communication over the telephone line involves Text transmission and is real time application. Discussion of the results obtained and conclusion are also presented in this chapter.

2

Background and Literature Review

2.1 Information Theory [3]

Suppose that a source alphabet of q symbols $s_1, s_2, \ldots, s_q$ each with its probability $p(s_1) = p_1$, $p(s_2)=p_2, \ldots, p(s_q) = p_q$ exist. When one of these symbols is received, how much information can be obtained? For example, if $p_1 = 1$ (and of course all other $p_i = 0$), then there is no "surprise," no information, since it is known that what the message have to be. On the other hand, if the probabilities were all different, then when a symbol with a low probability comes one would feel more surprised, get more information, than when a symbol with higher probability came. Thus information is somewhat inversely related to the probability of occurrence.

Surprise is additive- the information from two *independent* symbols is the sum of the information from each separately. Since the probability of two independent choices are multiplied together to get the probability of the compound event, it is natural to define the amount of information as

$$I(S_i) = -\log_2 p_i \tag{2.1}$$

And,

$$I(S_1) + I(S_2) = I(S_1, S_2) = -\log_2 p_1 - \log_2 p_2 \tag{2.2}$$

2.2 Entropy [3]

If one gets $I(s_i)$ units of information when the symbol s_i is received, then how much does one get on the average? The answer is that since p_i is the probability of getting the information $I(s_i)$, then on the average one gets, for each symbol s_i, the information given by

$$p_i I(S_i) = - p_i \log_2 p_i \tag{2.3}$$

It can be easily seen that on the average, over the whole alphabet of symbols s_i, one can get

$$\sum_{i=1}^{q} p_i \log_2 \left(\frac{1}{p_i} \right) \tag{2.4}$$

It is possible to use any base for the log system. If base 2 is used for the log system, the unit of information is called *bit* (binary digit). If base e is used, the unit of information is called *nat.* If base 10 is used the unit of information is called *Hartley*, after R.V.L. Hartley, who first proposed the use of the logarithmic measure of information.

For any base r called radix r, this important quantity is labeled as

$$H_r(S) = -\sum_{i=1}^{q} p_i \log_r p_i \tag{2.5}$$

Equation (2.5) is called the *entropy* of the signaling system S having symbols s_i and probability p_i. Of course,

$$H_r(S) = H_2(S) \log_r 2 \tag{2.6}$$

This is the entropy function for the distribution *when all that is considered are the probabilities* p_i. This function involves only the probability distribution that is, it is the function of a probability distribution p_i and does not involve the s_i.

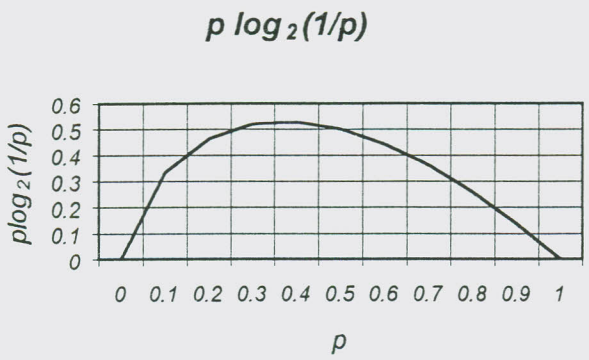


Figure 2.1. Plot of the function p $\log_2(1/p)$

The maximum value of the function *p* $\log_2(1/p)$ occurs at $p = 1/e$. The *entropy* function of a distribution summarizes one aspect of a distribution much as the *average or mean* in statistics provides information on the distribution. The *entropy* has properties of both the arithmetic mean (the average) and the geometric mean.

The entropy function measures the amount of uncertainty, surprise, or information one can get from the outcome of some situation, say the reception of a message or the outcome of some experiment. It is therefore an important function of the probabilities of the individual events that occur. Thus, in designing an experiment it is usually preferred to maximize the entropy function.

Consider two probability distributions, x_i and y_i such that

$$\sum x_i = 1 \qquad \sum y_i = 1 \tag{2.7}$$

It is known that $\log_e x \leq (x-1)$. Using this inequality it is possible to show that the following relationship will hold true

$$\sum_{i=1}^{q} x_i \log_2 \left(\frac{y_i}{x_i} \right) \leq 0 \tag{2.8}$$

For any distribution of equally likely distribution having q members, it is possible to show that the entropy of the signaling system S also satisfies the relation

$$H_2(S) \leq \log_2 q \tag{2.9}$$

In other words, for any other distribution than the equally likely distribution, the entropy is less than $\log_2 q$.

2.3 Entropy and Coding [3]

Consider *any instantaneous code* that has some definite code word length l_i represented in some radix r. From the Kraft inequality (to be presented latter)

$$K = \sum_{i=1}^{q} \frac{1}{r^{l_i}} \leq 1 \tag{2.10}$$

Define the numbers Q_i :

$$Q_i = \frac{r^{l_i}}{K} \tag{2.11}$$

Where

$$\sum_{i=1}^{q} Q_i = 1 \tag{2.12}$$

This Q_i may be regarded as a probability distribution. By using, the fundamental inequality (2.8)

$$\sum_{i=1}^{q} p_i \log_2 \left(\frac{Q_i}{p_i} \right) \leq 0 \tag{2.13}$$

and in expanding the log term into a sum of logs, one term leads to the entropy function,

$$H_2(S) = \sum_{i=1}^{q} p_i \log_2 \left(\frac{1}{p_i} \right) \leq \sum_{i=1}^{q} p_i \log_2 \left(\frac{1}{Q_i} \right) \tag{2.14}$$

Using (2.11) in the right hand side,

$$\begin{aligned} H_2(S) &\leq \sum_{i=1}^{q} p_i \left(\log_2 K - \log_2 r^{-l_i} \right) \\ H_2(S) &\leq \log_2 K + \sum_{i=1}^{q} p_i l_i \log_2 r \end{aligned} \tag{2.15}$$

Again by Kraft inequality $K \leq 1$, so $\log_2 K \leq 0$. Dropping this can only strengthen the inequality. Therefore,

$$H_2(S) \leq \sum_{i=1}^{q} (p_i l_i) \log_2 r = L \log_2 r \tag{2.16}$$

Or

$$L \geq H_r(S) \tag{2.17}$$

Where L is the average code word length,

$$L = \sum p_i l_i \tag{2.18}$$

This is the fundamental result needed: *the entropy supplies a lower bound on the average code length L for any instantaneous decodable system.*

2.4 The Noiseless Coding Theorem [2]

When encoding a source S, it certainly seems reasonable that one will need at least as many bits of information in the encoding as there are in the source. However, for efficient encoding, one may need a few extra bits in the encoding. Since the entropy of S measures the amount of information in S, it should come as no surprise that the minimum average code word length of any encoding of S should be at least as great as the entropy of S. This is the essence of the theorem giving the lower bound on the average code word length which is formally stated as follows:

Let S be an information source. Then

$$H_r(S) \leq \text{Minimum Average Code Length}(S) \tag{2.19}$$

Where the minimum average code word length among all uniquely decodable encoding schemes of S

2.5 Variable Length Code Words [1]

Suppose that a discrete memoryless source U has K letter alphabets $a_1, \ldots, a_K$ with probabilities $P(a_1), \ldots, P(a_K)$. Each source letter is to be represented by a code word consisting of a sequence of letters from a prescribed code alphabet. Denote the number of letters in the code word corresponding to a_K by n_K. Later, the source letters to be sequences of characters from a simpler source shall be considered, thus generalizing the above situation considerably.

One will primarly be interested in $\bar{n}$; the average number of code letters per source letter u. From the law of large numbers, if a very long sequence of source letters is encoded by the

above encoding scheme, then the number of code letters per source letter will be close to $\bar{n}$ with high probability.

$$\bar{n} = \sum_{k=1}^{K} P(a_k) n_k \tag{2.20}$$

Source Letters	*P(a_K)*	*Code I*	*Code II*	*Code III*	*Code IV*
a_1	0.5	**0**	**0**	**0**	**0**
a_2	0.25	**0**	**1**	**10**	**01**
a_3	0.125	**1**	**00**	**110**	**011**
a_4	0.125	**10**	**11**	**111**	**0111**

Table 2.1. Sample types of variable length codes.

Here it can be seen that Code **I** has the uniform property that letters a_1 and a_2 are both coded into the same code word, **0**. Thus this code word cannot be uniquely decoded into source letter that gave rise to it. Such codes cannot represent the letters from the source.

Code **II** suffers the same defect as Code **I**, although in a more subtle way. If the sequence a_1 a_1 is emitted from the source, it will be encoded into **00**, which is the same as the code word for a_3. This will present no problem as far as decoding is concerned if some sort of spacing or separation is provided between successive code words. On the other hand, if such spacing is available, it should be considered as a separate symbol, *s*, in the code alphabet, and code words in the Code **II** shall be given as as **0**s, **1**s, **00**s, and **11**s must be listed. By explicitly denoting spacing when required such codes can be considered merely as special cases of codes with no spacing. For this reason, such codes are excluded from further special consideration.

It can be observed that neither Code **I** nor Code **II** in Table 1 can be used to represent the sources since neither is uniquely decodable. This leads us to the following definition: *A code is uniquely decodable if for each source sequence of finite length, the sequence of the code letters corresponding to that source sequence is different from the sequence of code letters corresponding to any other source sequence.*

The above definition does not immediately suggest any way of determining whether or not a code is uniquely decodable. One is, however, primarily interested in a special class of codes that satisfy a restriction known as the prefix condition; these codes are easily shown to be uniquely decodable.

To illustrate this definition, let the *k*th code word in a code be represented by $\mathbf{X_k} = (x_{k,1}, \ldots, x_{k,nk})$, where $x_{k,1}, \ldots, x_{k,nk}$ denote the individual code letters making up the code word. Any sequence made up of initial part of $\mathbf{X_k}$, that is, $x_{k,1}, \ldots, x_{k,ni}$ for some $i \leq n_k$ is called a prefix of $\mathbf{X_k}$. *A prefix condition code is defined as code in which no code word is the prefix of any other code word.*

In the above table it can be seen that Code **I** is not a prefix condition code since **1**, the code word for a_3, is a prefix of **10**, the code word for a_4. Also, looking carefully at the definition of a prefix, it can be seen that **0**, the code word for a_1, is a prefix of **0**, the code word for a_2. In other words, any code with two or more code words the same can not form a prefix condition code. Not every uniquely decodable code satisfies a prefix condition. To see this consider Code **IV** in the above table. Here every code word is a prefix of every longer code word. On the other hand, unique decoding is trivial since symbol **0** always indicates the beginning of a new code word. Prefix condition codes are distinguished from the other uniquely decodable codes, however, by the end of a code word always being recognizable so that decoding can be accomplished without the delay of observing subsequent code words. For this reason, prefix condition codes are sometimes called *instantaneous* codes.

A convenient graphical representation of a set of code words satisfying the prefix condition can be obtained by representing each code word by a terminal node in a tree. The tree representing code word of Code **III** is shown below. Starting from the root of the tree, the two branches leading to the first order nodes correspond to the choice between **0** and **1** as the first letter of the code words. Similarly, the two branches stemming from the right hand first order node correspond to the choice between **0** and **1** for the second letter of the code word if the first letter is **1**; the same representation applies to the other branches. The successive digits of each code word can be thought of as providing the instructions for climbing from the root of the tree to the terminal code representing the desired source letter. A tree can be used to

represent the code words of a code that does not satisfy the prefix condition, but in this case, some intermediate nodes in the tree will correspond to code words.

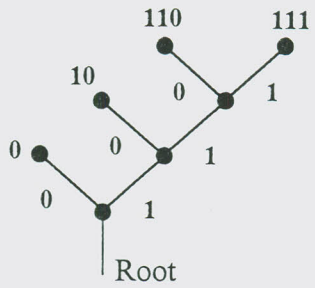


*Figure 2.2. Tree representing code words of Code **III** in Table 1.*

Next, the problem of choosing a prefix condition code in such a way as to minimize $\bar{n}$ will be considered. First, the problem is treated in a heuristic way, then some general theorems about the code word lengths will be presented, and finally an algorithm for constructing a code that minimizes it will be presented.

For a code satisfying the prefix condition, a receiver observing a sequence of code letters and tracing its way up a tree as in Figure 2.2 to decode a source message can be visualized. At each node in this tree, the next code digit provides information about which branch to take. It can be seen that the sum of the mutual information at successive nodes leading to a given terminal node is just the self-information of the associated source digit. Thus, to achieve a small $\bar{n}$, it is desirable to achieve large average mutual information at each of the intermediate nodes in the tree. This, in turn, suggests trying to choose the code words in such a way that each branch rising from a node in the associated tree is equiprobable. If a code existed with $\bar{n} < H(U)$ ($H(U)$ being the entropy of the source), it is possible, for large L, to encode most source sequences of length L with fewer than $H(U)$ code letters per symbol which is impossible. For the above example $H(U) = 1.75$; from this it can be concluded that the minimum value for $\bar{n}$ is 1.75.

2.6 Kraft Inequality [1].

If the integers $n_1, n_2, \ldots, n_K$ satisfy the inequality

$$\sum_{k=1}^{K} D^{n_k} \leq 1 \tag{2.21}$$

then a prefix condition of alphabet size D exists with these integers as code word lengths. Conversely, the lengths of every prefix condition code satisfy (for maximum entropy)

$$P(a_k) = D^{-n_k} \tag{2.22}$$

(It can be seen that the theorem does not say that any code whose lengths satisfy (2.21) is a prefix code.

Define a *full tree* of order n and alphabet size D nodes of order i stemming from each node i-1 for each i, $1 \leq i \leq n$. Observe that a fraction D^{-1} of the nodes of each of each order $i \geq 1$ stem from each of the D nodes of order one. Likewise, a fraction D^{-2} of the nodes of each order $i \geq 2$ stem from each of D^2 nodes of order 2 and a fraction D^{-i} of the nodes of each order greater than or equal to i stem from each of the D^i nodes of order i.

Now let $n_1, n_2, \ldots, n_K$ satisfy (2.21). A method of constructing a prefix condition code with these lengths by starting a full tree of order n equal to the largest of the n_k and assigning various nodes in this full tree as terminal nodes in the code tree will be shown. Thus, when the construction is completed, the code tree will imbed in the full tree as shown in the following figure. To simplify the notation, assume that the n_k are arranged in increasing order, $n_1 \leq n_2 \leq \ldots \leq n_k$.

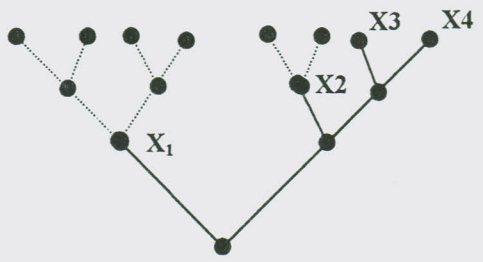


Figure 2.3. Binary code tree (solid) imbedded in full tree (dashed) of order 3.

Pick any node of order n_1, say $\mathbf{x_1}$, in the full tree as the first terminal node in the code tree. All nodes on the full tree of each order greater than or equal to n_1 are still available for use as terminal nodes in the code tree except for the fraction D^{-n1} that stem from the node $\mathbf{x_1}$. Next pick any available node of order n_2, say $\mathbf{x_2}$, as the next terminal node in the code tree. All nodes in the full tree of each order greater than or equal to n_2 are still available except for the fraction $D^{-n1} + D^{-n2}$ that either $\mathbf{x_1}$ or $\mathbf{x_2}$. Continuing in this way, after the assignment of the kth terminal node in the code tree, all nodes in the full tree of each order greater than or equal to n_k are still available except for the fraction

$$\sum_{i=1}^{k} D^{-n_i} \tag{2.23}$$

stemming from $\mathbf{x_1}$ to $\mathbf{x_k}$. From (2.21), this fraction is always strictly less than 1 for $k < K$, thus there is always a node available to be assigned as the next terminal node.

The code tree corresponding to any prefix condition code can be imbedded in a full tree whose order is the largest of the code word lengths. A terminal node of order n_k in the code tree has stemming from it a fraction D^{-nk} of the terminal node in the full tree. Since the sets of terminal nodes in the full tree stemming from different terminal nodes in the code tree are disjoint, these fractions can sum to at most 1, yielding (2.21).

Let a code have code word lengths $n_1, n_2, \ldots, n_K$ and have D symbols in the code alphabet. If the code is uniquely decodable, then the Kraft inequality (2.21) must be satisfied. **[1]**

Let L be an arbitrary positive integer, and consider the identity

$$\left(\sum_{k=1}^{K} D^{-nk}\right)^{L} = \sum_{k_1=1}^{K}\sum_{k_2=1}^{K}\cdots\sum_{k_L=1}^{K} D^{-[n_{k_1}+n_{k_2}+\cdots+n_{k_L}]} \tag{2.24}$$

It can be observed that there is a distinct term on the right-hand side of (2.24) corresponding to each possible sequence of L code words. Furthermore, the quantity $n_{k1} + n_{k2} + \ldots + n_{kL}$ gives the total length in the code letters of the corresponding sequence of code words. Thus, if A_i be the number of sequences of L code words having a total length of i code letters, (2.24) can be rewritten as

$$\left[\sum_{k=1}^{K} D^{-nk}\right]^{L} = \sum_{i=1}^{L_{n\max}} A_i D^{-i} \tag{2.25}$$

where $n_{\max}$ is the largest of the n_k.

If the code is uniquely decodable, all code word sequences with a length of i code letters are distinct and thus $A_i \le D^i$. Substituting this into (2.24),

$$\left[\sum_{k=1}^{K} D^{-nk}\right]^{L} \le \sum_{i=1}^{L_{n\max}} 1 = L_{n_{\max}} \tag{2.26}$$

$$\sum_{k=1}^{K} D^{-n_k} \le \left(L_{n\max}\right)^{1/L} \tag{2.27}$$

Equation (2.27) must be valid for all positive integers L, and taking the limit $L\to\infty$, (2.28) is obtained.

$$\sum_{k=1}^{K} D^{-n_k} \le 1 \tag{2.28}$$

(2.28) is Kraft Inequality. Since the code word lengths for any uniquely decodable code satisfy (2.21), and since it is possible to construct a prefix condition code for any set of lengths satisfying (2.21), any uniquely decodable code can be replaced by a prefix condition

code without changing any of the code word lengths. Thus the subsequent theorems concerning average code word length apply equally to uniquely decodable codes and the subclass of prefix condition codes.

2.7 A Source Coding Theorem (Part I) [1]

Given a finite source ensemble U with entropy H (U) and given a code alphabet of D symbols, it is possible to assign code words to the source letters in such a way that the prefix condition is satisfied and the average length of the code words, $\bar{n}$, satisfies the inequality

$$\bar{n} < \frac{H(U)}{\log D} + 1 \tag{2.29}$$

Furthermore, for any uniquely decodable set of code words,

$$\bar{n} \geq \frac{H(U)}{\log D} \tag{2.30}$$

It is possible to show the equivalent inequality that

$$H(U) - \bar{n} \log D \leq 0 \tag{2.31}$$

Let $P(a_1), \ldots, P(a_K)$ be the probabilities of the source letters and let $n_1, \ldots, n_K$ be the code word lengths .

$$H(U) - \bar{n} \log D = \sum_{k=1}^{K} P(a_k) \log \frac{1}{P(a_k)} - \sum_{k=1}^{K} P(a_k) n_k \log D \tag{2.32}$$

Taking $-n_k$ inside the logarithm and combining terms, we have

$$H(U) - \bar{n} \log D = \sum_{k=1}^{K} P(a_k) \log \frac{D^{-n_k}}{P(a_k)} \tag{2.33}$$

Using the inequality $\log Z \leq (Z-1) \log e$ for $Z > 0$,

$$H(U) - \bar{n} \log D \leq (\log e)[\sum_{k=1}^{K} D^{-n_k} - \sum P(a_k)] \leq 0 \tag{2.34}$$

The last inequality in (2.34) follows from Kraft inequality, (2.21) that is valid for any uniquely decodable code. This verifies (2.30). The equality holds in (2.30) if and only if

$$P(a_k) = D^{-n_k} \qquad 1 \le k \le K \tag{2.35}$$

This is a previously derived condition for each code letter to have the maximum entropy. A code satisfying (2.29) will be choosen. If the code-word lengths did not have to be integers, it is possible to choose the n_k to satisfy (2.35), however, by choosing n_k to be the integer satisfying

$$D^{-n_k} \le P(a_k) \le D^{-n_k+1}; \qquad 1 \le k \le K \tag{2.36}$$

Summing (2.36) over k, the left-hand inequality becomes the Kraft inequality, (2.21), and a prefix condition code exists with such lengths. Taking the logarithm of the right side inequality of (2.36),

$$\begin{aligned} \log P(a_k) &< (-n_k + 1)\log D \\ n_k &< \frac{-\log P(a_k)}{\log D} + 1 \end{aligned} \tag{2.37}$$

Multiplying (2.37) by $P(a_k)$ and summing over k, one can obtain (2.29).

Stronger results can be achieved if, instead of providing code words for individual source letters, we provide code words directly for sequences of L source letters. This will be the subject of the next section.

2.8. A source Coding Theorem (Part II) [1]

Given a discrete memoryless source U with entropy H (U), and given a code alphabet of D symbols, it is possible to assign code words to sequences of L source letters in such a way that the prefix condition is satisfied and the average length of the code words per source letter $\bar{n}$, *satisfies the following inequalities*

$$\frac{H(U)}{\log D} \leq \bar{n} \leq \frac{H(U)}{\log D} + \frac{1}{L} \tag{2.38}$$

Furthermore, the left-hand inequality must be satisfied for any uniquely decodable set of code words. [1]

Consider the product ensemble of sequences of L source letters. The entropy of the product ensemble is $LH(U)$ and the average length of the code words is $L\bar{n}$, using $\bar{n}$ as the average length per source letter. The minimum achievable $L\bar{n}$, assigning a variable length code word to each sequence of L source letters, satisfies

$$\frac{LH(U)}{\log D} \leq \bar{n} \leq \frac{LH(U)}{\log D} + 1 \tag{2.39}$$

Dividing by L, the theorem is seen to be true.

Part II of A Source Coding Theorem is very similar to part I. If L is taken to be arbitrarily large and applying the law of large numbers to a long string of sequences each of L source letters, it can be seen that part II is a special case of the first part of the source coding theorem.

2.9 Variable-Length Encoding Procedure (Huffman Coding) [1]

D.A. Huffman proposed what is now referred to as Huffman Coding scheme in 1952. In what follows optimum is taken to mean that no other uniquely decodable set of code words have smaller average code word length than the given set.

Let the source letters, $a_1, \ldots, a_K$ have probabilities $P(a_1), \ldots, P(a_K)$, and assume for simplicity of notation that the letters are ordered so that $P(a_1) \geq P(a_2) \geq \ldots \geq P(a_K)$. Let $\mathbf{x_1}, \ldots, \mathbf{x_K}$ be a set of binary code words for this source and let $n_1, \ldots, n_K$ be the code word lengths. The code words $\mathbf{x_1}, \ldots, \mathbf{x_K}$ corresponding to an optimum code, in general, will not be unique and often the lengths $n_1, \ldots, n_K$ will also not be unique. In what follows, some restriction and conditions will be imposed that must be satisfied by at least one optimum code and then show how to construct the code using those conditions. Attention will be given to prefix condition codes since any set of lengths achievable in a unique decodable code is achieved in a prefix condition code.

For any given source with $K \geq 2$ letters, an optimum binary code exists in which the two least likely code words, $\mathbf{x}_k$ and $\mathbf{x}_{k-1}$, have the same length and differ in only the last digit, $\mathbf{x}_k$ ending in 1 and $\mathbf{x}_{k-1}$ in 0. **[1]**

For at least one optimum code, n_K is greater than or equal to each of the other word lengths. To see this, consider a code in which $n_K < n_i$ for some i. If the code words $\mathbf{x_i}$ and $\mathbf{x_k}$ are interchanged then the change in $\bar{n}$ is

$$\begin{aligned} \Delta &= P(a_i)n_K + P(a_K)n_i - P(a_i)n_i - P(a_K)n_K \\ \Delta &= \left[P(a_i) - P(a_K)\right]\left[n_K - n_i\right] \leq 0 \end{aligned} \tag{2.40}$$

Thus any code can be modified to make n_K the minimum length without increasing $\bar{n}$.

The problem of finding an optimum code has therefore now been reduced to the problem of finding an optimum code for an ensemble with one fewer bits. A systematic procedure for carrying out the above operations is demonstrated in the following figure **[1].**

First, the two least probable messages, a_4, and a_5 in this case are tied together, assigning **0** as the last digit for a_4 and **1** as the last digit for a_5. Adding the probabilities for a_4 and a_4'. On the next stage the two likely messages are a_1 and a_2, and at this point the two messages remain. Looking at the resulting figure, it can be seen a code tree has been constructed for U, starting at the outermost branches and working down to the trunk. The code words are read off the tree from right to left.

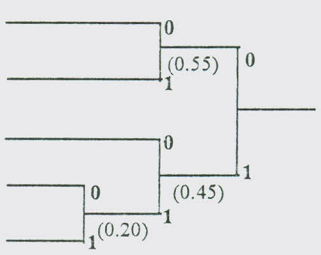
Code word	*Message*	*Pr(a_k)*	
00	a_1	0.3	
01	a_2	0.25	
10	a_3	0.25	
110	a_4	0.10	
111	a_5	0.10	

Table 2.2 Huffman coding Procedure

2.10 Compression [4]

Data Compression can be explained in relation to two general areas: Compression efficiency and Compression methods. The basic Data Compression block diagram is shown below. An original data stream operated upon according to one or more compression algorithm results in generation of a compressed data stream.

Figure 2.4 Basic data compression block diagram.

2.11 Compression Efficiency [4]

The compression of the original data is an encoding process and the compressed data is an encoded data stream and the reverse process is decoding process. The degree of data reduction as a result of compression process is measured by compression ratio. This ratio measures the quantity of compressed data in comparison with the quantity of the original data.

$$Compression\ ratio = \frac{Length\ of\ original\ data\ in\ bytes}{Length\ of\ compressed\ data\ in\ bytes} \tag{2.41}$$

It is obvious that the higher the compression ratio the more effective the compression technique employed. Another term used as measure of compression is the figure of merit,

$$Figure\ of\ merit = \frac{Length\ of\ compressed\ data\ in\ bytes}{Length\ of\ original\ data\ in\ bytes} \tag{2.42}$$

The figure of merit is the reciprocal of the compression ratio and must always be less that unity for compression process to be worth while. The fraction of data reduction is one minus the figure of merit.

$$Fraction\ of\ Data\ Reduction = 1 - Figure\ of\ Merit \tag{2.43}$$

Thus, a compression technique that results in one character of compressed data for every three characters in the original data stream would have a compression ratio of 3, a figure of merit of 0.33 and a fraction of data reduction of 0.66.

Random(Pure) data, by definition, should not have any redundancy. While the compression ratio for this type of data should be unity, in many instances the improper design of a compression algorithm or its improper application may result in a degree of data expansion, resulting in a compression ratio falling below unity.

Usually, average compression ratio is more important than the ratio achieved at a particular time. In general, good algorithms operating on text can be expected to achieve an average compression ratio of 2.0, while excellent algorithms based upon sophisticated processing techniques will achieve an average compression ratio exceeding 3.0.

2.12 Compression Methods[4]

Compression techniques are generally classified into two general groups: Lossless and Lossy compression techniques.

Lossless compression techniques are fully reproducible and are primarily restricted to data operations. This techinque is usually applied in areas like data consisting of accounts receivable, payroll and health insurance claims and so on. It is extremly important that once such data is comressed, its decompression should result in the exact reconstruction of the original data. Other terms used to describe lossless compression include 'reversible' and 'non-destructive' compression.

Lossy compression techniques may or may not be fully reproducible and are primarily restricted to operations on images, video and audio.Although the result of decompression may not provide an exact duplication of the original data, the differences between the original and reconstructed may be so minor as to be difficult to see or hear. Since the compression ratio so obtained by use of lossy compression can significantly exceed the compression ratio

obtainable by the use of lossless compression, the primary tradeoff will be governed by need for reproducibility versus the requirments for storage and transmission.

To see the effect of lossy compression the well known painting 'The Doni Madonna' by Michelangelo Buonarroti, which reflects his view of his statues is used. Figure 2.5 illustrates the printed version of the GIF file. The storage requirment for the GIF file is 200739 bytes. Compressed JPEG file were created using quality level of 5 where 100 would represent the best quality and 0 the worest quality. As quality decreases the size of resulting file decreases and the reproducibility of the image also decreases as the quality level decreases. Figure 2.6 with quality level 5 reduced the file storage requirment to 8876 bytes, expanded the size of blocks of pixels so they are fully noticable throughout the image, resulting in a very large degree of image distortion.

Figure 2.5. The Doni Madonna by Michelangelo printed from GIF file format

Figure 2.6 The Doni Madona by Michelangelo printed from a JPEG file created using a quality scale of 5.

The quality level versus storage space is illustrated in the following table:

Quality level	File storage
100	200739
75	48958
50	31835
25	20964
10	12439
5	8876

Table 2.3 Quality level versus file storage for the above sample Doni Madonna

2.13 Data Compression and Information Transfer [4]

Data transmitted between terminals, a terminal and a computer or two computers, several delay factors may be encountered which cumulatively affect the information transfer rate. Data transmission over a transmission medium must be converted into an acceptable format for that medium. When digital data is transmitted over analog telephone lines, modems must be employed to convert the digital pulses of the business machine into a modulated signal acceptable for transmission on the analog telephone circuit. The time between the first bit entering the modem and the first modulated signal produced by the device is the modem's internal delay time. Since two such devices are required for a point to point circuit, the total internal delay time encountered during a transmission sequence equals twice the modem's internal delay time. Such times can range from a few to 10 or more milliseconds (ms). The second delay encountered on a circuit is a function of the distance between points and circuits or propagation delay time. This is the time required for the signal to be propagated or transferred down the line to the distant end.

Once data is received at the distant end it must be acted upon, resulting in a propagation delay which is a function of the computer or terminal employed as well as the quantity of transmitted data which must be acted upon. Processing delay times can range from a few milliseconds where a simple error check is performed to determine if the transmitted data was received correctly to many seconds where a search of a database must occur in response to a transmitted query.

Each time the direction of transmission changes in a typical half duplex protocol, control signals change at the associated modem to computer and modem to terminal interfaces. The time required to switch control signals to change the direction of transmission is the line turn around time which can result in delays of up to 250 or more milliseconds, depending upon the transmission protocol employed.

If real time compression and decompression are employed on both the transmitter and receiver sides, there is an additional encoding and decoding delay times on the transmission and reception of the data. All these delays have a major effect on the information transfer rate.

Despite the above delays, data compression has benefits on communications in general. The following figures illustrate one of the benefits

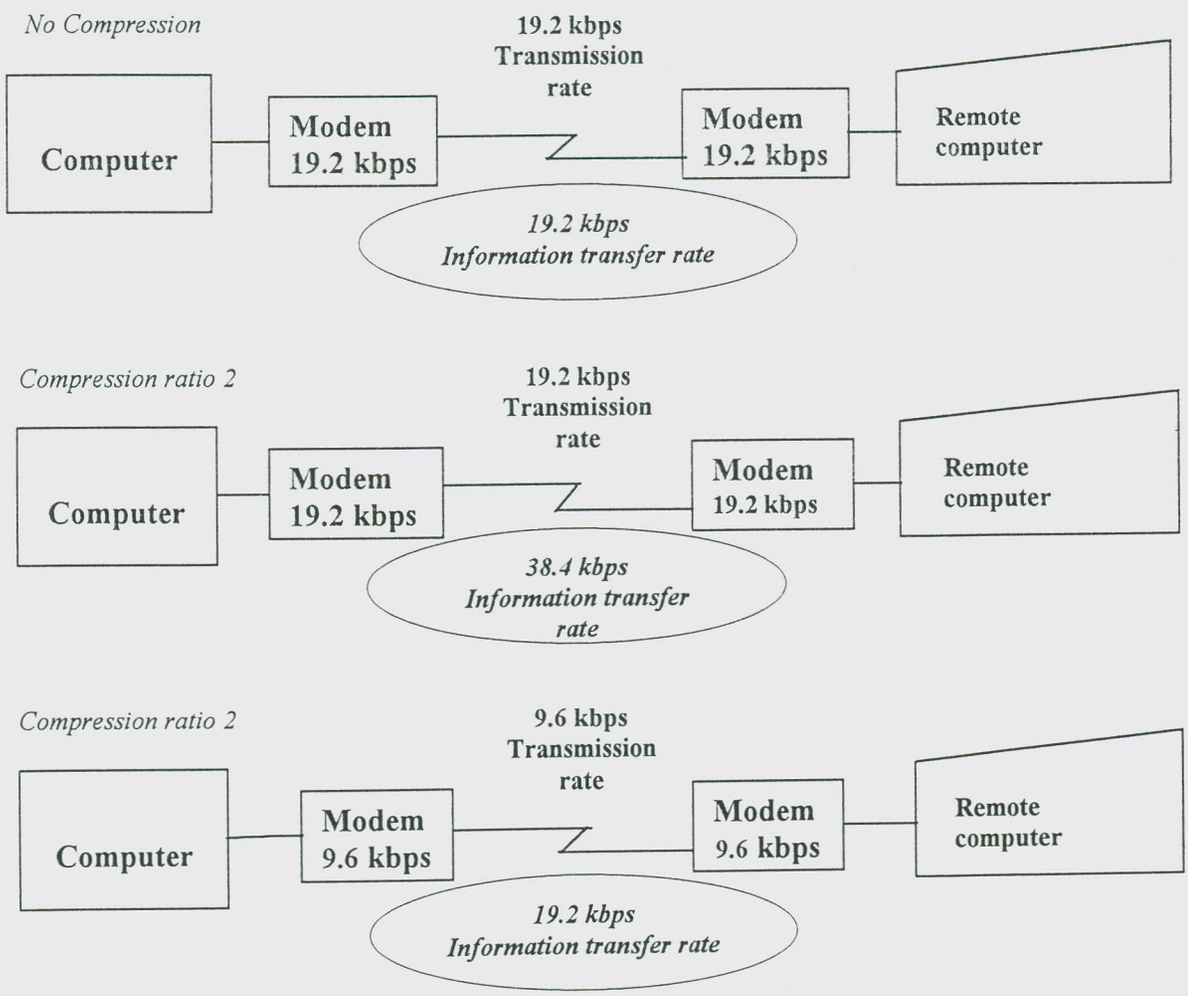


Figure 2.7 Data compression affects the information transfer rate (ITR); through the use of data compression, the methodology and structure of a data communications facility may be changed.

As can be seen from the diagrams, the information transfer rate is increased as the compression ratio of the transmitter is increased. Thus, we can see that by increasing the information transfer rate it is possible to decrease the cost of transmission.

For switched network modems, data compression provides the ability to obtain an information transfer capability at a fraction of the cost of higher speed modems. This is due to the complex modulation schemes used by modems operating at 19200 bps in comparison with less complex schemes used by 9600 bps modems. By incorporating a compression algorithm into a lower operating rate modem, it becomes possible to achieve a data throughput between 9600 and 19200 bps for the cost of a read only memory (ROM) chip.

2.14 Compression Effect on Information Transfer [4]

The effective information transfer ratio (EITR) can obtain by using the formula

$$EITR = \frac{B_{IC} D_1 (1 - P)}{T_R \left[T_C (D_2 + C) + 2(T_{PA} + T_L + T_P) + \left(\frac{A B_C}{T_R} \right) \right]} \qquad (2.44)$$

Where:

D_1 = original data block size in characters prior to compression

D_2 = compressed data block size in characters to include special compression indication characters.

B_{IC} = Information bits per character

B_C = Total bits per character

A = Characters in the acknowledgement message

C = Control characters per message block

T_R = Data transfer rate (bps)

T_C = Transmission time per character (B_C/ T_R)

T_{PA} = Processing and acknowledgment time

T_L = Line turn around time

T_P = Propagation delay time

P = Probability of one or more errors in block.

The effective information transfer ratio can be approximated by

$$EITR \approx ITR * CR \tag{2.45}$$

Where ITR is Information transfer rate and CR is compression ratio. The above formulas are used for a communication medium with full duplex model.

If a 'stop and wait ARQ' error control procedure is applied where there is a return channel available for the transmission of acknowledgments, then the transfer ratio becomes:

$$ITR = \frac{B_{IC} D_1 (1 - P)}{T_R \left[T_C (D_2 + C) + 2(T_{PA} + T_P) + \frac{A B_C}{T_S} \right]} \tag{2.46}$$

A typical plot of Information transfer ratio is shown below: (The plot is for the case 'stop and wait ARQ' not considered).

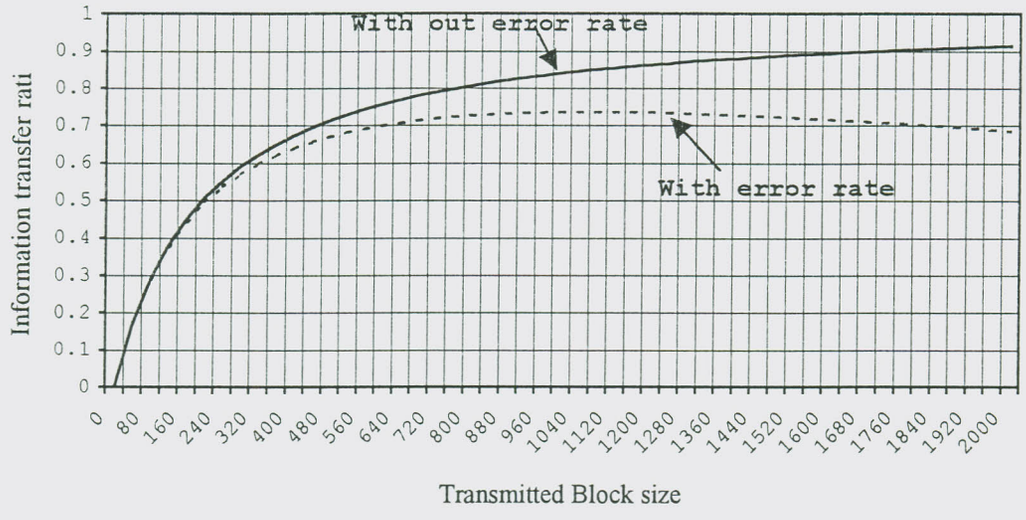


Figure 2.8 ITR and error rate

Since an error free line is not something a transmission engineer can reasonably expects, a maximum block size will exist beyond which our line efficiency will decrease. At this point, only data compression will result in additional transmission efficiencies. In addition from a physical standpoint, the buffer area of some devices may prohibit block sizes exceeding a certain number of characters. So data compression can become an effective mechanism for increasing transmission efficiency while keeping data buffer requirements within an acceptable level.

2.15 Encoder and Decoder Cost for Huffman Codes [5]

There are various implementations for Huffman codes. For all of them, whether they are implemented in hardware or software, there is a cost incurred in terms of storage space.

The encoder requires the following storage: (assuming restricted length Huffman codes)

- A look-up table for the code words. For a code with N code words, this table requires N times the size of the longest code in Bytes.
- A look-up table for the lengths of each code. For a code with N code words, this table requires N Bytes. (Assuming that the length of the longest code is not greater than 255).
- Storage for code tree. Even though it depends on a particular implementation, the maximum size of the code tree description is twice of that of code words.
- A storage for heap where sorting is mainly performed. This is the same as that of the code words.

On the decoder side the following storage is required:

- A look-up table for code words
- A look-up table of code word lengths
- Storage for code tree.

For large N the look-up table will become predominant in both the encoder and decoder. The major problem is selecting or minimizing the code words. This in turn leads to the need for selecting appropriate number of symbols. So if one selects appropriate (Optimal) number of symbols, then it is possible to reduce the cost of implementation.

Moreover, the algorithms (functions) used in encoding and decoding have major influence on the cost. Some algorithms require a lot of hardware and/or software commands while others require minimum hardware and/or software commands. The amount of hardware or software required to implement these algorithms will add to the cost. Again this will contribute to the cost of implementing the encoder or decoder.

Another cost is the encoding and decoding speed. However, not all applications require speedy decoding. This is the case, for example, in communication applications where the data processing bandwidth at the transmitter and receiver is much larger than the bandwidth of the communication channel.

3

Description of Algorithms and Their Implementation Procedures

The main purposes of this work is to study, design and implement efficient (both in terms of compression ratio and speed) variable length encoding schemes for various applications. Huffman encoding scheme is selected as the encoding and decoding scheme.

Algorithms have been developed and implemented for the purpose of file compression and decompression, real time text transmission with compression, and compression and decompression of graphics file.

In this chapter, the general schemes of algorithms and their implementations are presented and discussed. The presentation will focus on the general scheme of the algorithms, discussion of the major components of the algorithms and procedures developed for the implementation of the encoding and decoding schemes. The programs written to implement the algorithms are presented in the appendices.

3.1 General Flow Charts for Huffman Encoder and Decoders

A general flow chart for the encoder of a Huffman code used for compression where the input data is at our disposal is shown below:

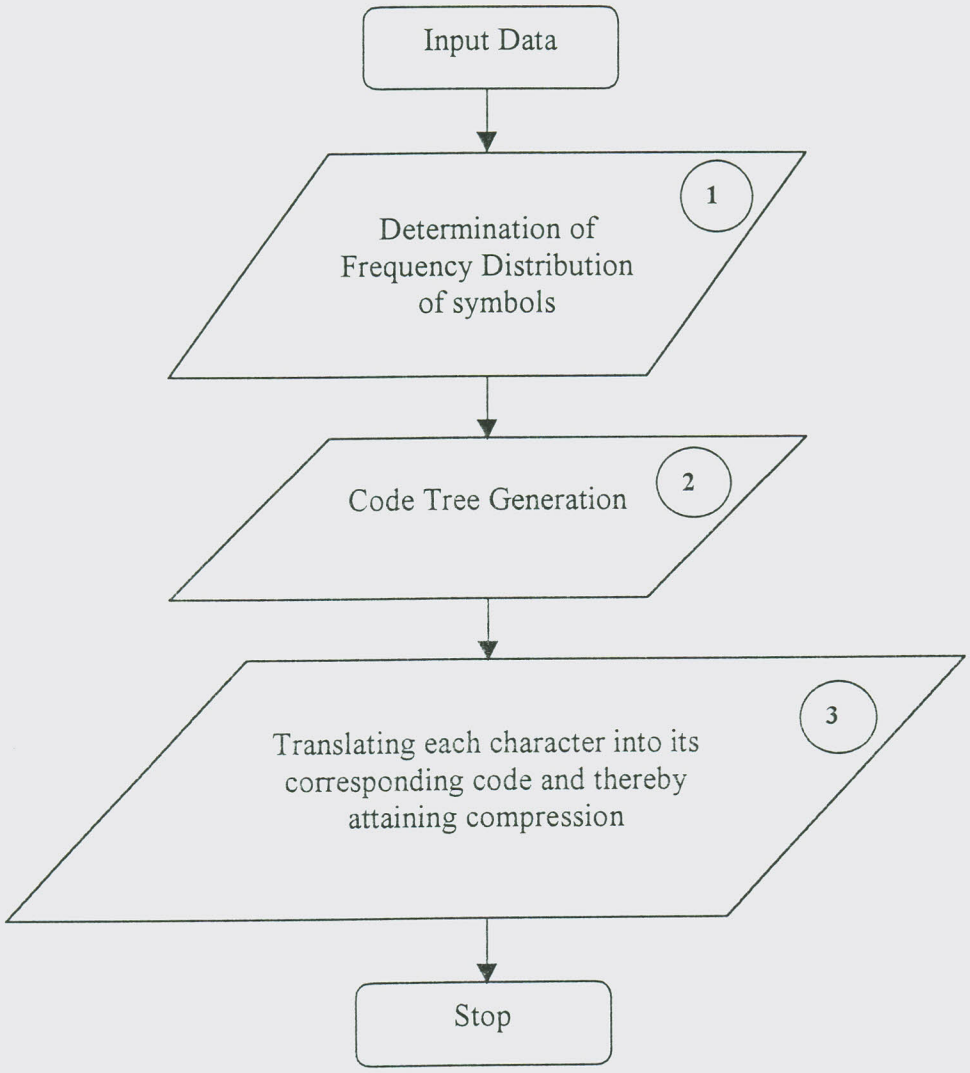


Figure 3.1 General Flow Chart for Huffman Encoder.

3.1.1 Determination of Frequency Distribution of Symbols

The data at hand is scanned and the frequency of occurrence of each symbol is determined. Once this is done it is possible to generate the code tree used for coding and decoding. Frequency distribution of symbols is determined into two ways. One way is to read each symbol one by one and counting the frequency of occurrence of each symbol. This is usually a slow process because of the access time to read each symbol from storage. The second way is to read a block of symbols from storage to a buffer and determining the frequency of occurrence of symbols. This method significantly minimizes the time required for determination of frequency of occurrence each symbol.

3.1.2 Code Tree Generation

Code tree generation is the most important step that a Huffman Encoder has to perform. This process requires efficient sorting and searching methods. Once the Code tree is generated it is possible to encode and compress the source data.

3.1.3 Perform Compression

After the Code tree is generated, the compression process can be established. This process requires searching for the code of an input data symbol. The code of the input symbol is then serialized for concatenation with the code of the next input symbol. If the length of the concatenated code (length measured in number of bits) is not greater than the length of a byte then a code of the next symbol is concatenated with the current concatenated code. This process continues until the length of the concatenated code is greater than or equal to the length of a byte. Then the byte that is formed will be saved to storage as compressed byte. The remaining bits are again concatenated with the next code until they form a byte and the process continues until all symbols are encoded. Thus an efficient methods for serializing and concatenating the codes are necessary for speedy operations.

The general flow chart for the decoder of Huffman code used for decompression is shown below:

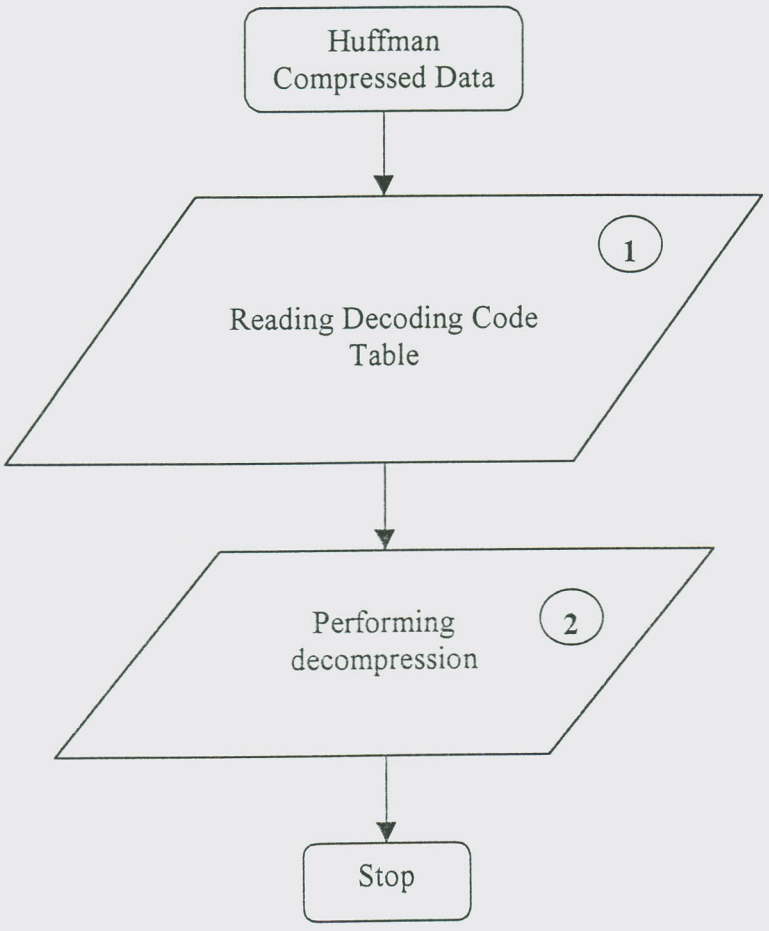


Figure 3.2 General Flow Chart for Huffman Decoder

3.1.4 Reading Decoding Table

The decoding table is usually saved at the beginning of the file or in another file. This table can be read once and transferred into a buffer to increase the reading speed of the decoding table.

3.1.5 Performing Decompression

The decompression process involves:

- Reading a byte
- Serializing the byte to determined whether it could be decompressed into two or more symbols. If it is not possible to decompress it, the next byte will be serialized and concatenated with the current symbol. Again it has to be determined if it is possible to decompress the current word (two bytes form a word). This process continues until the concatenated code can be decompressed. After a decomposable code is obtained then a search for a valid code is performed starting from the first bit. When a valid code is obtained the code will be translated into a decompressed symbol. This process continues until no other valid code is obtained. The left over bits from previous process are concatenated with the bits of the next byte and the whole process will repeat it self until all the bytes are decompressed.

As can be seen efficient serializing and valid code searching procedures are necessary for speedy operations.

3.2 Algorithms

There are two algorithms developed in this work.

3.2.1 Algorithm One

The steps in this algorithm for the encoder are:

- Determine the frequency distribution of each character. This is done either by reading each character (symbol) one by one or reading blocks of characters (symbols) into a buffer.

- Combine the least probable symbols until you form a tree. Here sorting is repeatedly used. The sorting algorithm used here is quick sort. The tree is stored in array of record having the following fields

    a. For input symbols the first field contains node numbers where the tips of the tree are ordered as the input symbols appear in ASCII table while the nodes of the tree are numbered in their order of combination. For combined symbols this field contains the resulting combined node number of the nodes in the forth and fifth fields.
    b. The second field of the record contains input symbols. In this case the symbols are characters. For combined symbols this field is meaningless and contains a character #.
    c. For input symbols the third field contains the frequency of the symbol. For combined symbols this field contains the number of symbols combined.
    d. For input symbols the forth field is meaningless and contains or set to –1, while for combined nodes this field contains the node number to be combined with the next field.
    e. For input symbols fifth field is again meaningless and contains –1 while for combined nodes this field contains the node number to be combined with the forth field.
    f. This node contains the status of that particular node whether that node is combined or not. If it is combined this field is set to `'C'` else if it is not combined it is set to `'N'`.

A sample table showing the tree stored in array of the above record for text file containing the statement `'This is trial one two three to test Huffman Code Tree Development.'` is shown in table 3.1.

Node Number	*Character*	*Frequency*	*Forth node to be combined*	*Fifth node to be combined*	*Status*
0	(space)	11	-1	-1	C
1	.	1	-1	-1	C
2	C	1	-1	-1	C
3	D	1	-1	-1	C
4	H	1	-1	-1	C
5	T	2	-1	-1	C
6	a	2	-1	-1	C
7	d	1	-1	-1	C
8	e	10	-1	-1	C
9	f	2	-1	-1	C
10	h	2	-1	-1	C
11	i	3	-1	-1	C
12	l	2	-1	-1	C
13	m	2	-1	-1	C
14	n	3	-1	-1	C
15	o	5	-1	-1	C
16	p	1	-1	-1	C
17	r	3	-1	-1	C
18	s	3	-1	-1	C
19	t	7	-1	-1	C
20	u	1	-1	-1	C
21	v	1	-1	-1	C
22	W	1	-1	-1	C
Combination of nodes starts from here					
23	#	2	21	7	C
24	#	2	22	4	C
25	#	2	3	20	C
26	#	2	2	16	C
27	#	3	1	23	C
28	#	4	13	12	C
29	#	4	9	10	C
30	#	4	26	25	C
31	#	4	6	24	C
32	#	5	5	17	C
33	#	6	14	18	C
34	#	6	11	27	C
35	#	8	29	28	C
36	#	8	30	31	C
37	#	10	15	32	C
38	#	12	33	34	C
39	#	15	19	35	C
40	#	18	36	37	C
41	#	21	8	0	C
42	#	27	38	39	C
43	#	39	40	41	C
44	#	66	42	43	N

Table 3.1 Sample tree stored in the above array of records

- Generate the code: From the tree stored in the above array, traversing through the tree until the tip of the tree will generate the code. The code for the above sample is shown below.

Character	*Frequency*	*Length of the code*	*Code Generated*
(Space)	11	3	111
.	1	5	00000
C	1	6	100111
D	1	6	100101
H	1	6	100000
T	2	5	10100
a	2	5	10001
d	1	6	000010
e	10	3	110
f	2	5	01111
h	2	5	01110
i	3	4	0001
l	2	5	01100
m	2	5	01101
n	3	4	0011
o	5	4	1011
p	1	6	100110
r	3	5	10101
s	3	4	0010
t	7	3	010
u	1	6	100100
v	1	6	000011
w	1	6	100001

Table 3.2 Code generated for the above sample

For example the code for character 't' is generated as follows:

1. The first step is to identify the node number. In this case the node number for 't' is 19.
2. The second step is to search this node number in the fourth and fifth fields of the record in the combined nodes. As can be seen from the table 3.1 the status of the combined nodes is 'C'. From table 3.1 node number 19 is combined with node number 35 resulting in node number 39.
3. If the node number in the combined nodes is in the fourth field of the record, bit 0 is assigned as part of its code else bit 1 is assigned as part of

its code. As can be seen from table 3.1 node number 19 is stored in the fourth field of the record thus the first bit of its code will become 0.

4. Steps 1 to 3 are repeated on the resulting node number and the second bit of the code will then be determined. In this case node number 39 and 38 are combined to form node number 42. Since node number 39 is stored in the fifth field of the record the next bit of the code will become 1. Again steps 1 to 3 are repeated for node number 42. Node number 42 is combined with node number 43 resulting in node number 44 the last node number. Again since node number 42 is in the fourth field of the last record the last bit for the code will be 0. Since node number 44 is the last node the process of determining the code of letter `'t'` will stop. And the resulting code for `'t'` is `'010'`.

- Perform compression: This procedure contains the following steps in it:

  a. Reading block of text into a buffer
  b. Search for a character (character taken from the input block) in the code table.
  c. Read the code of the character and serialize the bits to be concatenated with the next code of the next character.
  d. When the length of the concatenated codes is greater than certain specified length, where the length is a multiple of 8, then save the concatenated codes as bytes thus resulting in compressed bytes.
  e. Repeat Steps a to e until the end of the file.

- Finally calculate average length of the codes, Entropy and size of compressed file. These can be calculated right just after the code table is generated.

The algorithm for the decoder is:

- Read the decoding table. This table contains
  - A character or symbol

- The length of the code for this symbol, and
- The code itself where each bit is represented as one character. Bit 0 is represented by character '0' and bit 1 is represented by character '1'.

- Initialize the decoder. This step is used to skip the decoding table that is saved at the beginning of the compressed file.

- Decode and decompress the file. This step contains the following steps:
  - Read compressed character
  - Translate or decompress the character. This procedure has the following steps:
    a. Load the code of the compressed character on a buffer called decompression buffer.
    b. Step through each bit of the decompression buffer and search if the resulting code really exists. If the code exists in the decoding table then the character for that code is the decompressed character. Continue until all the compressed codes contained in the decompression buffer are decompressed. The next compressed character is read and concatenated with the left over bits and step 2 will repeat. This process continues until end of file is reached.

The compression algorithm has the following buffers:

- A buffer for the look up table of size 15360 bytes for holding a code having maximum length of 60.
- A buffer for compression of size 128 bytes
- A buffer for reading input symbols of size 128 bytes
- A buffer for holding the frequency distribution of characters read of size 256 bytes
- Two buffers for holding characters whose frequency is different from zero. The total size of the two buffers is 768 bytes

While the decompression algorithm has got the following buffers:

- A buffer for holding the decoding table of size 15360
- A decompression buffer of size 6400 bytes
- A buffer for holding the characters and their lengths of size 6144 bytes
- Different buffers used for different purposes of size 316 bytes.

3.3.2 Algorithm Two

The second algorithm is described as follows:

The steps in the encoding algorithm are:

- Determine the frequency distribution of each character. This is done either by reading each character (symbol) one by one or reading blocks of characters (symbols) into a buffer.

- Build initial heap. This procedure has two steps in it:
  - For characters with non-zero frequency distribution set the value of the elements of the heap buffer to their index. Here index refers to the index of the elements of the buffer containing the frequency distribution of the characters.
  - A reheap procedure again rearranges the heap to obtain a 'legal' heap, which depends on the frequency distribution of characters of non-zero frequency. This procedure simply rearranges the heap and makes it suitable for the combination process in developing the code tree.

- Build code Tree: This procedure builds the code tree from the arranged heap and stores it in a buffer. This procedure repeatedly uses the reheap routine to rearrange the combined nodes.

- Generate Code Table: This routine generates the code tree by traversing through the tree structure. It saves both the code in one buffer and the code length in another buffer. This routine is developed to be highly efficient because unlike Algorithm One, which saves the bits of the code as characters, this routine deals and saves the bits of the codes as bits of particular word (combination of bytes). This significantly increases the speed of generation of the code.

- Perform Compression: This routine performs compression as follows:

    a. First it reads in a character and looks for its code in the code buffer.
    b. Shift the code to the right with the *length of code minus one* times and check if there is a byte to be saved as a compressed character. If there is a byte to be saved as a compressed character then save the compressed character.
    c. Again shift the code to the right with the *length of code minus two* times and check if there is a byte to be saved as a compressed character. Save the compressed character if there is a byte to be saved as a compressed character.
    d. Shift the code to the right with *length of the code minus n* times where *n* is an integer and check if there is a byte to be saved as compressed character until (*length of the code minus n*) is zero.
    e. Repeat steps a to e until all the characters are compressed.

- Generate report: The purpose of this report is to display the results obtained in applying this algorithm. From the report one can observe how far the algorithm achieved compression and how far the attained average code length is near to the entropy of the source. This routine generates reports on
    - The original file size
    - Size of header characters: The header contains the original file size, the code and the code length. The Size of the header is 772 bytes.
    - Size of compressed characters
    - Total size of resulting output characters
    - Percentage savings

- Average code length attained
- Entropy of the source file

The steps in the decoder algorithms are:

- Read the original file size, the code and the code length, all contained in the header of the compressed file.

- Build the decompression code tree from the code saved in the header. This is the reverse of the Generate code table in the encoder algorithm. From the code table one can easily generate the decompression code tree.

- Perform decompression: The decompression algorithm has the following steps:
  a. Read in compressed character
  b. Obtain the current index of the decompression code array
  c. Shift the character *n* times to the right and *bit wise and* it with *one* and add it to the current index shifted *one* times to the left and assign it to new current index.
  d. If the value of the decompression tree at the newly calculated index is less than or equal to *zero* then the value of the decompressed character is negative of the value of the decompression tree at this newly calculated index.
  e. Repeat Step c and d for *n* running from *seven* to *zero*
  f. Repeat Steps from a to e until all the characters are decompressed.

The compression algorithm has the following buffers:

- A buffer for determining the frequency distribution of size 1024 bytes
- Buffer for building heap of size 257 bytes
- Buffer for holding the resulting code of size 512 bytes
- Buffer for holding the length of the codes generated of size 256 bytes
- Buffer used to generate both the code table and the code tree of size 1024 bytes
- Buffer for reading in the input symbols of size 8000 bytes

The decompression algorithm has the following buffers:

- Buffer for holding the code of size 512 bytes
- Buffer for holding the code length of size 256 bytes
- Buffer for holding the decompression tree of size 1024

The maximum code length supported for this algorithm is 16 bits (Length measured in bits). But the code length can be increased at the expense of the size of Header characters. Reasonably it is found that a code length of 16 is enough for blocks of data of size 50 kilobytes.

3.3 Real Time Text Transmission Over Telephone Line

Real time text transmission over telephone lines is accomplished using modems. The purpose of modems is to convert the binary bits to signals in audio range of duration T seconds which is suitable for transmission and to demodulate the received audio signal of duration T back to binary bits.

3.3.1 Implementation of Communication Program [6]

The model for the real time text transmission over telephone line used in this work is shown in figure 3.2

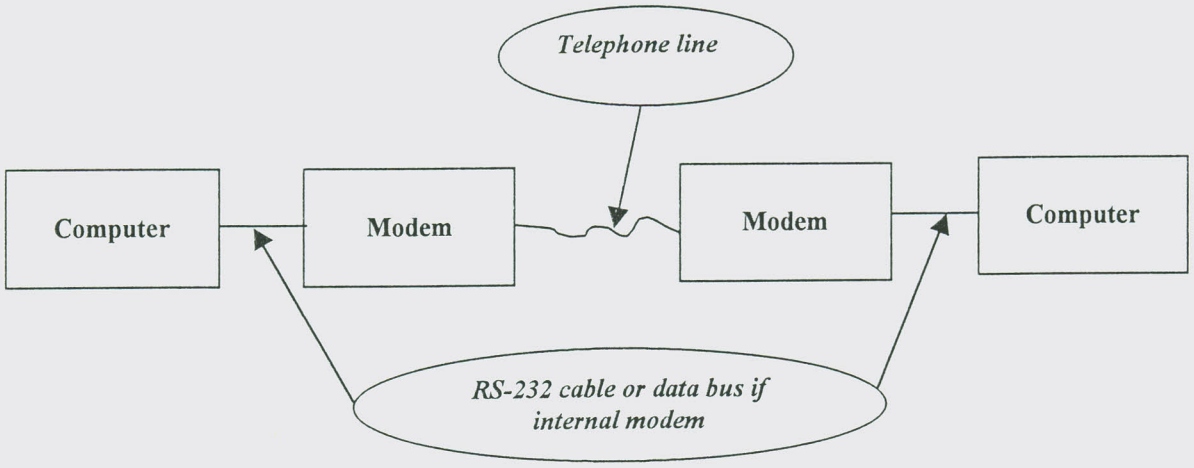


Figure 3.3 Model for real time text transmission over telephone line

The communication program is developed by using windows 32-bit communication functions. The discussion of these functions is presented in (Appendix 1).

These communication functions are used to program the serial and parallel ports of the computer. The serial port of the computer is connected to the modem for communication over telephone line. If the modem is external the serial port is connected with the modem using the RS-232 / V.24 interface.

Devices that represent the origin or destination of data over a communication system are formally classified as *data terminal equipment (DTE)*. Interposed between the DTE and the communications system are additional devices collectively called *data circuit terminating equipment (DCE)*. A DCE converts the output from DTE into a form that is appropriate for transmission over the communications system. A modem is an example of a DCE that converts the output of a PC's serial port into signal that can be transmitted over the telephone network.

The RS-232 standard defines the electrical characteristics of the individual interchange signals and associated circuitry. Signals are assumed to be originated by a *generator* and detected by a *receiver*. The circuit voltage is measured at the point where the two devices connect, which is known as the *interface point*. Normal signal voltages on an RS-232 interchange circuit represent one of two states. The signal is in the first state when the voltage is between –15V and -3V. The second state is defined as a voltage between +3V and + 15V. The region from –3V to + 3V is defined as the *transition region*, and the state of the circuit is undefined when the voltage is within this region. The following table summarizes the correspondence between the circuit states and definitions for data transmission, timing and control functions.

Application	Voltage Level	
	$-15 \leq V \leq -3$	$3 \leq V \leq 15$
Data (Signal State)	MARK	SPACE
Data (Binary Value)	1	0
Timing and control Functions	OFF	ON

Table 3.3 RS-232 Signal Definitions

The MARK signal level is used to indicate that a transmission line is idle and the SPACE is set by the receiver that an error or a condition requiring attention is present.

Real time text transmission using Huffman encoding can be implemented by placing the decoding table at the receiver side and by placing an encoding table at the transmitter side. Both the encoding and decoding tables must be optimal for compression to be effective.

An optimal code for a text document can be obtained by studying the frequency distribution of characters for different sample text documents. From the frequency distribution thus obtained it is possible to generate the corresponding Huffman code.

The generated Huffman code is then tested for different text documents and the attained compression ratio is investigated to see how far it is satisfactory. The program developed reads the code generated from a file, hence it suffices only to play around the code until one obtains efficient codes in terms of compression ratio.

The algorithm used in this program is the second algorithm because smaller size of file is needed for the generated code. Moreover, the speed of the second algorithm is superior to the first algorithm. The speed of the second algorithm to compress a single character is much higher than the transmission speed of currently available modems to transmit a single character. Hence the delay due to encoding and decoding times is negligible.

3.3 Implementation of File Compressor Program

A file compressor and decompressor is implemented using both algorithms. The first being used only for text compression and decompression. The second is used for compression and decompression of any form of file. The performance of both algorithms is tested in both implementations.

3.4 Implementation of Bitmap Compressor Program

A bitmap is a graphics format representation in Windows operating system (Refer to Appendix 2). The second algorithm is modified to be applied to compression and decompression of a bitmap graphics file. Because of limited length code in the second algorithm the size of the bitmap is restricted to be below 50 kilobytes. This program is also tested for different bitmaps having different number of colors.

4

Results and Conclusions

4.1 File Compressor and Decompressor Programs

The two algorithms are tested on text files and the following results are obtained. Section 4.1.2 presents result obtained in applying Algorithm Two on word processor documents.

4.1.1 Text Files

4.1.1.1 Algorithm one

	File size in byte	*Compressed File size in byte*	*Percentage savings*	*Average code length*	*Entropy*	*Percentage deviation of Average code length from Entropy*
1	605	3119	-415.5	4.26446	4.21728	1.12
2	939	3368	-258.7	3.16933	3.14028	0.93
3	3066	5517	-79.9	4.43738	4.37944	1.32
4	4378	6223	-42.1	4.20786	4.16058	1.14
5	8537	10133	-18.7	5.20089	5.16994	0.60
6	11794	11439	3.0	4.95913	4.92385	0.72
7	13829	13520	2.2	5.06913	5.0409	0.56
8	14902	13550	9.1	4.62281	4.58442	0.84
9	16483	14100	14.5	4.86277	4.8286	0.71
10	16690	14762	11.6	4.98933	4.95796	0.63
11	17773	15287	14.0	5.00219	4.95658	0.92
12	23377	18888	19.2	4.93151	4.89959	0.65
13	24030	19186	20.2	5.05493	5.02462	0.60
14	26295	21201	19.4	4.98505	4.9386	0.94
15	28537	22873	19.8	5.37229	5.34259	0.56
16	28561	22191	22.3	4.95126	4.9189	0.66
17	32530	24695	24.1	4.94626	4.90813	0.78

Continued...

	File size in byte	Compressed File size in byte	Percentage savings	Average code length	Entropy	Percentage deviation of Average code length from Entropy
18	32530	24695	24.1	4.94626	4.90813	0.78
19	38517	28733	25.4	5.01685	4.9837	0.67
20	39795	28286	28.9	4.82902	4.7887	0.84
21	42191	30078	28.7	4.82729	4.80516	0.46
22	42485	30391	28.5	4.86948	4.83497	0.71
23	43636	31458	27.9	4.91761	4.89122	0.54
24	49484	33320	32.7	4.62281	4.58442	0.84
25	49613	35216	29.0	4.91851	4.88744	0.64
26	50585	30522	39.7	4.03889	4.0099	0.72
27	72752	54911	24.5	5.55402	5.52974	0.44
28	72924	53682	26.4	5.43508	5.41285	0.41
29	74270	46167	37.8	4.48878	4.44092	1.08
30	83681	54063	35.4	4.70006	4.67332	0.57

Table 4.1 Results of compression program for different sample text files for Algorithm One.

The negative result shows that the over all size of compressed file including the header characters is greater than the size of original file; but still this does not mean that there is no compression as can be seen from the average code length obtained.

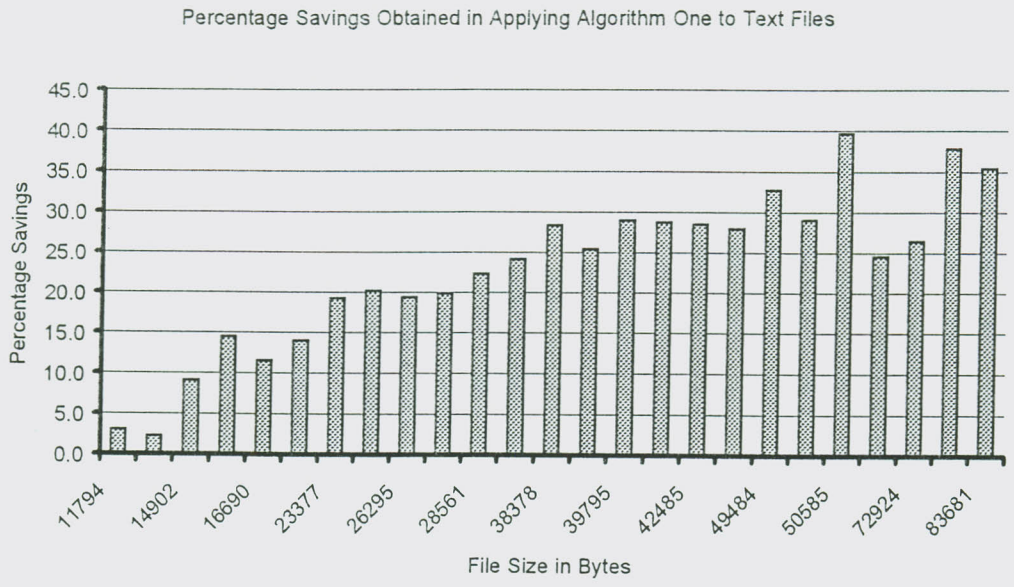


Figure 4.1 Bar graph for Algorithm One.

From the above table one can generalize that the percentage saving obtained by Algorithm One is 31.9 percent. The maximum percentage saving for this algorithm occurs for file size of 50585 bytes. On average the optimal block size for this algorithm is 50 Kilobytes.

4.1.1.2 Algorithm Two

	File size in byte	Compressed File size in byte	Percentage Savings	Average code length	Entropy	Percentage deviation of Average code length from Entropy
1	605	1095	-81.0	4.26446	4.21728	1.12
2	939	1145	-21.9	3.16933	3.14028	0.93
3	3066	2473	19.3	4.43738	4.37944	1.32
4	4378	3075	29.8	4.20786	4.16058	1.14
5	8537	6323	25.9	5.20089	5.16994	0.60
6	11794	8084	31.5	4.95913	4.92385	0.72
7	13829	9535	31.1	5.06913	5.0409	0.56
8	14902	10029	32.7	4.62281	4.58442	0.84
9	16483	10792	34.5	4.86277	4.8286	0.71
10	16690	11182	33.0	4.98933	4.95796	0.63
11	17773	11886	33.1	5.00219	4.95658	0.92
12	23377	15183	35.1	4.93151	4.89959	0.65
13	24030	15956	33.6	5.05493	5.02462	0.60
14	26295	17158	34.7	4.98505	4.9386	0.94
15	28537	19936	30.1	5.37229	5.34259	0.56
16	28561	18449	35.4	4.95126	4.9189	0.66
17	32530	20885	35.8	4.94626	4.90813	0.78
18	38378	23581	38.6	4.75439	4.72348	0.65
19	38517	24927	35.3	5.01685	4.9837	0.67
20	39795	24794	37.7	4.82902	4.7887	0.84
21	42191	26231	37.8	4.82729	4.80516	0.46
22	42485	26633	37.3	4.86948	4.83497	0.71
23	43636	27596	36.8	4.91761	4.89122	0.54
24	49484	29367	40.7	4.62281	4.58442	0.84
25	49613	31275	37.0	4.91851	4.88744	0.64
26	50585	26311	48.0	4.03889	4.0099	0.72
27	72752	51281	29.5	5.55402	5.52974	0.44
28	72924	50316	31.0	5.43508	5.41285	0.41
29	74270	42445	42.9	4.48878	4.44092	1.08
30	83681	49936	40.3	4.70006	4.67332	0.57

Table 4.2 Results of compression program for different sample text files for Algorithm Two.

Again here the negative result shows that the overall size of compressed file including the header characters is greater than the size of the original file.

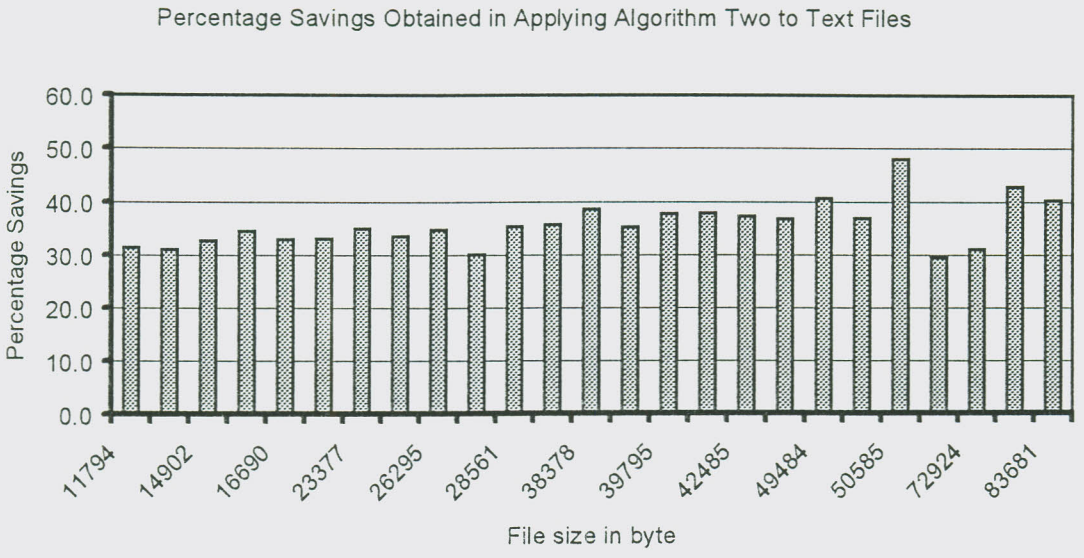


Figure 4.2 Bar graph for Algorithm Two.

From table 4.2 one can easily see that Algorithm Two applied on the same set of files resulted in better percentage savings. The average code length resulted from both algorithms is the same that is, they both have the same code words. However, the first algorithm saves the bits of its code words as characters and this makes the size of the header larger as compared to the header size of Algorithm Two. This made Algorithm One to have less percentage savings than Algorithm Two. The header size of Algorithm Two is constant and is 772 bytes.

The average percentage saving for Algorithm Two is 37 percent. The maximum percentage saving for this algorithm occurs for file size of 50585 bytes. The optimal block size for this algorithm is again 50 Kilobytes.

4.1.2 Word Processor Documents

Size of file in bytes	*Compressed File Size in bytes*	*Percentage Savings*	*Average Code Length*	*Entropy*
10752	7856	27	5.27037	5.2312
19456	4775	75	1.64556	1.2499
24576	7765	68	2.2762	2.0505
29184	8172	72	2.02844	1.7479
31232	10795	65	2.56733	2.4075
32768	13824	58	3.18631	3.1103
35328	13797	61	2.94928	2.8382
35328	15824	55	3.40832	3.3619
36352	15189	58	3.17267	3.0917
39936	17178	56	3.40663	3.3459
40960	17778	57	3.32139	3.2656
43520	18892	57	3.33084	3.2715
44544	19853	55	3.42679	3.354
89088	47747	46	4.21822	4.1978
101888	45476	53	3.72215	3.6928
101888	48178	53	3.72215	3.6928
219648	106603	51	3.85456	3.83
359136	21836	41	4.74613	4.7094
422912	206729	51	3.89597	3.8762
1287168	836473	35	5.19404	5.1624

Table 4.3 Results of compression program for different sample word processor files

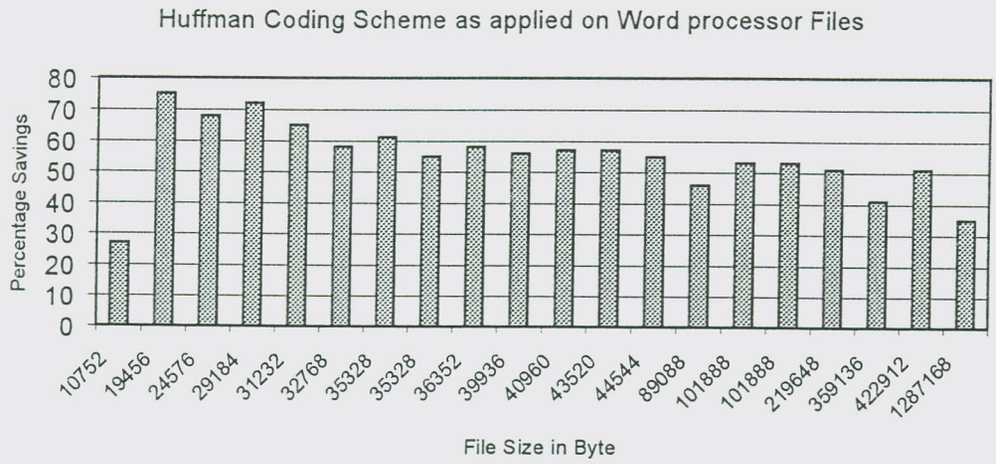


Figure 4.3 Bar graph obtained for word processor documents

It can be easily seen that the average percentage savings obtained for word processor documents is 50 percent. The maximum percentage saving is for block size of 29184 bytes.

The optimal block size for word processor documents is about 35 Kilobytes. As can be seen from the above table applying Huffman coding to word processor documents is advantageous.

4.2 Real Time Text Transmission Program

The developed program is tested for different text files and the following result is obtained. The modem transmission rate is at 9600bps. Two codes are used in the program. The first code is generated based upon the standard frequency of English alphabets and the second is generated by studying different text samples. For simplicity, the code generated by using the standard frequency of English alphabets shall be called *code 1* and the generated code by studying different text samples shall be called *code 2*.

	File Size in byte	Transmitted characters		Percentage savings		Information Transfer Rate Attained in bps	
		code 1	code 2	code 1	code 2	code 1	code 2
1	605	379	403	37.4	33.4	15324.54	14411.91
2	744	458	467	38.4	37.2	15594.76	15294.22
3	939	473	500	49.6	46.8	19057.93	18028.8
4	1056	665	689	37.0	34.8	15244.51	14713.5
5	2180	1552	1621	28.8	25.6	13484.54	12910.55
6	2461	1533	1603	37.7	34.9	15411.35	14738.37
7	2609	1789	1864	31.4	28.6	14000.22	13436.91
8	3728	2151	2234	42.3	40.1	16638.21	16020.05
9	4111	2524	2602	38.6	36.7	15636.13	15167.41
10	4116	2851	2945	30.7	28.4	13859.56	13417.18
11	4315	2534	2649	41.3	38.6	16347.28	15637.6
12	4636	3295	3396	28.9	26.7	13507.01	13105.3
13	5122	3481	3652	32.0	28.7	14125.6	13464.18
14	6072	4497	4636	25.9	23.6	12962.24	12573.6
15	6870	4055	4155	41.0	39.5	16264.36	15872.92
16	7309	4427	4567	39.4	37.5	15849.65	15363.78
17	8984	5468	5545	39.1	38.3	15772.93	15553.9
18	9736	6157	6234	36.8	36.0	15180.38	14992.88
19	9875	7122	7354	27.9	25.5	13310.87	12890.94
20	11794	7268	7489	38.4	36.5	15578.21	15118.49
21	11881	8004	8334	32.6	29.9	14250.07	13685.82
22	13829	9618	9973	30.5	27.9	13803.12	13311.78
23	14049	8440	8440	39.9	39.9	15979.91	15979.91
24	14902	9158	9441	38.5	36.6	15621.23	15152.97
25	16690	10513	10849	37.0	35.0	15240.56	14768.55
26	23377	14290	14813	38.9	36.6	15704.63	15150.15
27	24776	14811	15146	40.2	38.9	16058.98	15703.79
28	28760	16952	17593	41.1	38.8	16286.93	15693.51
29	28838	17914	18479	37.9	35.9	15454.1	14981.59
30	49613	31546	31968	36.4	35.6	15098.1	14898.8

Table 4.4 Results obtained in applying the real time text transmission program

The attained information transfer rate is plotted as shown below:

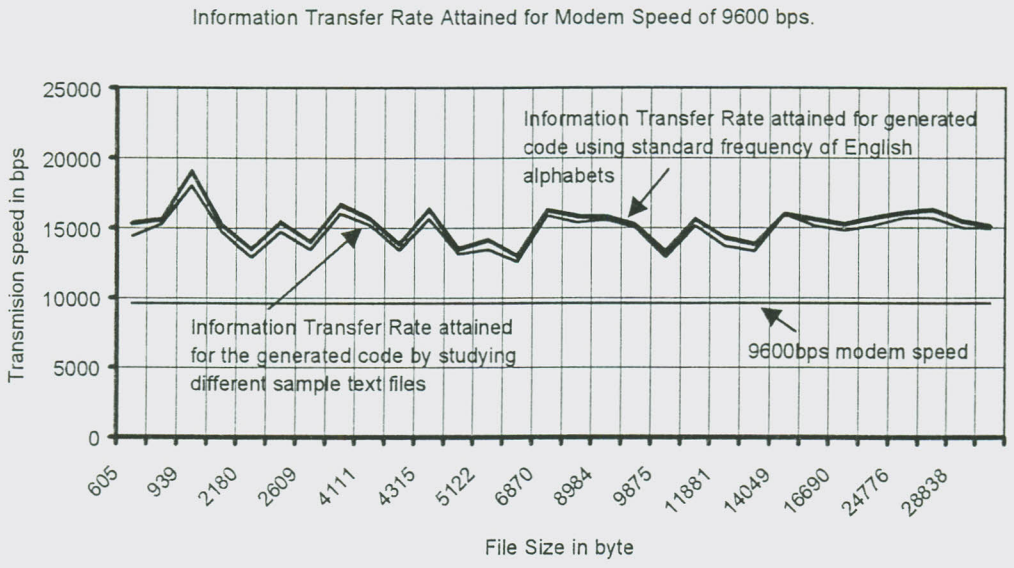


Figure 4.4 Attained information transfer rate for modem speed of 9600bps.

From table 4.4 the average information transfer rate for *code1* can be calculated to be 15221.6 bps. This shows that there is an increase of information transfer rate on average by 5621.6 bps. On the hand, for *code 2* the average information transfer rate attained is 14734.65 bps, and the increase in information transfer rate for this code is on average 5134.65 bps.

4.3 Bitmap Compressor Program

The Bitmap compressor and decompressor program was tested and the following result is obtained:

Bitmap File size in bytes	*Compressed file size in bytes*	*Percentage savings*
2118	1218	42.4
4678	2086	55.4
20278	12624	37.74
20278	10906	46.2
20278	12967	36.0
20278	10998	45.7
20278	13092	35.4
20278	11574	42.9
20278	11810	41.7
20278	10012	50.6
20278	11056	45.4
20278	11060	45.4
21238	5177	75.6
25718	10534	59.0
32850	16956	48.3
36182	19179	46.9

Table 4.5 Results obtained in applying Bitmap Compressor Program

From the above table one can see that the average percentage saving obtained for Bitmap Compressor and Decompressor Program is 45 %. Huffman coding is completely lossless and one can reconstruct the original image again.

4.4 Conclusion

The developed algorithms have been tested with different file formats and the above results are obtained. From the results of table 4.1, 4.2, and 4.3 one can see that the code generated is so optimal, that is Huffman coding scheme will always guarantees the generation of optimal code words.

The average code word length obtained is near to entropy of a given source (file) supporting the optimality of the code generated.

The Huffman code that is used in text transmission is good as can be seen from the results of table 4.4. The Information transfer rate obtained is satisfactory for text transmission. Standard frequency distribution of English alphabets resulted in a code, which has better efficiency than the code that is generated by studying different sample texts. This is because the standard frequency distribution of English alphabets is obtained by studying large number of text samples. Therefore, it is better to use the standard frequency distribution of English alphabets for the generation of the code. The frequency distribution for English alphabets is shown in Appendix 3.

If a standard code for Amharic language is developed, then one can easily study frequency distribution of Amharic alphabets for different sample texts to generate an optimal code for Amharic text transmission.

The results obtained for the Bitmap Compressor and Decompressor Program is fairly good for bitmap files whose number of color content is less than 256. For bitmap colors having more color content the compression efficiency tremendously decrease. In practice Huffman coding is applied only as the last step of compression of Images. Before Huffman coding is applied the following steps are executed as shown in the following diagram (This is a sample JPEG compression algorithm).

Figure 4.5 The basic JPEG compression algorithm

In Pixel blocking, the first step is to group pixels in an 8 x 8 pixel blocks, organized as chrominance (color) and luminance (intensity or brightness) components. When applied to a standard computer monitor image composed of red, green and blue (RGB) pixels, this results in a YUV transformation, where Y represents intensity and U and V represent color values. In YUV form the same image picture requires less storage than in RGB form; however, there is no perceptible loss of image quality.

After the pixels in an image are grouped into blocks, a discrete cosine transformation (DCT) process converts blocks of YUV pixels into sets of coefficients representing the isolated frequency components of the colors and intensities of the block. During this process each block is converted into a set of 64 coefficients- one dc coefficient and 63 ac coefficients. This process converts the pixel block from the spatial domain to the frequency domain, resulting in the most of the transformed block's energy being concentrated in the lower frequencies.

The quantization step results in the use of a table of 64 quantization values, which the 63-ac block coefficients are compared against. Since most matches are not exact but are approximations, this action results in the lossy component of JPEG.

Run-length and Huffman coding further reduces the size of the image. In data packing, bit sequences produced by the Huffman coder are grouped into bytes.

Algorithm Two is used to measure how far certain compression algorithms have achieved efficient compression in terms of data reduction. This is true because the codes generated by Huffman coding schemes are so optimal that an efficiently compressed file can not be compressed again. The following table shows the comparison made by applying Huffman coding on the compressed image resulted from two standard image codecs. These image

codecs are also used in video compression and decompression. The quality of the image used in the compression process is 0.80.

	Original Bitmap Image Size in byte	*Compression achieved by each encoder*	*Percentage saving on the compressed file after Huffman encoding is applied*
Microsoft Video 1 Codec	1,179,702	113768	11.46%
Cinpack Codec Version 1.10	1,179,702	50476	2.73%

Table 4.6 Comparisons of two video codecs found in Windows 95 Operating System.

As can be seen from the table the Cinpack Codec is superior to the Microsoft Video Codec in that the compressed file resulting from the Microsft Video Codec is more compressible than the compressed file resulting from the Cinpack Codec.

Applying adaptive Huffman coding scheme can further expand this work. This type of coding scheme uses Markov process to model the sources. If the process is ergodic, the Markov structure of the source can be used to improve the encoding of the source. For each state in the Markov system, one can use the appropriate Huffman code obtained from the corresponding transition probabilities for leaving that state. Depending on how variable are the probabilities are for each state, the encoding will gain a lot or not.

A video compression and decompression program using Huffman encoding was attempted. This program was aimed to transmit real time video over a Local Area network (LAN) with and without compression. The program plays AVI video (Audio Video Interleave) file format. This program plays both the sound and the video tracks independently. The variation of the sound signal is displayed on the same screen where the video frames are displayed. If the network communication module is developed successfully, it is possible to examine the

performance of Algorithm Two in video communication. Adaptive Huffman coding scheme can also be applied in this area for further expansion of this work.

APPENDIX 1

Windows Communication Functions Used In the Development of the Text Transmission Program [7]

Windows has over 1000 *32-bit* functions called *win-32* functions for short. Among these functions there are function dedicated for communication.

The following explanation assumes that the reader knows C programming language

1. Opening a serial port

A serial port is opened using the following function

```
HANDLE CreateFile(LPCTSTR lpFileName, DWORD dwDesiredAccess,
                  DWORD dwShareMode,
                  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                  DWORD dwCreationDistribution,
                  DWORD dwFlagsAndAttributes,
                  HANDLE hTemplateFile );
```

This function is not only used to open a serial port but also used to open or create files. For serial port the arguments of the above function will be set as discussed below.

lpFileName : Points to a null-terminated string that specifies the name of the object (file, pipe, mailslot, communications resource, disk device, console, or directory) to create or open. For serial port this name is set to 'COM1', 'COM2', or 'COM*n*' where 'COM1', 'COM2', or 'COM*n*' refers to Communication port 1(Comm1 or serial port 1), Comm2 or Comm*n* if the computer has *n* communication ports.

dwDesiredAccess: Specifies the type of access to the object. An application can obtain read access, write access, read-write access, or device query access. This argument can be set to one of the following:

0	Specifies device query access to the object. An application can query device attributes without accessing the device.
GENERIC_READ	Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with GENERIC_WRITE for read-write access
GENERIC_WRITE	Spcifies write acesss to the object. Data can be written to the file and the file pointer can be moved. Combine with GENERIC_READ for read-write access.

For serial port GENERIC_READ | GENERIC_WRITE combination can be used for writing and reading from the serial port. This is necessary because both operations are necessary for full duplex communication.

dwShareMode: Set of bit flags that specifies how the object can be shared. If *dwShareMode* is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed. Since serial ports are not sharable *dwShareMode* set to zero.

lpSecurityAttributes: Pointer to a SECURITY_ATTRIBUTES structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited. For serial port this is set to NULL assuming that it is not inherited by child process.

dwCreationDistribution: Specifies which action to take on files that exist, and which action to take when files do not exist. Since serial ports are already existing this argument is set to OPEN_EXISTING.

dwFlagsAndAttributes: Specifies the file attributes and flags for the file. For serial port this argument is set to FILE_ATTRIBUTE_NORMAL.

hTemplateFile: Specifies a handle with GENERIC_READ access to a template file. The template file supplies file attributes and extended attributes for the file being created. Under Windows 95 this value must be NULL.

If the function succeeds, the return value is an open handle to the specified file. If the function fails, the return value is INVALID_HANDLE_VALUE.

2. Closing a serial port

After communication is over the serial port has to be closed before the application quits. If the serial port is left open other applications may not be able to use the opened serial port. A function for closing a serial port is:

```
BOOL CloseHandle( HANDLE hObject );
```

hObject : Identifies an open object handle. This handle is obtained from the return value of the *CreateFile* function if it opened the serial port successfully.

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

3. Configuring a serial port

A serial port has to be configured to control the serial communication. For example it may be necessary to increase the speed of the serial port. The following function is used to configure the serial port:

```
BOOL SetCommState (HANDLE hFile, LPDCB lpDCB );
```

The *SetCommState* function configures a communications device according to the specifications in a device-control block (a DCB structure). The function reinitializes all hardware and control settings, but it does not empty output or input queues.

hFile: Identifies the communications device. The CreateFile function returns this handle.

lpDCB: Points to a DCB structure containing the configuration information for the specified communications device.

Under Windows 95 the DCB structure is as shown below:

The DCB structure defines the control setting for a serial communications device.

```
typedef struct _DCB { // dcb
    DWORD DCBlength;           // sizeof(DCB)
    DWORD BaudRate;            // current baud rate
    DWORD fBinary: 1;          // binary mode, no EOF check
    DWORD fParity: 1;          // enable parity checking
    DWORD fOutxCtsFlow:1;      // CTS output flow control
    DWORD fOutxDsrFlow:1;      // DSR output flow control
    DWORD fDtrControl:2;       // DTR flow control type
    DWORD fDsrSensitivity:1;   // DSR sensitivity
    DWORD fTXContinueOnXoff:1; // XOFF continues Tx
    DWORD fOutX: 1;            // XON/XOFF out flow control
    DWORD fInX: 1;             // XON/XOFF in flow control
    DWORD fErrorChar: 1;       // enable error replacement
    DWORD fNull: 1;            // enable null stripping
    DWORD fRtsControl:2;       // RTS flow control
    DWORD fAbortOnError:1;     // abort reads/writes on error
    DWORD fDummy2:17;          // reserved
    WORD wReserved;            // not currently used
    WORD XonLim;               // transmit XON threshold
    WORD XoffLim;              // transmit XOFF threshold
    BYTE ByteSize;             // number of bits/byte, 4-8
    BYTE Parity;               // 0-4=no,odd,even,mark,space
    BYTE StopBits;             // 0,1,2 = 1, 1.5, 2
    char XonChar;              // Tx and Rx XON character
    char XoffChar;             // Tx and Rx XOFF character
    char ErrorChar;            // error replacement character
    char EofChar;              // end of input character
    char EvtChar;              // received event character
    WORD wReserved1;           // reserved; do not use
} DCB;
```

The BaudRate member an actual baud rate value, or one of the following baud rate indexes where:

Indexes	*Actual speed in bps*
CBR_110	110
CBR_300	300
CBR_600	600
CBR_1200	1200
CBR_2400	2400
CBR_4800	4800
CBR_9600	9600
CBR_14400	14400
CBR_19200	19200
CBR_38400	38400
CBR_56000	56000
CBR_57600	57600
CBR_115200	115200
CBR_128000	128000
CBR_256000	256000

Table A1 The indexes and actual speeds in bits per second.

The configuration for a serial port can also be retrieved by the following function

```
BOOL GetCommState( HANDLE hFile, LPDCB lpDCB );
```

The *GetCommState* function fills in a device-control block (a DCB structure) with the current control settings for a specified communications device.

hFile: Identifies the communications device. The *CreateFile* function returns this handle.

lpDCB: Points to the DCB structure in which the control settings information is returned.

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

4. Handshaking and Flow Control [7]

Most of the members of the DCB structure are easy to understand, but the members that specify handshaking and flow control can be confusing. Let's start with the *fDtrControl* member, which is used to turn DTR (*Data Terminal ready*) ON or OFF or to specify DTR handshaking. When the port is used with modem, turning DTR ON tells the modem that the port is ready to receive bytes. DTR acts as an enable signal so that when DTR is turned OFF the modem stops sending bytes to the port. The port driver thus turns DTR OFF when the receive buffer gets close to full, suspending reception until the receive buffer is serviced. *fDtrControl* can be used with *fOutxDsrFlow* to control the transmission and reception of bytes to and from a port. The following figure illustrates the transmission flow control of the above process that can result from the following code:

```
fOutDsFlow = TRUE;
fDtrControl = DTR_CONTROL_HANDSHAKE;
```

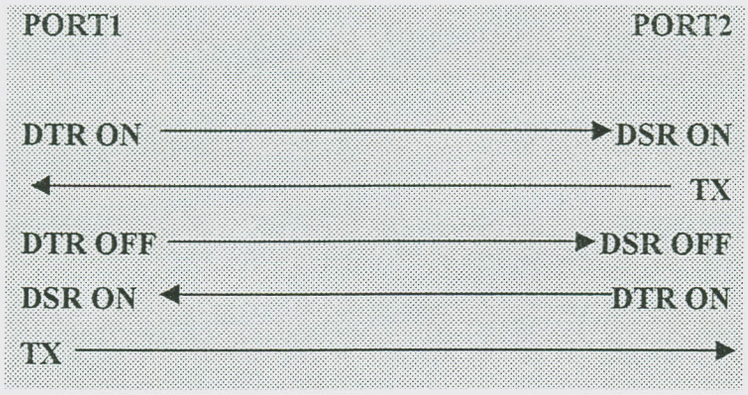


Figure A Transmission flow between two ports

If DTR handshaking is used, *fDsrSensitivity* should be set to TRUE. This way, bytes received when DSR is OFF are ignored.

Many members of the DCB structure relate to XON/XOFF flow control. This flow control uses special *embedded signals* to control the transmission and reception of bytes between two

ports. Embedded signals are special characters used *with in the data stream* to tell the port when to transmit and receive. For example, when the receive buffer is full, you can send ASCII character 19 to tell the remote port to stop sending byte. A special character with this purpose is called XOFF. When there is once again room in receive buffer, you can send ASCII 17 to tell remote port to resume transmitting bytes. A special character with this purpose is called XON. *XoffChar* and *XonChar* specify the XON and XOFF characters.

It is possible to set the member *fOutX* to TRUE to tell the port driver to suspend transmission of bytes when the remote port sends XOFF. Transmission resumes when the remote port sends XON. Or it is also possible to set member *fInX* to TRUE to tell the port driver to send XOFF to the remote port when the number of bytes in the receive buffer exceeds a certain threshold. This threshold is specified by *XoffLim*. *XoffLim* does not actually specify the maximum number of bytes allowed in the receive buffer before XOFF is sent. Rather it specifies the minimum number of available bytes permitted to remain unfilled in the receive buffer before XOFF is sent. The maximum number of bytes permitted in the receive buffer is computed by subtracting *XoffLim* from total buffer size (*buffSize*). *XoffLim* sometimes called the *high-water mark*, is the threshold that triggers a shutoff of data transfer while the receiver buffer is read down. Setting *fInX* to TRUE also tells the port driver to send XON to resume transmission once the number of bytes in the receive buffer falls below a certain threshold, specified in *XonLim*. *XonLim* sometimes called the *low-water mark*.

When *fInX* is TRUE, XOFF is sent when the number of bytes in the receive buffer exceeds the high-water mark defined by *XoffLim*. XON is sent when the number of bytes in the receive buffer falls below the threshold value specified by *XonLim*.

When the member *fTXContinueOnXoff* is set to TRUE, transmission is not affected by the above operations. In other words, transmission continues ever after XOFF has been received. When *fTXContinueOnXoff* is set to FALSE, transmission is suspended and resumed in sync with reception.

The final type of handshaking that can be specified in the DCB is RTS/CTS (*Request to send / Clear to send*) handshaking. This is the "Classic" hardware flow control of RS-232 serial communications. Using the member *fRtsControl*, RTS can be turned ON(

RTS_CONTROL_ENABLE) or OFF (RTS_CONTROL_DISABLE), or RTS handshaking can be enabled (RTS_CONTROL_HANDSHAKE). With RTS handshaking enabled, a port turn RTS ON tells the remote port or the attached modem that it is ready to receive bytes. The port turns RTS OFF when the receive buffer is full, to tell the remote port or attached modem to stop sending bytes. The remote port or attached modem turns CTS ON to indicate that it is ready to receive bytes. When CTS is OFF, the remote port or attached modem is not prepared to receive bytes, so the port does not transmit until CTS is turned ON. With RTS/ CTS handshaking the member *fOutxCtsFlow* is set to TRUE to tell the port not to transmit any bytes when CTS is OFF.

5. Timeout Settings

A major concern facing any developer of communications application is how to handle unexpected events that occur while reading or writing data. Timeout settings let the application how long a read or write function waits before giving up.

```
typedef struct _COMMTIMEOUTS {
      DWORD ReadIntervalTimeout;
      DWORD ReadTotalTimeoutMultiplier;
      DWORD ReadTotalTimeoutConstant;
      DWORD WriteTotalTimeoutMultiplier;
      DWORD WriteTotalTimeoutConstant;
} COMMTIMEOUTS,*LPCOMMTIMEOUTS;
```

There are two types of timeouts. The first type, *internal timeout*, applies only to reading from the port. It specifies how much time can elapse between the reading of two characters. Windows starts an internal timer each time a character is received. If the timer exceeds the internal timeout before the next character arrives, the read function gives up. The second type of timeout, the *total timeout*, applies to both reads and writes from the port. This timeout is triggered when the total time required to read or write a specific number of bytes exceeds a threshold.

6. Timeout Equations

Timeout values are specified in milliseconds. Windows computes the total timeouts for reading and writing from the constant and the multiplier using the following formulas:

```
ReadTotalTimeout    =    (ReadTotalTimeoutMultiplier    *    bytes_to_read)    +
                          ReadTotalTimeoutConstant;
WriteTotalTimeout   =     (WriteTotalTimeoutMultiplier    *bytes_to_read)    +
                          WriteTotalTimeoutConstant;
```

This equation makes the total timeouts flexible tool. For example considering *WriteTotalTimeout* , the result is that if the *average* value of the time to write each byte exceeds *WriteTotalTimeoutMultiplier/ bytes_to_read*, the timeout is triggered.

7. Implementing Timeout Settings

It is possible to implement timeout settings by using the following function:

```
BOOL SetCommTimeouts(HANDLE hFile,
                     LPCOMMTIMEOUTS lpCommTimeouts );
```

The *SetCommTimeouts* function sets the time-out parameters for all read and write operations on a specified communications device.

hFile: Identifies the communications device. The *CreateFile* function returns this handle.

lpCommTimeouts: Points to a COMMTIMEOUTS structure that contains the new time-out values.

If the function succeeds, the return value is nonzero. If the function fails, the return value is zero.

8. Reading and writing data to a port

The following function is used for writing data to a serial port.

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer,
               DWORD nNumberOfBytesToWrite,
               LPDWORD lpNumberOfBytesWritten,
               LPOVERLAPPED lpOverlapped );
```

This function writes data to a file and is designed for both synchronous and asynchronous operation. The function starts writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with FILE_FLAG_OVERLAPPED. If the file handle was created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the write operation is finished.

hFile: Identifies the file to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

lpBuffer: Points to the buffer containing the data to be written to the file.

nNumberOfBytesToWrite: Specifies the number of bytes to write to the file.

lpNumberOfBytesWritten: Points to the number of bytes written by this function call. *WriteFile* sets this value to zero before doing any work or error checking. If *lpOverlapped* is NULL, *lpNumberOfBytesWritten* cannot be NULL.

pOverlapped: Points to an OVERLAPPED structure. This structure is required if *hFile* was opened with FILE_FLAG_OVERLAPPED.

The following function is used for reading data from serial port.

```
BOOL ReadFile( HANDLE hFile, LPVOID lpBuffer,
               DWORD nNumberOfBytesToRead,
               LPDWORD lpNumberOfBytesRead,
               LPOVERLAPPED lpOverlapped);
```

The *ReadFile* function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the read operation.

hFile: Identifies the file to be read. The file handle must have been created with GENERIC_READ access to the file.

lpBuffer: Points to the buffer that receives the data read from the file.

nNumberOfBytesToRead: Specifies the number of bytes to be read from the file.

lpNumberOfBytesRead: Points to the number of bytes read. *ReadFile* sets this value to zero before doing any work or error checking. If this parameter is zero when *ReadFile* returns TRUE on a named pipe, the other end of the message-mode pipe called the *WriteFile* function with *nNumberOfBytesToWrite* set to zero. If *lpOverlapped* is NULL, *lpNumberOfBytesRead* cannot be NULL.

lpOverlapped: Points to an OVERLAPPED structure. This structure is required if *hFile* was created with FILE_FLAG_OVERLAPPED.

If the function succeeds, the return value is nonzero.

APPENDIX 2

The Bitmap Graphics File Format

This topic describes the graphics-file formats used by the Windows operating system. Graphics files include bitmap files, icon-resource files, and cursor-resource files.

1. Bitmap File Formats

Windows bitmap files are stored in a device-independent bitmap (DIB) format that allows Windows to display the bitmap on any type of display device. The term "device independent" means that the bitmap specifies pixel color in a form independent of the method used by a display to represent color. The default filename extension of a Windows DIB file is .BMP.

2. Bitmap File Structures

Each bitmap file contains a bitmap-file header, a bitmap-information header, a color table, and an array of bytes that defines the bitmap bits.

The bitmap-file header contains information about the type, size, and layout of a device-independent bitmap file. The header is defined as a BITMAPFILEHEADER structure.

The bitmap-information header, defined as a BITMAPINFOHEADER structure, specifies the dimensions, compression type, and color format for the bitmap.

The color table, defined as an array of RGBQUAD structures, contains as many elements as there are colors in the bitmap. The color table is not present for bitmaps with 24 color bits because each pixel is represented by 24-bit red-green-blue (RGB) values in the actual bitmap data area. The colors in the table should appear in order of importance. This helps a display driver render a bitmap on a device that cannot display as many colors as there are in the bitmap. If the DIB is in Windows version 3.0 or later format, the driver can use the *biClrImportant* member of the BITMAPINFOHEADER structure to determine which colors are important.

The BITMAPINFO structure can be used to represent a combined bitmap-information header and color table.

The bitmap bits, immediately following the color table, consist of an array of BYTE values representing consecutive rows, or "scan lines," of the bitmap. Each scan line consists of consecutive bytes representing the pixels in the scan line, in left-to-right order. The number of bytes representing a scan line depends on the color format and the width, in pixels, of the bitmap. If necessary, a scan line must be zero-padded to end on a 32-bit boundary. However, segment boundaries can appear anywhere in the bitmap. The scan lines in the bitmap are stored from bottom up. This means that the first byte in the array represents the pixels in the lower-left corner of the bitmap and the last byte represents the pixels in the upper-right corner.

The *biBitCount* member of the BITMAPINFOHEADER structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. These members can have any of the following values:

Value	Meaning
1	Bitmap is monochrome and the color table contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the color table. If the bit is set, the pixel has the color of the second entry in the table.

4	Bitmap has a maximum of 16 colors. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	Bitmap has a maximum of 256 colors. Each pixel in the bitmap is represented by a 1-byte index into the color table. For example, if the first byte in the bitmap is 0x1F, the first pixel has the color of the thirty-second table entry.
24	Bitmap has a maximum of 2^24 colors. The *bmiColors* (or *bmciColors*) member is NULL, and each 3-byte sequence in the bitmap array represents the relative intensities of red, green, and blue, respectively, for a pixel.

The *biClrUsed* member of the TBITMAPINFOHEADER structure specifies the number of color indexes in the color table actually used by the bitmap. If the *biClrUsed* member is set to zero, the bitmap uses the maximum number of colors corresponding to the value of the *biBitCount* member.

An alternative form of bitmap file uses the BITMAPCOREINFO, BITMAPCOREHEADER, and RGBTRIPLE structures.

3. Bitmap Compression

Windows versions 3.0 and later support run-length encoded (RLE) formats for compressing bitmaps that use 4 bits per pixel and 8 bits per pixel. Compression reduces the disk and memory storage required for a bitmap.

4. Compression of 8-Bits per Pixel Bitmaps

When the *biCompression* member of the BITMAPINFOHEADER structure is set to BI_RLE8, the DIB is compressed using a *run-length encoded* format for a 256-color bitmap.

This format uses two modes: encoded mode and absolute mode. Both modes can occur anywhere throughout a single bitmap.

Encoded Mode

A unit of information in encoded mode consists of two bytes. The first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte.

The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the second byte of the pair, which must be in the range 0x00 through 0x02. Following are the meanings of the escape values that can be used in the second byte:

Second byte	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The two bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

Absolute Mode

Absolute mode is signaled by the first byte in the pair being set to zero and the second byte to a value between 0x03 and 0xFF. The second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. Each run must be aligned on a word boundary.

Following is an example of an 8-bit RLE bitmap (the two-digit hexadecimal values in the second column represent a color index for a single pixel):

Compressed data	Expanded data
03 04	04 04 04
05 06	06 06 06 06 06
00 03 45 56 67 00	45 56 67
02 78	78 78
00 02 05 01	Move 5 right and 1 down
02 78	78 78
00 00	End of line
09 1E	1E 1E 1E 1E 1E 1E 1E 1E 1E
00 01	End of RLE bitmap

5. Compression of 4-Bits per Pixel Bitmaps

When the *biCompression* member of the BITMAPINFOHEADER structure is set to BI_RLE4, the DIB is compressed using a run-length encoded format for a 16-color bitmap. This format uses two modes: encoded mode and absolute mode.

Encoded Mode

A unit of information in encoded mode consists of two bytes. The first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte.

The second byte contains two color indexes, one in its high-order nibble (that is, its low-order 4 bits) and one in its low-order nibble. The first pixel is drawn using the color specified by the high-order nibble, the second is drawn using the color in the low-order nibble, the third is drawn with the color in the high-order nibble, and so on, until all the pixels specified by the first byte have been drawn.

The first byte of the pair can be set to zero to indicate an escape that denotes the end of a line, the end of the bitmap, or a delta. The interpretation of the escape depends on the value of the

second byte of the pair. In encoded mode, the second byte has a value in the range 0x00 through 0x02. The meaning of these values is the same as for a DIB with 8 bits per pixel.

Absolute Mode

In absolute mode, the first byte contains zero, the second byte contains the number of color indexes that follow, and subsequent bytes contain color indexes in their high- and low-order nibbles, one color index for each pixel. Each run must be aligned on a word boundary.
Following is an example of a 4-bit RLE bitmap (the one-digit hexadecimal values in the second column represent a color index for a single pixel):

Compressed data	Expanded data
03 04	0 4 0
05 06	0 6 0 6 0
00 06 45 56 67 00	4 5 5 6 6 7
04 78	7 8 7 8
00 02 05 01	Move 5 right and 1 down
04 78	7 8 7 8
00 00	End of line
09 1E	1 E 1 E 1 E 1 E 1
00 01	End of RLE bitmap

6. BITMAPFILEHEADER

The BITMAPFILEHEADER structure contains information about the type, size, and layout of a file that contains a device-independent bitmap (DIB).

```
typedef struct tagBITMAPFILEHEADER { // bmfh
        WORD    bfType;
        DWORD   bfSize;
        WORD    bfReserved1;
        WORD    bfReserved2;
        DWORD   bfOffBits;
} BITMAPFILEHEADER;
```

Members

bfType Specifies the file type. It must be BM.

bfSize Specifies the size, in bytes, of the bitmap file.

bfReserved1 Reserved; must be zero.

bfReserved2 Reserved; must be zero.

bfOffBits Specifies the offset, in bytes, from the BITMAPFILEHEADER structure to the bitmap bits.

Remarks

A BITMAPINFO or BITMAPCOREINFO structure immediately follows the BITMAPFILEHEADER structure in the DIB file.

7. BITMAPINFOHEADER

The BITMAPINFOHEADER structure contains information about the dimensions and color format of a device-independent bitmap (DIB).

```
typedef struct tagBITMAPINFOHEADER{ // bmih
   DWORD  biSize;
   LONG   biWidth;
   LONG   biHeight;
   WORD   biPlanes;
   WORD   biBitCount
   DWORD  biCompression;
   DWORD  biSizeImage;
   LONG   biXPelsPerMeter;
   LONG   biYPelsPerMeter;
   DWORD  biClrUsed;
   DWORD  biClrImportant;
} BITMAPINFOHEADER;
```

Members

biSize Specifies the number of bytes required by the structure.

biWidth Specifies the width of the bitmap, in pixels.

biHeight Specifies the height of the bitmap, in pixels. If *biHeight* is positive, the bitmap is a bottom-up DIB and its origin is the lower left corner. If *biHeight* is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

biPlanes Specifies the number of planes for the target device. This value must be set to 1.

biBitCount Specifies the number of bits per pixel. This value must be 1, 4, 8, 16, 24, or 32.

biCompression Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed).

BI_BITFIELDS Specifies that the bitmap is not compressed and that the color table consists of three double word color masks that specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16- and 32-bits-per-pixel bitmaps.

biSizeImage: Specifies the size, in bytes, of the image. This may be set to 0 for BI_RGB bitmaps.

biXPelsPerMeter: Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter: Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

biClrUsed: Specifies the number of color indices in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the *biBitCount* member for the compression mode specified by *biCompression*.

If *biClrUsed* is nonzero and the *biBitCount* member is less than 16, the *biClrUsed* member specifies the actual number of colors the graphics engine or device driver accesses. If *biBitCount* is 16 or greater, then *biClrUsed* member specifies the size of the color table used

to optimize performance of Windows color palettes. If *biBitCount* equals 16 or 32, the optimal color palette starts immediately following the three double word masks.

If the bitmap is a packed bitmap (a bitmap in which the bitmap array immediately follows the BITMAPINFO header and which is referenced by a single pointer), the *biClrUsed* member must be either 0 or the actual size of the color table.

biClrImportant: Specifies the number of color indices that are considered important for displaying the bitmap. If this value is zero, all colors are important.

Remarks

The BITMAPINFO structure combines the BITMAPINFOHEADER structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about DIBs, see the description of the BITMAPINFO data structure.

APPENDIX 3

Standard Frequency Distribution of English Alphabets

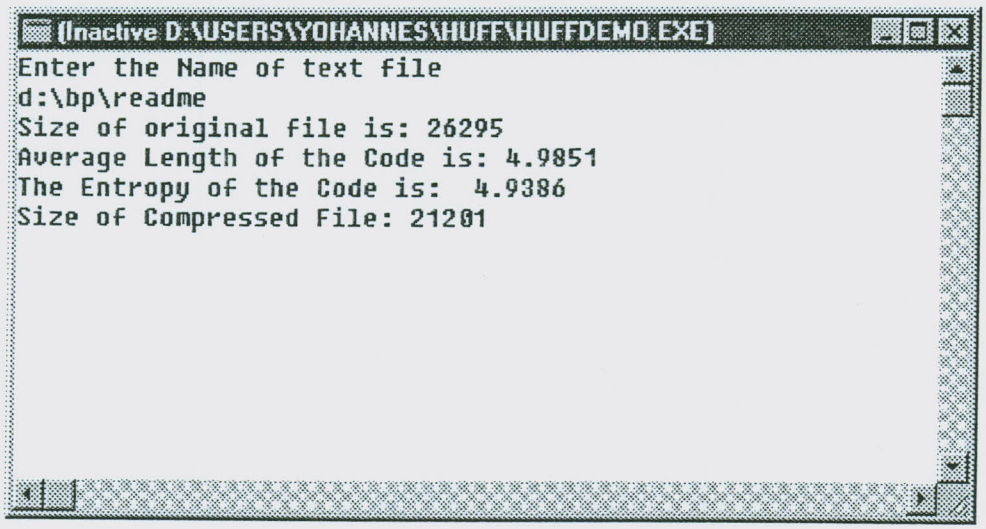
Source: H. Beker and F. Piper, Cipher Systems, Wiley-Interscience, 1982

Alphabet	*Percentage occurrence in a text document*
A	8.167
B	1.492
C	2.782
D	4.253
E	12.702
F	2.228
G	2.015
H	6.094
I	6.996
J	0.153
K	0.772
L	4.025
M	2.406
N	6.749
O	7.507
P	1.929
Q	0.095
R	5.987
S	6.327
T	9.056
U	2.758
V	0.978
W	2.36
X	0.15
Y	1.974
Z	0.074

Table A3 Standard frequency distribution of English Alphabets.

APPENDIX 4

Text Compression and Decompression Programs using Algorithm One



1. Compression Program

```
Program Huffman_Coding_Compression;

{This program is used to compress a text file and saves
 the compressed file in another file. The program first
 determines the frequency distribution of each character
 in the text file. Then it generates the Huffman code and
 finally code the text file and pack the generated bits
 so as to save it as compressed file.

 Author: Yohannes Kassahun
 Advisor: Dr.-Ing Hailu Ayele

 }

Uses WinCrt,Strings;
Const
 AConst:array[0..7] of Byte = (128,64,32,16,8,4,2,1);
 Max = 128;
Type
  AsciiStat=Record
      Ch:Char;
      Num:Longint;
      Prob:Real;
  End;

  AsciiNonZero=Record
      Ch,Status:Char;
      Num,Node1,Node2,CurNode,Length:Longint;
  End;

  List=array[0..255] of AsciiNonZero;
  Buff=array[1..Max] of Byte;

Var
   f:File;
   f1,f2,f4:Text;
   f5:file of byte;
   Ascii:array[0..255] of AsciiStat;
   AsciiCopy:array[0..511] of AsciiNonZero;
   AsciiCopy1:List;
   i,Count,IncNode,TotalNode,Node,m,k,r:Longint;
   ch:Char;
   s:String;
   LookUpTable:Array[0..255,0..60] of Char;
   CompArray:Array[1..Max] of Char;
   Table:Buff;
   Sum,SizeOfComFile,sizeOfComFile1:LongInt;
   Count1:Word;
   Position:Integer;
   AveLen,Entropy:Real;

{ This procedure initializes the Ascii Array,which is used to hold
  the frequency distribution of each character.}

Procedure FillArray;
Var i:Longint;
Begin
  For i:= 0 to 255 do
    Begin
      Ascii[i].ch:=Char(i);
      Ascii[i].Num:=0;
    End;
End;
```

```
{This procedure gets the frequency of each character}

Procedure CountNumOfChars(Name:string);
Var i:Integer;
    Ch:Byte;
Begin
  Assign(f,Name);
  Reset(f,1);
  Repeat
    BlockRead(f, Table, SizeOf(Table), Count1);
    For i:=1 to Count1 do Begin  Inc(Ascii[Table[i]].Num); End;
  Until (Count1 = 0) or (Count1 < Max);
  Close(f);
End;

{This procedure copies characters whose frequency is not Zero}

Procedure CopyOfNonZeroChars;
Var i:Longint;
Begin
  Count := 0;
  For i:=0 to 255 do
    Begin
       If (Ascii[i].Num <> 0 ) then
          Begin
             AsciiCopy[Count].Ch:=Ascii[i].Ch;
             AsciiCopy[Count].Num:=Ascii[i].Num;
             AsciiCopy[Count].CurNode:=Count;
             AsciiCopy[count].Node1:=-1;
             AsciiCopy[Count].Node2:=-1;
             AsciiCopy[Count].Status:='N';
             Inc(Count);
          End;
    End;
  IncNode:=Count; TotalNode := Count-1;
End;

{ This procedure saves copy of characters whose
  frequency is not Zero}

Procedure CopyOfNonZeroChars1;
Var i:Longint;
Begin
  Count:=0;
  For i:= 0 to IncNode-1 do
  Begin
    If (AsciiCopy[i].Status = 'N') then
      Begin
        AsciiCopy1[Count].Ch:=AsciiCopy[i].Ch;
        AsciiCopy1[Count].Num:=AsciiCopy[i].Num;
        AsciiCopy1[Count].CurNode:=AsciiCopy[i].CurNode;
        AsciiCopy1[Count].Node1:=AsciiCopy[i].Node1;
        AsciiCopy1[Count].Node2:=AsciiCopy[i].Node2;
        AsciiCopy1[Count].Status:=AsciiCopy[i].Status;
        Inc(Count);
      End;
  End;
End;


{ This is a Quick Sort procedure.}

Procedure QuickSort(var A: List; Lo, Hi: Longint);

Procedure Sort(l, r: Longint);
Var
  i, j, x, y,Node1,CurNode,Node2: Longint;
  Ch,Status:Char;
Begin
  i := l; j := r; x := a[(l+r) DIV 2].Num;
  Repeat
    While a[i].Num < x do i := i + 1;
    While x < a[j].Num do j := j - 1;
    If i <= j then
    Begin
```

```
      y := a[i].Num; a[i].Num := a[j].Num; a[j].Num := y;
      Ch := a[i].Ch; a[i].Ch := a[j].Ch; a[j].Ch := Ch;
      Node1 := a[i].Node1; a[i].Node1 := a[j].Node1; a[j].Node1 := Node1;
      Node2 := a[i].Node2; a[i].Node2 := a[j].Node2; a[j].Node2 := Node2;
      CurNode := a[i].CurNode; a[i].CurNode := a[j].CurNode;
        a[j].CurNode := CurNode;
      i := i + 1; j := j - 1;
    End;
  Until i > j;
  If l < j then Sort(l, j);
  If i < r then Sort(i, r);
End;

Begin {QuickSort};
  Sort(Lo,Hi);
End;

{ This procedure combines the least probable characters until it forms a tree}

Procedure Combine;
Var i:Longint;
Begin
  While Not ((Count-2)=0) do
   Begin
     CopyOfNonZeroChars1;
     QuickSort(AsciiCopy1,0,Count-1);
     AsciiCopy[IncNode].Num:=AsciiCopy1[0].Num + AsciiCopy1[1].Num;
     AsciiCopy[IncNode].Ch:='#';
     AsciiCopy[IncNode].Node1:=AsciiCopy1[0].CurNode ;
     AsciiCopy[IncNode].Node2:=AsciiCopy1[1].CurNode ;
     AsciiCopy[IncNode].CurNode:=IncNode ;
     AsciiCopy[IncNode].Status:='N';
     AsciiCopy[AsciiCopy1[0].CurNode].Status :='C';
     AsciiCopy[AsciiCopy1[1].CurNode].Status:='C';
     Inc(IncNode);
   End;
End;

{ This function returns the minimum of two Long Integers}

Function MinFrequency(A,B:Longint):Longint;
Begin
  If AsciiCopy[A].Num > AsciiCopy[B].Num  then
       MinFrequency := AsciiCopy[B].Num
  Else MinFrequency := AsciiCopy[A].Num;
End;

{ This Procedure generates the code from the saved code tree}

Procedure Generate_Code;
Var i,j:Longint;
Begin
  Assign(f4,'Code.txt');
  ReWrite(f4);
  For i:=0 to TotalNode do
    Begin
      m:=0;
      For j:=TotalNode + 1 to IncNode-1 do
        Begin
         If ((AsciiCopy[j].Node1 = i) or (AsciiCopy[j].Node2 = i)) then
          Begin
              r:=j;
              If (AsciiCopy[j].Node1 = i) then
                 Begin
                    Write(f4,AsciiCopy[AsciiCopy[j].Node1].Ch,#9);
                    Node:=AsciiCopy[j].Node1;
                 End
              Else
                 Begin
                   Write(f4,AsciiCopy[AsciiCopy[j].Node2].Ch,#9);
                   Node:=AsciiCopy[j].Node2;
                 End;


             If (AsciiCopy[AsciiCopy[j].Node1].Num = AsciiCopy[AsciiCopy[j].Node2].Num) then
             Begin
               If AsciiCopy[j].Node1 =Node then
```

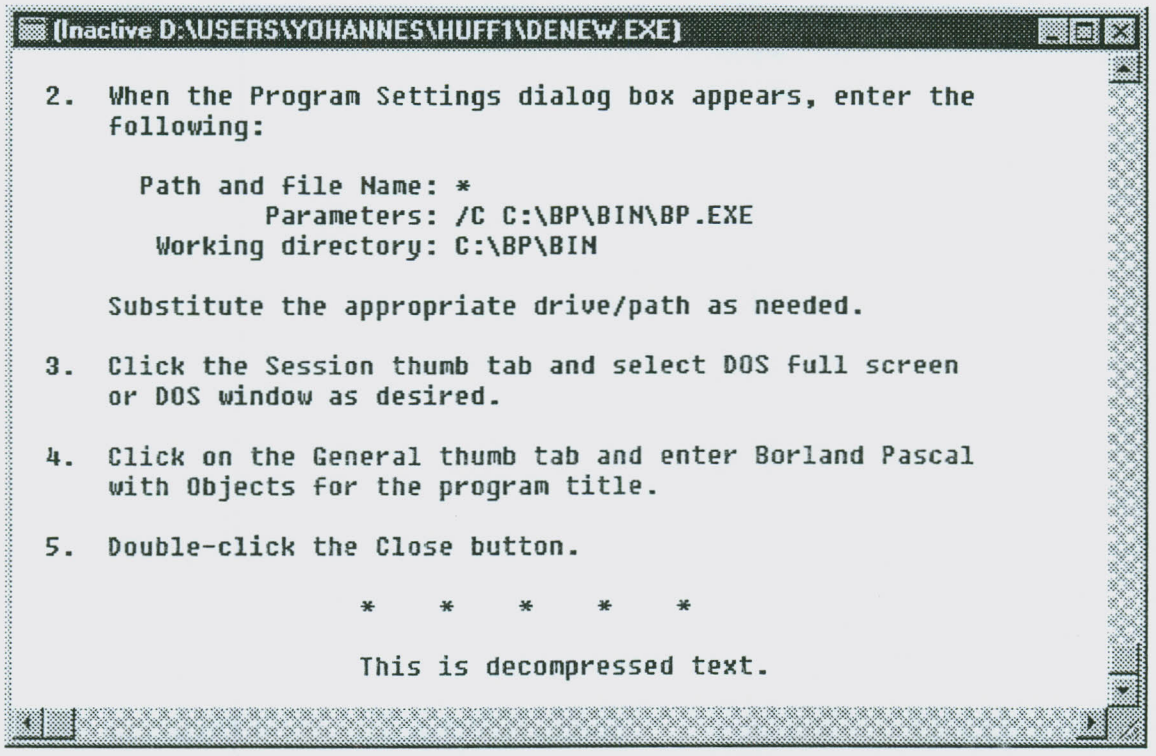
```
                    Begin
                      Write(f4,1);LookUpTable[i,m]:='1';
                    End
                  Else Begin Write(f4,0);LookUpTable[i,m]:='0'; End;
            End
         Else
            Begin
               If MinFrequency(AsciiCopy[j].Node1,AsciiCopy[j].Node2)=
                                    AsciiCopy[Node].Num) then
                  Begin Write(f4,0); LookUpTable[i,m]:='0' End
               Else
                  Begin Write(f4,1); LookUpTable[i,m]:='1' End;
            End;
         Repeat
              For k:=r+1 to IncNode - 1 do
               Begin
                  If ((AsciiCopy[k].Node1 = r) or
                           (AsciiCopy[k].Node2 = r)) then
                    Begin
                      Inc(m);
                      If (AsciiCopy[k].Node1 =
                                 AsciiCopy[r].CurNode) then
                          Begin
                           Node:=AsciiCopy[k].Node1;
                          End
                       Else
                         Begin
                            Node:=AsciiCopy[k].Node2;
                         End;
                        If (AsciiCopy[AsciiCopy[k].Node1].Num =
                                  AsciiCopy[AsciiCopy[k].Node2].Num) then
                             Begin
                                If AsciiCopy[k].Node1 =Node then
                                           Begin
                                             Write(f4,1);LookUpTable[i,m]:='1'
                                           End
                                   Else
                                           Begin
                                             Write(f4,0);LookUpTable[i,m]:='0'
                                           End;
                             End
                        Else
                           Begin
                              If (MinFrequency(AsciiCopy[k].Node1,
                                           AsciiCopy[k].Node2)
                                           =AsciiCopy[Node].Num) then
                                   Begin
                                         Write(f4,0);
                                         LookUpTable[i,m]:='0'
                                   End
                               Else
                                   Begin
                                         Write(f4,1);
                                         LookUpTable[i,m]:='1'
                                   End;
                           End;
                      r:=k;
                    End;
               End;

           Until (r < IncNode);
        End;
      End;
    WriteLn(f4);
    Inc(m); AsciiCopy[i].Length := m;LookUpTable[i,m]:='2';
    End;
  Close(f4);
End;
```

```
{ This procedure reverses the code to make it a perfix condition code}

Procedure ReverseCode;
Var i,m,j:Integer;
    Temp:Array[0..30] of Char;
Begin
  For i:=0 to TotalNode do
    Begin
      m:=0;
      Repeat
        Temp[m]:=LookUpTable[i,m];
        Inc(m);
      Until LookUpTable[i,m] = '2';
      For j:=m-1 downto 0 do
        Begin
          LookUpTable[i,m-1-j]:=Temp[j];
        End;
    End;
End;

{ This function returns the index of the character in the
  buffer containing characters whose frequency is non Zero}

Function SearchForChar(Ch:Char):Byte;
Var m:Integer;
Begin
  m:=-1;
  Repeat
    Inc(m);
  Until (Ch = AsciiCopy[m].Ch);
  SearchForChar := m;
End;

{ This procedure copies the codes of a Character in the
  Compression buffer.}

Procedure Serialize(k,Pos:Integer);
Var m:Integer;
Begin
   m:=0;
   While Not (LookUpTable[k,m] = '2') do
     Begin
       CompArray[Pos]:=LookUpTable[k,m];
       Inc(m);Inc(Pos);
     End;
   Position := Pos;
End;

{ This procedure packs the bits in the compression buffer to be
  saved as characters.}

Procedure SaveCompressedFile;
Var i,m,k,Counter:Integer;
    value:Byte;
    Final :String;
    Result,Current:Word;
Begin
  i:=1;k:=1;
  While Not (i >((Position-1) div 8)*8) do
   Begin
     value := 0;
     For m:=0 to 7 do
       Begin
         If (CompArray[i+m] = '1' ) then  value := value + Aconst[m];
       End;
     Write(f2,Char(value));
     i:=i+8;
     Inc(k);
   End;
  If ((Position -1-((Position-1) div 8)*8) <> 0 ) then
    Begin
      For m:=1 to (Position-1-((Position-1) div 8)*8) do
        Begin
          CompArray[m]:=CompArray[((Position-1) div 8)*8 + m];
          Counter:=m;
```

```
        End;
      Position:=Counter+1;
    End
  Else Position:=1;
End;

{ This procedure calls procedures searchForChar, SavecompressedFile,
  and Serialize function to accomplish the compression process of
  charcters read in from the file to be compressed}

Procedure SerCoAndComP(Var A:Buff;Count:Word);
Var i,n,j,value:Integer;
Begin
  For i:=1 to Count do
    Begin
      n:=SearchForChar(Char(A[i]));
      If ((AsciiCopy[n].Length + Position-1) > Max) then  SaveCompressedFile;
      Serialize(n,Position);
    End;
End;

{This procedure performs compression}

Procedure Compress;
Var i,value,j:Integer;
Begin
  Assign(f2,'try.cmp');
  ReWrite(f2);
  { Save the decoding table }
  WriteLn(f2,'Start of Decoding Table');
  sizeOfComFile1:=0;
  Writeln(f2,TotalNode);
  For i:=0 to TotalNode do
    Begin
      m:=0;
      Write(f2,AsciiCopy[i].Ch,#9);
      Repeat
         Write(f2,LookUpTable[i,m]);
         Inc(m);
         Inc(sizeOfComFile1);
      Until LookUpTable[i,m] = '2';
      WriteLn(f2);
      sizeOfComFile1 := SizeOfComFile1 + 44;
    End;
  WriteLn(f2,'End of Decoding Table');

  Assign(f,s);
  Reset(f,1);
  Repeat
    BlockRead(f, Table, SizeOf(Table), Count1);
    SerCoAndComp(Table,Count1);
  Until (Count1 = 0) or (Count1 < Max);
  SaveCompressedFile;
  value :=0;
  For j:=1 to Position -1 do
   Begin
     If (CompArray[j] = '1') then value := value + Aconst[j-1];
   End;
  Write(f2,Char(value));
  Close(f);
  Close(f2);
End;
```

```
{Main Program}

Begin
  FillArray;
  WriteLn('Enter the Name of text file');
  ReadLn(s);
  CountNumOfChars(s);
  CopyOfNonZeroChars;
  Combine;
  Generate_Code;
  ReverseCode;
  Position :=1;
  Compress;
  Sum:=0;
  For i:=0 to TotalNode do Sum:=Sum + AsciiCopy[i].Num;
  AveLen:=0;
  Entropy:=0;
  SizeOfComFile:=0;

  {Save the code in file Code.txt and also calculate the Entropy and average
    code word length obtained}

  Assign(f1,'Code.txt');
  ReWrite(f1);
  For i:=0 to TotalNode do
    Begin
      m:=0;
      Write(f1,AsciiCopy[i].Ch,'    ',AsciiCopy[i].Num:8,
               '   ',AsciiCopy[i].Length:4,'    ');
      AveLen:=AveLen + (AsciiCopy[i].Num *AsciiCopy[i].Length)/Sum;
      Entropy :=Entropy + (AsciiCopy[i].Num /Sum) * (ln(sum/AsciiCopy[i].Num)/ln(2));
      SizeOfComFile:=SizeOfComFile + AsciiCopy[i].Length *AsciiCopy[i].Num;
      Repeat
         Write(f1,LookUpTable[i,m]);
         Inc(m);
      Until LookUpTable[i,m] = '2';
      WriteLn(f1);
    End;
  WriteLn(f1);
  Assign(f5,s);
  Reset(f5);
  WriteLn('Size of original file is: ', filesize(f5));
  WriteLn(f1,'Average Length of the Code is: ',AveLen:6:4);
  WriteLn(f1,'The Entropy of the Code is:  ',Entropy:6:4);
  WriteLn(f1,'Size of Compressed File: ',
            (SizeOfComFile div 8) +1 + sizeOfComFile1);
  Close(f1);
End.
```

Decompression Program



2. Decompression Program

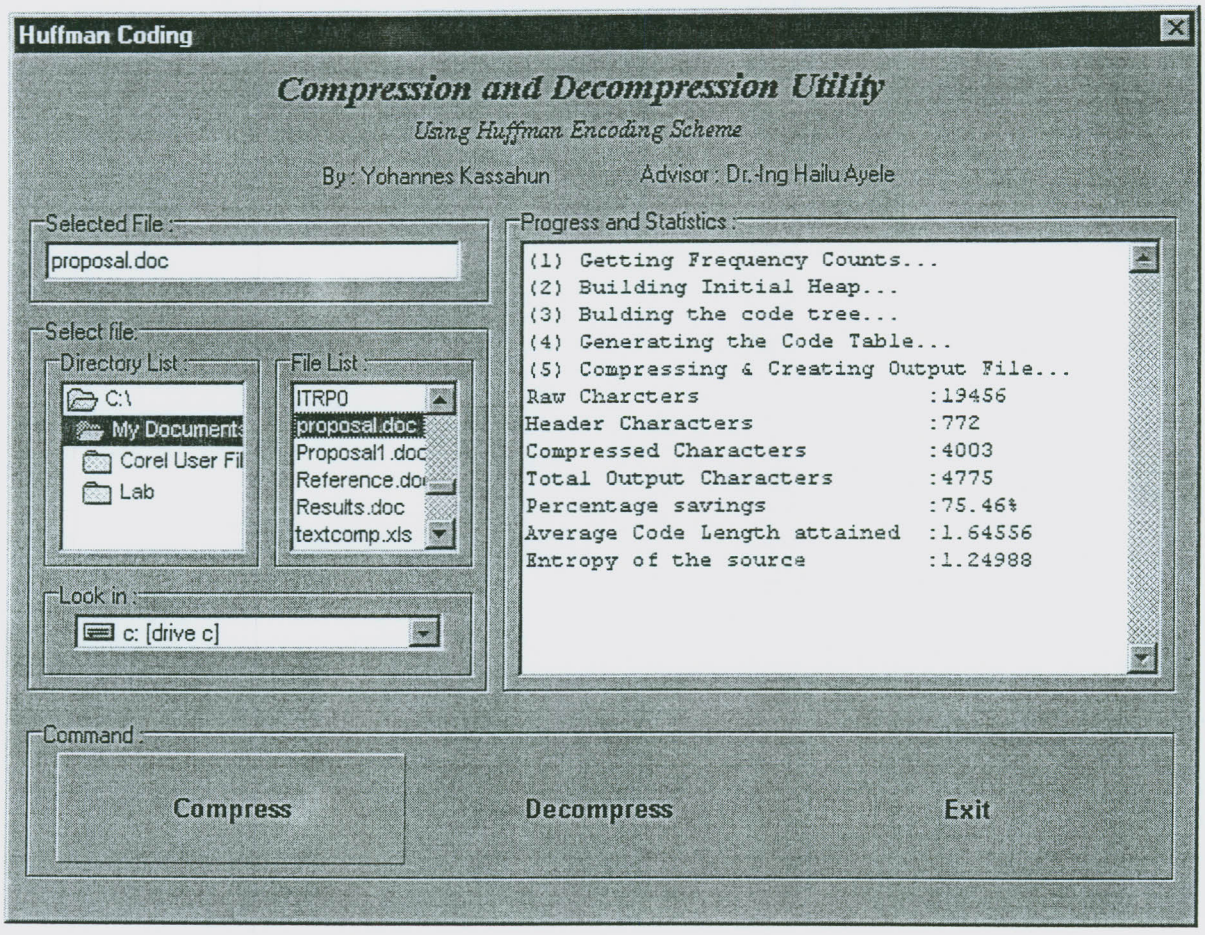
```
Program Decompress;

{This program performs decompression of compressed text files.
  Author: Yohannes Kassahun
  Advisor: Dr.-Ing Hailu Ayele
}

Uses WinCrt;
Const Max=64;
type
    AsciiRecord=Record
       Ch:Char;
       Length:Integer;
    End;

Var f:File of Byte;
    f1:Text;
    AsciiCode:array[0..255,0..30] of Char;
    Ascii:array[0..255] of AsciiRecord;
    AsciiDemmy:Array[0..255] of Char;
    DecompArray:array[1..Max] of Char;
    Temp:array[0..60] of Char;
    s:String;
    TotalNode,i,m,Return,Count:Integer;
    Ch:Char;
    B:Byte;
    Found,FoundVer:Boolean;

{This procedure reads the decoding table.}

Procedure Read_Decoding_Table;
Var i,m:Integer;
     Ch:Char;
Begin
  Count:=52;
  ReadLn(f1);
  ReadLn(f1,TotalNode);
  For i:= 0 to TotalNode do
  Begin
    Read(f1,Ascii[i].Ch);m:=0;
    While not (EOLN(f1)) do Begin Read(f1,AsciiCode[i,m]);Inc(m); End;
    Ascii[i].Length := m-1;
    AsciiCode[i,m]:='2';
    Count:=Count + m + 3;
    ReadLn(f1);
  End;
  ReadLn(f1);
  Close(f1);
End;

{This procedure converts the ord of a character to its base two equivalent}

Procedure ChangeToBinary(Ch:Char; i:integer);
Var bi,m:Byte;
Begin
  bi:= Ord(Ch);
  For m:= 7 downto 0 do
   Begin
      DecompArray[m+i]:=Char((bi mod 2) + 48);
      bi:=bi div 2;
   End;
End;
```

```
{This procedure searches for a code if it exists in the decoding table}

Procedure SearchForCode(x:Integer);
Var k,h:Integer;
Begin
   FoundVer:=False;
   For k:=0 to TotalNode do
    Begin
      Found:=True;
      If (Ascii[k].Length = x) then
        Begin
          For h:=1 to x do
            Begin
              If (AsciiCode[k,h] <> Temp[h-1]) then Found := False;
            End;
        End
      Else Found:=False;
      If Found then Begin Return:=k; FoundVer:=True; End;
    End;
End;

{This procedure performs the actual decoding process.}

Procedure TranslateCode;
var m,j:Integer;
      s:LongInt;
Begin
 m:=0;
 For j:=1 to i-1  do
  Begin
     Temp[m]:=DecompArray[j];
     SearchForCode(m+1);
     If FoundVer then Begin m:=-1;Write(Ascii[Return].Ch); End;
     Inc(m);
  End;
  If not(FoundVer) then
    Begin
      For j:=0 to m-1 do
       DecompArray[j+1]:=Temp[j];
      i:=m+1;
    End
  Else i:=1;
End;

{This procedure initializes the decoder and skips the header of the compressed text file.}

Procedure Initialize;
Var i:Integer;
    Demmy:Byte;
Begin
  For i:=1 to Count do Read(f,Demmy);
  For i:=0 to 255 do AsciiDemmy[i]:= Char(i);
End;

{This procedure calls TranslateCode to decode the read in compressed characters.}

Procedure Decoder;
Begin
  i:=1;
  While not Eof(f) do
  Begin
    Read(f,B);
    Ch:=AsciiDemmy[B];
    If ((i+8) > Max) then Begin TranslateCode  End;
    ChangeToBinary(Ch,i);
    i:=i+8;
  End;
  TranslateCode;
  Close(f);
End;
```

```
{Main Program}

Begin
  WriteLn('Enter the name of compressed text you want to decompress:');
  Read(s);
  Assign(f1,s);
  Reset(f1);
  Read_Decoding_Table;
  Assign(f,s);
  Reset(f);
  Initialize;
  Decoder;
End.
```

APPENDIX 5

File Compressor and Decompressor Program Using Algorithm Two



File compressor and decompressor Program

1. Main Form

```
unit Unit1;

{This program is a Delphi program implementing Algorithm Two for compressing and decompressing
any from of file.

Author: Yohannes Kassahun
Advisor: Dr.-Ing Hailu Ayele
}

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
  StdCtrls, FileCtrl, ExtCtrls;

type
  TForm1 = class(TForm)
    GroupBox1: TGroupBox;
    GroupBox2: TGroupBox;
    DriveComboBox1: TDriveComboBox;
    GroupBox3: TGroupBox;
    GroupBox4: TGroupBox;
    DirectoryListBox1: TDirectoryListBox;
    FileListBox1: TFileListBox;
    GroupBox5: TGroupBox;
    Edit1: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    GroupBox6: TGroupBox;
    Panel1: TPanel;
    Panel2: TPanel;
    Panel3: TPanel;
    Panel4: TPanel;
    GroupBox7: TGroupBox;
    Memo1: TMemo;
    Label3: TLabel;
    procedure Panel2MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure Panel3MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure Panel1MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure FormMouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure Panel2MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel2MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel3MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel3MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel2Click(Sender: TObject);
    procedure Panel4MouseMove(Sender: TObject; Shift: TShiftState; X,
      Y: Integer);
    procedure Panel4MouseDown(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel4MouseUp(Sender: TObject; Button: TMouseButton;
      Shift: TShiftState; X, Y: Integer);
    procedure Panel4Click(Sender: TObject);
    procedure Panel3Click(Sender: TObject);
    private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;
  Frequency_Count : Array[0..511] of LongInt;
  Heap : Array[0..256] of LongInt;
```

```
  code : Array[0..255] of LongInt; {Word;}
  code_length :Array[0..255] of Char;
  F:File of byte;
  Fo,FCheck:File;
  F1: File;
  heap_length :Word;
  father : Array[0..511] of Integer;
  Compress_charcount,File_Size :LongInt;
  I:Integer;
  Savings :Real;
  header_charcount : Word;
  output_characters : LongInt;
  decomp_tree  : Array[0..511] of Integer;

implementation

{$R *.DFM}
Uses Unit2,Unit3,Math;

{*****************************************************************************}

Compression Algorithm

{*****************************************************************************}
{This procedure performs the actual data compression}

Procedure Compress_Image(S : String);
Var
  current_code,current_length ,dvalue :LongInt{Word};
  thebyte :Word;
  loop1,curbit   :Integer;
  loop,curbyte   :LongInt;
  F      :File;
  Size,Result :Integer;
  Buff :Array[0..2000] of byte;
  OBuff:Array[0..2000] of byte;
  Counter,I :Word;
Begin
  thebyte :=0;curbyte :=0;curbit :=7;
  AssignFile(F,S);
  Reset(F,1);
  Size :=SizeOf(Buff);
  Repeat
    BlockRead(F,Buff,Size,Result);
    Counter :=0;
    For i:=0 to Result-1 do
      Begin
        dvalue := Word(Buff[i]);
        current_code := code[dvalue];
        current_length :=Word(code_length[dvalue]);
        For loop1 :=(current_length-1) downto 0 do
          Begin
            If (((current_code shr loop1) and 1) <> 0) then
              thebyte := thebyte or Byte(Char(1 shl curbit));
            Dec(curbit);
            If (curbit < 0) then
              Begin
                OBuff[Counter] :=Byte(thebyte);
                Inc(Counter);
                thebyte :=0;
                Inc(curbyte);
                curbit :=7;
              End;
          End;
      End;
    BlockWrite(Fo,OBuff,Counter);
  Until (Result < Size);
  BlockWrite(Fo,Byte(thebyte),1);
  Inc(curbyte);
  compress_charcount :=curbyte;
  CloseFile(F);
End;
```

```
{This procedure calculates the result of compression achieved}

Procedure Compression_Report;
Begin
  header_charcount := 768 + sizeof(File_Size);
  output_characters := Word(header_charcount) + compress_charcount;
  Savings := 100 - (output_characters/File_size) * 100;
End;

{This function generates the compression code table}

Function Generate_Code_Table :Word;
Var
  loop,current_length,current_bit,bit_code :LongInt{Word};
                                    Parent:Integer;
Begin
  For loop :=0 to 255 do
    If (Frequency_Count[loop] <> 0) then
      Begin
        bit_code := 0;
        current_length:=bit_code;
        current_bit :=1;
        parent := father[loop];

        while (parent <> 0) do
         Begin
           If (parent < 0) then
             Begin
               Inc(bit_code,current_bit);
               parent := -parent;
             End;
           parent := father[parent];
           current_bit := current_bit shl 1;
           Inc(current_length);
         End;
       code[loop] := bit_code;
       If (current_length > 32 {>16}) then Generate_Code_Table :=0
       Else code_length[loop] := Char(current_length);
      End
    Else
      Begin
         code_length[loop] :=#0;
         code[loop] := 0;
      End;
End;

{This procedure creates a "legal " heap from the current heap structure}

Procedure Reheap(heap_entry : Word);
Var Index :Word;
    Flag  :Word;
    heap_value :LongInt;
Begin
   Flag := 1;
   heap_value := heap[heap_entry];
   while ((heap_entry <= (heap_length shr 1)) and (Flag <> 0)) do
    Begin
      Index := heap_entry shl 1;
      If (Index < heap_length) then
       If (Frequency_Count[heap[Index]] >= Frequency_Count[heap[Index+1]])
         then Inc(Index);

      If(Frequency_Count[heap_value] < Frequency_Count[heap[Index]]) then
        Dec(Flag)
      Else
        Begin
          heap[heap_entry] := heap[Index];
          heap_entry := Index;
        End;
    End;
    heap[heap_entry] := heap_value;
End;
```

```
{This procedure builds the compression code tree}

Procedure Build_Code_Tree;
Var
  findex : Word;
  heap_value: LongInt;
Begin
  while (heap_length <> 1) do
    Begin
      heap_value := heap[1];
      heap[1] := heap[heap_length];
      Dec(heap_length);

      reheap(1);
      findex := heap_length + 255;
      Frequency_Count[findex] := Frequency_Count[heap[1]]
                                 + Frequency_Count[heap_value];
      father[heap_value] := findex;
      father[heap[1]] := -findex;
      heap[1] := findex;

      reheap(1);
    End;
  father[256] :=0;
End;

{This procedure builds a heap from initial frequency count data}

Procedure Build_Initial_Heap;
Var Loop :Word;
Begin
  heap_length :=0;

  For Loop:=0 to 255 do
   If (Frequency_Count[Loop] <> 0) then
     Begin
       Inc(heap_length);
       heap[heap_length] := LongInt(Loop);
     End;

  For Loop := heap_length downto 1 do
    reheap(loop);
End;

{This procedure counts the number of occurrences of each byte in the data
 that are to be compressed.}

Procedure Get_Frequency_Count(S:String);
Var  Loop :Word;
     Result :Integer;
     F    :File;
     Buff :Array[0..8000] of byte;
     Size :LongInt;
Begin
  AssignFile(F,S);
  Reset(F,1);
  Size :=SizeOf(Buff);
  Repeat
    BlockRead(F,Buff,Size,Result);
    For Loop:=0 to Result-1 do Inc(Frequency_count[Buff[loop]]);
  Until (Result <Size);
  CloseFile(F);
End;
```

```
{*************************************************************************}

Decompression Algorithm

{*************************************************************************}

{This procedure builds the decompression Tree}

Procedure Build_decomp_tree;
Var
   loop1,current_index,current_node : LongInt;
                               loop : Integer;
Begin
  current_node := 1;
  decomp_tree[1] := 1;
  For loop :=0 to 255 do
    Begin
      If(Byte(code_length[loop]) <> 0) then
        Begin
          current_index := 1;
          For loop1 := (Byte(code_length[loop])-1) downto 1 do
            Begin
              current_index := (decomp_tree[current_index] shl 1) +
                    ((code[loop] shr loop1) and 1);
              If ((decomp_tree[current_index]) = 0 ) then
               Begin
                 Inc(current_node);
                 decomp_tree[current_index] := current_node;
               End;
            End;
          decomp_tree[(decomp_tree[current_index] shl 1) +
                (code[loop] and 1)] := -loop;
        End;
    End;
End;

{This Procedure decompresses the compressed data.}

Procedure decompress_Image(S:String);
Var
  cindex   :LongInt{Word};
  curchar  : Array[0..500] of Byte;
  OBuff    : Array[0..3000] of Byte;
  bitshift :Integer;
  Size,Result :Integer;
  Fo          :File;
  i           :LongInt{Word};
  counter     :LongInt{Word};
  charcount   :LongInt;

Begin
  Size := SizeOf(curchar);
  Assign(Fo,S);
  ReWrite(Fo,1);
  charcount :=0;
  cindex :=1;
  Repeat
    BlockRead(F1,curchar,Size,Result);
    counter :=0;
    For i:=0 to Result-1 do
     Begin
       For bitshift := 7 downto 0 do
         Begin
           cindex := (cindex shl 1) + ((curchar[i] shr bitshift) and 1);
           If (decomp_tree[cindex] <= 0) then
             Begin
               OBuff[counter] := Byte(-decomp_tree[cindex]);
               Inc(counter);
               Inc(charcount);
               {If (charcount = File_Size) then bitshift:=0}
               {Else} cindex :=1;
             End
           Else
```

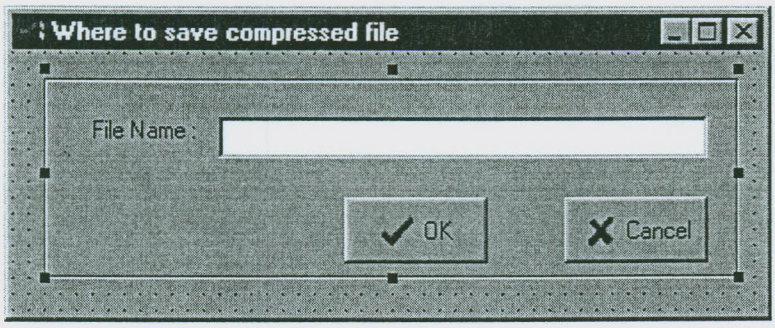
```
            cindex := decomp_tree[cindex];
          End;
      End;
     BlockWrite(Fo,OBuff,counter);
  Until (Result < Size);
  Close(Fo);
End;

{*************************************************************************}

procedure TForm1.Panel2MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
   Panel2.BevelOuter := bvRaised;
   Panel2.Font.Color := clBlue;
   Panel3.BevelOuter := bvNone;
   Panel3.Font.Color := clBlack;
   Panel4.BevelOuter := bvNone;
   Panel4.Font.Color := clBlack;
end;

procedure TForm1.Panel3MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
   Panel3.BevelOuter := bvRaised;
   Panel3.Font.Color := clBlue;
   Panel2.BevelOuter := bvNone;
   Panel2.Font.Color := clBlack;
   Panel4.BevelOuter := bvNone;
   Panel4.Font.Color := clBlack;
end;

procedure TForm1.Panel1MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
   Panel3.BevelOuter := bvNone;
   Panel2.BevelOuter := bvNone;
   Panel4.BevelOuter := bvNone;
   Panel2.Font.Color := clBlack;
   Panel3.Font.Color := clBlack;
   Panel4.Font.Color := clBlack;
end;

procedure TForm1.FormMouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
   Panel3.BevelOuter := bvNone;
   Panel2.BevelOuter := bvNone;
   Panel4.BevelOuter := bvNone;
   Panel2.Font.Color := clBlack;
   Panel3.Font.Color := clBlack;
   Panel4.Font.Color := clBlack;
end;

procedure TForm1.Panel2MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Panel2.BevelOuter := bvLowered;
end;

procedure TForm1.Panel2MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Panel2.BevelOuter := bvRaised;
end;

procedure TForm1.Panel3MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Panel3.BevelOuter := bvLowered;
end;
```

```
procedure TForm1.Panel3MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Panel3.BevelOuter := bvRaised;
end;


procedure TForm1.Panel2Click(Sender: TObject);
Var Counter,loop               : Integer;
    S,S1                       : String;
    AverageLength,Entropy      : Real;
    total                      : LongInt;
begin
   Memo1.Clear;
   Counter := 0;
   AssignFile(F,Edit1.Text);
   Reset(F);
   File_Size := FileSize(F);
   CloseFile(F);
   S:='(1) Getting Frequency Counts...' + #13 + #10;
   Memo1.Lines[Counter] := S;
   Inc(Counter);
   For loop := 0 to 255 do Frequency_Count[loop] :=0;
   Get_Frequency_Count(Edit1.Text);
   S:='(2) Building Initial Heap...' + #13 + #10;
   Memo1.Lines[Counter] := S;
   Inc(Counter);
   Build_Initial_Heap;
   S:='(3) Bulding the code tree...' + #13 + #10;
   Memo1.Lines[Counter] := S;
   Inc(Counter);
   Build_Code_Tree;
   S :='(4) Generating the Code Table...' + #13 + #10;
   Memo1.Lines[Counter] := S;
   Inc(Counter);
   If (Generate_Code_Table =0 ) then
   Begin
     S := 'Can''t Compress!!' + #13 + #10;
     Memo1.Lines[Counter] := S;
     Inc(Counter);
   End
   Else
   Begin
     S := '(5) Compressing & Creating Output File...' + #13 + #10;
     Memo1.Lines[Counter] := S;
     Inc(Counter);
     with TForm2.Create(Self) do
     try
        ShowModal;
     finally
       Free;
     end;
     If (not CancelSelected) then
     Begin
       AssignFile(fo,SEdit);
       Rewrite(Fo,1);
       BlockWrite(Fo,File_Size,SizeOf(File_Size));
       BlockWrite(Fo,code,SizeOf(code));
       BlockWrite(Fo,code_length,SizeOf(code_length));
       compress_image(Edit1.Text);
       Compression_Report;
       S := 'Raw Charcters              :'+ IntToStr(File_Size) + #13+ #10;
       Memo1.Lines[Counter] := S;
       Inc(Counter);
       S := 'Header Characters :' + IntToStr(header_charcount) + #13 + #10;
       Memo1.Lines[Counter] := S;
       Inc(Counter);
       S := 'Compressed Characters: '
                                  + IntToStr(compress_charcount)+ #13 + #10;
       Memo1.Lines[Counter] := S;
       Inc(Counter);
       S := 'Total Output Characters    :'
                                 + IntToStr(output_Characters) + #13 + #10;
       Memo1.Lines[Counter] := S;
       Inc(Counter);
```

```
        Str(Savings:4:2,S1);
        S := 'Percentage savings            :' + S1 + '%' + #13 + #10;
        Memo1.Lines[Counter] := S;
        Inc(Counter);

        {Calculation of attained average code Length}

        AverageLength := 0;
        For loop := 0 to 255 do
          Begin
            AverageLength := AverageLength +
                             Frequency_Count[loop]*Word(Code_Length[loop]);
          End;
        AverageLength := AverageLength /File_Size;
        Str (AverageLength:6:5,S1);
        S := 'Average Code Length attained  :' + S1 + #13 + #10;
        Memo1.Lines[Counter] := S;
        Inc(Counter);

        {Calculation of Entropy }

        Entropy := 0;
        For loop := 0 to 255 do
          Begin
            If (Frequency_Count[loop] <> 0) then
              Entropy := Entropy +
                             (Frequency_Count[loop]/File_Size)
                             *log2(File_Size/Frequency_Count[loop]);
           End;
        Str(Entropy:6:5,S1);
        S :='Entropy of the source          :' + S1 + #13 + #10;
        Memo1.Lines[Counter] := S;
        Inc(Counter);

        CloseFile(Fo);
      End;

    End;
end;

procedure TForm1.Panel4MouseMove(Sender: TObject; Shift: TShiftState; X,
  Y: Integer);
begin
  Panel2.BevelOuter := bvNone;
  Panel2.Font.Color := clBlack;
  Panel3.BevelOuter := bvNone;
  Panel3.Font.Color := clBlack;
  Panel4.BevelOuter := bvRaised;
  Panel4.Font.Color := clBlue;
end;

procedure TForm1.Panel4MouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
   Panel4.BevelOuter := bvLowered;
end;

procedure TForm1.Panel4MouseUp(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Integer);
begin
  Panel4.BevelOuter := bvRaised;
end;

procedure TForm1.Panel4Click(Sender: TObject);
begin
  PostQuitMessage(0);
end;

procedure TForm1.Panel3Click(Sender: TObject);
Var S        :string;
    Counter  :Integer;
begin
  Memo1.Clear;
  Counter := 0;
  AssignFile(F1,Edit1.Text);
```

```
  Reset(F1,1);
  BlockRead(F1,File_Size,SizeOf(File_Size));
  BlockRead(F1,code,SizeOf(code));
  BlockRead(F1,code_length,SizeOf(code_length));
  S := '(1) Building the tree... ' + #13 + #10;
  Memo1.Lines[Counter] := S;
  Inc(Counter);
  Build_decomp_tree;
  with TForm3.Create(Self) do
  try
   ShowModal;
  finally
   Free;
  end;

  If (not CancelSelected1) then
  Begin
    S := '(2) Decompressing and creating output file... ' + #13 + #10;
    Memo1.Lines[Counter] := S;
    Inc(Counter);
    decompress_image(SEdit1);
    S := '(3) Finshed Decompressing.' + #13 + #10;
    Memo1.Lines[Counter] := S;
    Inc(Counter);
  End;

  CloseFile(F1);
end;


end.
```

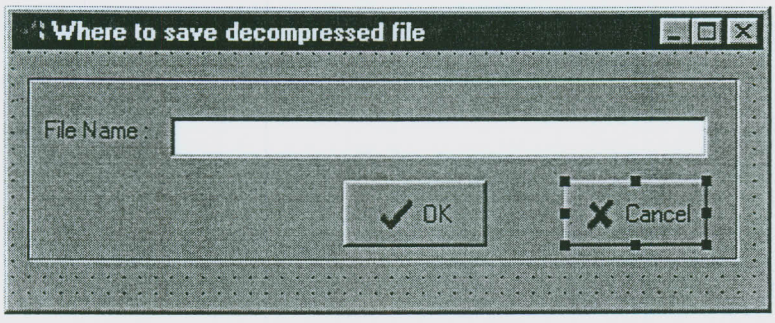
2. Save Compressed File Dialog Box



The code that handles the above dialog box is shown below

```
unit Unit2;

interface

uses    windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
        StdCtrls, Buttons;

type

  TForm2 = class(TForm)
    GroupBox1: TGroupBox;
    Edit1: TEdit;
    Label1: TLabel;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure BitBtn2Click(Sender: TObject);
    procedure BitBtn1Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form2: TForm2;
  CancelSelected : Boolean;
  SEdit : String;

implementation

{$R *.DFM}

procedure TForm2.BitBtn2Click(Sender: TObject);
begin
  CancelSelected := True;
end;

procedure TForm2.BitBtn1Click(Sender: TObject);
begin
 CancelSelected := False;
 SEdit := Edit1.Text;
end;

end.
```

3. Save Decompressed File Dialog Box

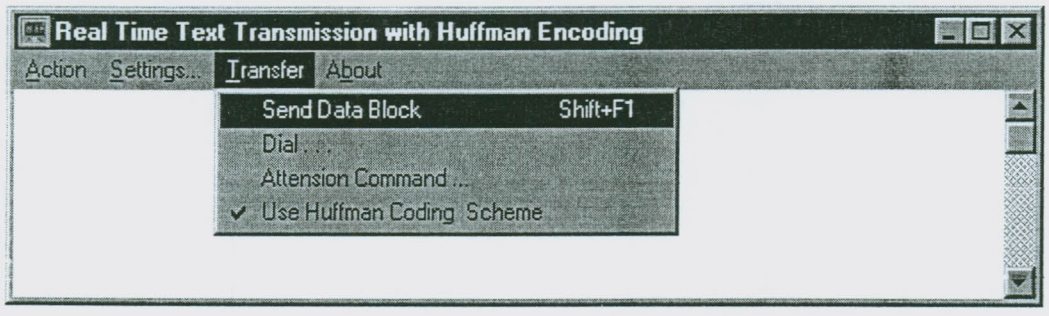


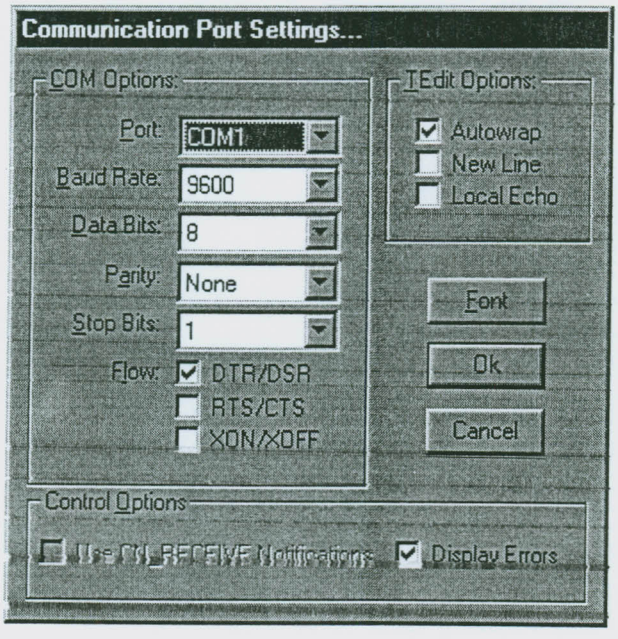
The code that handles the above dialog box is shown below

```
unit Unit3;

interface

uses   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs,
       StdCtrls, Buttons;

type
  TForm3 = class(TForm)
    GroupBox1: TGroupBox;
    Label1: TLabel;
    Edit1: TEdit;
    BitBtn1: TBitBtn;
    BitBtn2: TBitBtn;
    procedure BitBtn1Click(Sender: TObject);
    procedure BitBtn2Click(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form3           : TForm3;
  SEdit1          : String;
  CancelSelected1 : Boolean;

implementation

{$R *.DFM}

procedure TForm3.BitBtn1Click(Sender: TObject);
begin
   CancelSelected1 := False;
   SEdit1 := Edit1.Text;
end;

procedure TForm3.BitBtn2Click(Sender: TObject);
begin
  CancelSelected1 := True;
end;

end.
```

APPENDIX 6

Real Time Text Transmission Program With and Without Huffman Encoding Scheme





1. Header File

```
/*****************************************************************************\
*       This is the header for the Hufman.c                                   *
\*****************************************************************************/

#define USECOMM      // yes, we need the COMM API

//#undef NO_STRICT    // be bold!

//#define HINSTANCE HANDLE

#include <windows.h>
#include <commdlg.h>
#include <string.h>
#include <io.h>
#include <memory.h>

#include "version.h"
#include "resource.h"

// constant definitions

#define GWL_NPTTYINFO        0
#define TTYEXTRABYTES        sizeof( LONG )

#define ABOUTDLG_USEBITMAP  1

#define ATOM_TTYINFO       0x100

// hard coded maximum number of ports for device under Win32

#define MAXPORTS        4

// terminal size

#define MAXROWS         25
#define MAXCOLS         80

#define MAXBLOCK        1000

#define MAXLEN_TEMPSTR  81

#define RXQUEUE         4096
#define TXQUEUE         4096

// cursor states

#define CS_HIDE         0x00
#define CS_SHOW         0x01

// Flow control flags

#define FC_DTRDSR       0x01
#define FC_RTSCTS       0x02
#define FC_XONXOFF      0x04

// ascii definitions

#define ASCII_BEL       0x07
#define ASCII_BS        0x08
```

```
#define ASCII_LF        0x0A
#define ASCII_CR        0x0D
#define ASCII_XON       0x11
#define ASCII_XOFF      0x13

#define BUFSIZE     80

// data structures

typedef struct tagTTYINFO
{
   HANDLE  idComDev ;
   BYTE    bPort, abScreen[ MAXROWS * MAXCOLS ] ;
   BOOL    fConnected, fXonXoff, fLocalEcho, fNewLine, fAutoWrap,
           fUseCNReceive, fDisplayErrors;
   BYTE    bByteSize, bFlowCtrl, bParity, bStopBits ;
   DWORD   dwBaudRate ;
   WORD    wCursorState ;
   HFONT   hTTYFont ;
   LOGFONT lfTTYFont ;
   DWORD   rgbFGColor ;
   int     xSize, ySize, xScroll, yScroll, xOffset, yOffset,
           nColumn, nRow, xChar, yChar ;
   HANDLE      hPostEvent, hWatchThread, hWatchEvent ;
   HWND        hTermWnd ;
   DWORD       dwThreadID ;
   OVERLAPPED  osWrite, osRead ;

} TTYINFO, NEAR *NPTTYINFO ;

// macros ( for easier readability )

#define GETHINST( x )  ((HINSTANCE) GetWindowLong( x, GWL_HINSTANCE ))
#define GETNPTTYINFO( x ) ((NPTTYINFO) GetWindowLong( x, GWL_NPTTYINFO ))
#define SETNPTTYINFO( x, y ) SetWindowLong( x, GWL_NPTTYINFO, (LONG) y )

#define COMDEV( x ) (x -> idComDev)
#define PORT( x )   (x -> bPort)
#define SCREEN( x ) (x -> abScreen)
#define CONNECTED( x ) (x -> fConnected)
#define XONXOFF( x ) (x -> fXonXoff)
#define LOCALECHO( x ) (x -> fLocalEcho)
#define NEWLINE( x ) (x -> fNewLine)
#define AUTOWRAP( x ) (x -> fAutoWrap)
#define BYTESIZE( x ) (x -> bByteSize)
#define FLOWCTRL( x ) (x -> bFlowCtrl)
#define PARITY( x ) (x -> bParity)
#define STOPBITS( x ) (x -> bStopBits)
#define BAUDRATE( x ) (x -> dwBaudRate)
#define CURSORSTATE( x ) (x -> wCursorState)
#define HTTYFONT( x ) (x -> hTTYFont)
#define LFTTYFONT( x ) (x -> lfTTYFont)
#define FGCOLOR( x ) (x -> rgbFGColor)
#define XSIZE( x ) (x -> xSize)
#define YSIZE( x ) (x -> ySize)
#define XSCROLL( x ) (x -> xScroll)
#define YSCROLL( x ) (x -> yScroll)
#define XOFFSET( x ) (x -> xOffset)
#define YOFFSET( x ) (x -> yOffset)
#define COLUMN( x ) (x -> nColumn)
#define ROW( x ) (x -> nRow)
```

```
#define XCHAR( x ) (x -> xChar)
#define YCHAR( x ) (x -> yChar )
#define USECNRECEIVE( x ) (x -> fUseCNReceive)
#define DISPLAYERRORS( x ) (x -> fDisplayErrors)

#define POSTEVENT( x ) (x -> hPostEvent)
#define TERMWND( x ) (x -> hTermWnd)
#define HTHREAD( x ) (x -> hWatchThread)
#define THREADID( x ) (x -> dwThreadID)
#define WRITE_OS( x ) (x -> osWrite)
#define READ_OS( x ) (x -> osRead)

#define SET_PROP( x, y, z )  SetProp( x, MAKEINTATOM( y ), z )
#define GET_PROP( x, y )     GetProp( x, MAKEINTATOM( y ) )
#define REMOVE_PROP( x, y )  RemoveProp( x, MAKEINTATOM( y ) )

// global stuff

HWND     hTTYWnd ;
char     gszTTYClass[] = "TTYWndClass" ;
char     gszAppName[] = "Real Time Text Transmission with Huffman Encoding" ;
HANDLE   ghAccel ;

DWORD    BaudTable[] =
         {
            CBR_110, CBR_300, CBR_600, CBR_1200, CBR_2400,
            CBR_4800, CBR_9600, CBR_14400, CBR_19200, CBR_38400,
            CBR_56000, CBR_128000, CBR_256000
         } ;

DWORD    ParityTable[] =
         {
            NOPARITY, EVENPARITY, ODDPARITY, MARKPARITY, SPACEPARITY
         } ;

DWORD    StopBitsTable[] =
         {
            ONESTOPBIT, ONE5STOPBITS, TWOSTOPBITS
         } ;

// CRT mappings to NT API

#define _fmemset   memset
#define _fmemmove  memmove

// function prototypes (private)

BOOL NEAR InitApplication( HANDLE ) ;
HWND NEAR InitInstance( HANDLE, int ) ;
LRESULT NEAR CreateTTYInfo( HWND ) ;
BOOL NEAR DestroyTTYInfo( HWND ) ;
BOOL NEAR ResetTTYScreen( HWND, NPTTYINFO ) ;
BOOL NEAR KillTTYFocus( HWND ) ;
BOOL NEAR PaintTTY( HWND ) ;
BOOL NEAR SetTTYFocus( HWND ) ;
BOOL NEAR ScrollTTYHorz( HWND, WORD, WORD ) ;
BOOL NEAR ScrollTTYVert( HWND, WORD, WORD ) ;
BOOL NEAR SizeTTY( HWND, WORD, WORD ) ;
BOOL NEAR ProcessTTYCharacter( HWND, BYTE ) ;
BOOL NEAR WriteTTYBlock( HWND, LPSTR, int ) ;
int NEAR ReadCommBlock( HWND, LPSTR, int ) ;
```

```
BOOL NEAR WriteCommBlock( HWND , LPSTR, DWORD);
BOOL NEAR MoveTTYCursor( HWND ) ;
BOOL NEAR OpenConnection( HWND ) ;
BOOL NEAR SetupConnection( HWND ) ;
BOOL NEAR CloseConnection( HWND ) ;
VOID NEAR GoModalDialogBoxParam( HINSTANCE, LPCSTR, HWND, DLGPROC, LPARAM ) ;
VOID NEAR FillComboBox( HINSTANCE, HWND, int, DWORD NEAR *, WORD, DWORD ) ;
BOOL NEAR SelectTTYFont( HWND ) ;
BOOL NEAR SettingsDlgInit( HWND ) ;
BOOL NEAR SettingsDlgTerm( HWND ) ;

// function prototypes (public)

void  build_decomp_tree ();
void  decompress_image (HWND, LPBYTE, int);
void compress_image (HWND, const char*);
void compress_char (HWND, BYTE);
LRESULT FAR PASCAL TTYWndProc( HWND, UINT, WPARAM, LPARAM ) ;
BOOL FAR PASCAL AboutDlgProc( HWND, UINT, WPARAM, LPARAM ) ;
BOOL FAR PASCAL SettingsDlgProc( HWND, UINT, WPARAM, LPARAM ) ;
BOOL CALLBACK DialingDlgProc(HWND, UINT,WPARAM, LPARAM) ;
BOOL CALLBACK AttensionDlgProc(HWND, UINT,WPARAM, LPARAM) ;
BOOL CALLBACK InformationDlgProc( HWND, UINT, WPARAM, LPARAM);
DWORD FAR PASCAL CommWatchProc( LPSTR ) ;


//---------------------------------------------------------------------------
//  End of File: Huffman.h
//---------------------------------------------------------------------------
```

2. Main Program

```
/*****************************************************************************\
*       This program is  to demonstrate real time text transmission with and   *
*        without compression over a telephone line.The compression algorithm    *
*        used is Huffman Encoding Scheme. The codes are first generated and     *
*        and stored in a file called "Code". These codes are tested to be       *
*        optimal for text compression.                                           *
*                                                                                *
*                            By:       Yohannes Kassahun                        *
*                            Adivisor:  Dr.-Ing Hailu Ayele                     *
*                                                                                *
*****************************************************************************/

//---------------------------------------------------------------------------
//
//  Module: Huffman.c
//
//  Purpose:
//     The program application demonstrates Huffman Encoding
//      as applied to text transmissin over telephone line. It also
//      uses the communicatin functions, the COMM API. It implements
//     the new COMM API of Windows 3.1.
//
//     NOTE:  no escape sequences are translated, only
//            the necessary control codes (LF, CR, BS, etc.)
//
//  Description of functions:
//     Descriptions are contained in the function headers.
//
//---------------------------------------------------------------------------
//
//  Author : Yohannes Kassahun
//
//---------------------------------------------------------------------------

#include "tty.h"
#include "stdio.h"


HWND hwnd;
BOOLEAN Huffman_Coding = FALSE;

short                           father[512];
unsigned short                  code[256], heap_length;
unsigned long                   heap[257];
unsigned char                   code_length[256];
short                           decomp_tree[512];

unsigned short                  cindex;

char                            curchar;
short                           bitshift;
BYTE                            bout;
unsigned short                  charcount = 0L;

unsigned long                   compressed_count;
unsigned long                   file_size;
DWORD                           oldtime,newtime;
BYTE                            newchar = 32,counter;
```

```
/*************************************************************************

 BUILD_DECOMP_TREE ()

 This function builds the decompression tree needed for decompression.
 *************************************************************************/

void  build_decomp_tree ()
{
   register unsigned short  loop1;
   register unsigned short  current_index;

   unsigned short  loop;
   unsigned short  current_node = 1;


   decomp_tree[1] = 1;

   for (loop = 0; loop < 256; loop++)
   {
      if (code_length[loop])
      {
               current_index = 1;
               for (loop1 = code_length[loop] - 1; loop1 > 0; loop1--)
               {
                       current_index = (decomp_tree[current_index] << 1) +
                           ((code[loop] >> loop1) & 1);
           if (!(decomp_tree[current_index]))
              decomp_tree[current_index] = ++current_node;
               }
               decomp_tree[(decomp_tree[current_index] << 1) +
                       (code[loop] & 1)] = -loop;
         }
   }
}

/*************************************************************************

 DECOMPRESS_IMAGE (HWND hwnd, LPBYTE lpBlock, int nLength)

 This function decompresses the compressed image.
 *************************************************************************/


void  decompress_image (HWND hWnd, LPBYTE lpBlock, int nLength )
{
   int i;

   charcount = 0;

   newtime = GetTickCount();

   if ((newtime-oldtime) > 30)
   {
        cindex =1;
        bout = 8;
       if (counter > 1){
                 for (i=0; i < counter-1; i++)
                             WriteTTYBlock(hWnd,&bout,1);
         }
   }

   counter = 0;

   while (charcount < nLength)
```

```
    {
       curchar = (char)lpBlock[charcount];

       for (bitshift = 7; bitshift >= 0; --bitshift)
       {
                cindex = (cindex << 1) + ((curchar >> bitshift) & 1);

                if (decomp_tree[cindex] <= 0)
                {
                        bout =(BYTE)(int) (-decomp_tree[cindex]);
                        counter++;
                        newchar = bout;
                        if ((bout > 126) || ((bout < 32) && ((bout != 10)&&(bout != 13))))
                        {
                                cindex = 1;
                                return;
                        }
                        else
                        {
                                WriteTTYBlock(hWnd,&bout,1);
                                cindex = 1;
                        }
                }
                else{
                        cindex = decomp_tree[cindex];
                }

       }
           charcount = charcount + 1;
    }

  oldtime = GetTickCount();
}


/*************************************************************************

 COMPRESS_IMAGE (HWND hwnd, const char* FileName)

 This function performs the actual data compression.
 *************************************************************************/

void compress_image (HWND hWnd, const char* FileName)
{
   BYTE                 thebyte = 0;
   register short           loop1;
   register unsigned short  current_code;
   register unsigned long   loop;

   unsigned short             current_length, dvalue;
   unsigned long              curbyte = 0;
   short                      curbit = 7;


   FILE * ifile;

   compressed_count = 0;

   if ((ifile = fopen (FileName, "rb")) != NULL){
          fseek (ifile, 0L, 2);
       file_size = (unsigned long) ftell (ifile);

          fseek (ifile, 0L, 0);

          for (loop = 0L; loop < file_size; loop++)
          {
               dvalue         = (unsigned short) getc (ifile);
```

```
                current_code   = code[dvalue];
                current_length = (unsigned short) code_length[dvalue];
               for (loop1 = current_length-1; loop1 >= 0; --loop1)
               {
                      if ((current_code >> loop1) & 1)
               thebyte |= (char) (1 << curbit);

                      if (--curbit < 0)
                      {
                             WriteCommBlock(hWnd,&thebyte,1);
                             compressed_count++;/* Count compressed characters */
                             thebyte = 0;
                             curbyte++;
                             curbit = 7;
                      }
                }
          }
   }

   WriteCommBlock(hWnd,&thebyte,1);
   compressed_count ++;
   fclose (ifile);

   // call Information dialog box to display information about the
   // attained compression.
   hwnd = hWnd;
   DialogBox( GETHINST( hWnd ),
      MAKEINTRESOURCE(INFORMATION_RATE),
      hWnd,InformationDlgProc);
}

/*************************************************************************

 COMPRESS_CHAR (HWND hwnd, const char* FileName)

 This function is used to compress a single character during typing.
 *************************************************************************/

void compress_char (HWND hWnd, BYTE character)
{
   BYTE                thebyte = 0;
   register short          loop1;
   register unsigned short  current_code;

   unsigned short             current_length, dvalue;
   unsigned long              curbyte = 0;
   short                      curbit = 7;

   dvalue         = (unsigned short) character;
   current_code   = code[dvalue];
   current_length = (unsigned short) code_length[dvalue];
   for (loop1 = current_length-1; loop1 >= 0; --loop1)
   {
          if ((current_code >> loop1) & 1)
               thebyte |= (char) (1 << curbit);

       if (--curbit < 0)
          {
                 WriteCommBlock(hWnd,&thebyte,1);
                compressed_count++;/* Count compressed characters */
                 thebyte = 0;
               curbyte++;
               curbit = 7;
          }
   }

   WriteCommBlock(hWnd,&thebyte,1);
```

```
}

//---------------------------------------------------------------------------
//  int PASCAL WinMain( HANDLE hInstance, HANDLE hPrevInstance,
//                      LPSTR lpszCmdLine, int nCmdShow )
//
//  Description:
//     This is the main window loop!
//
//  Parameters:
//     As documented for all WinMain() functions.
//
//---------------------------------------------------------------------------

int PASCAL WinMain( HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpszCmdLine, int nCmdShow )
{
   MSG   msg ;

   if (!hPrevInstance)
      if (!InitApplication( hInstance ))
         return ( FALSE ) ;

   if (NULL == (hTTYWnd = InitInstance( hInstance, nCmdShow )))
      return ( FALSE ) ;

   while (GetMessage( &msg, NULL, 0, 0 ))
   {
      if (!TranslateAccelerator( hTTYWnd, ghAccel, &msg ))
      {
         TranslateMessage( &msg ) ;
         DispatchMessage( &msg ) ;
      }
   }
   return ( (int) msg.wParam ) ;

} // end of WinMain()

//---------------------------------------------------------------------------
//  BOOL NEAR InitApplication( HANDLE hInstance )
//
//  Description:
//     First time initialization stuff.  This registers information
//     such as window classes.
//
//  Parameters:
//     HANDLE hInstance
//        Handle to this instance of the application.
//
//---------------------------------------------------------------------------

BOOL NEAR InitApplication( HANDLE hInstance )
{
   WNDCLASS  wndclass ;

   FILE * ifile;

   // register tty window class

   wndclass.style =         0 ;
   wndclass.lpfnWndProc =   TTYWndProc ;
   wndclass.cbClsExtra =    0 ;
   wndclass.cbWndExtra =    TTYEXTRABYTES ;
   wndclass.hInstance =     hInstance ;
   wndclass.hIcon =         LoadIcon( hInstance, MAKEINTRESOURCE( TTYICON ) );
   wndclass.hCursor =       LoadCursor( NULL, IDC_ARROW ) ;
   wndclass.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1) ;
   wndclass.lpszMenuName =  MAKEINTRESOURCE( TTYMENU ) ;
```

```
    wndclass.lpszClassName = gszTTYClass ;


    //Load the code and code length used for transimmission
    //This code is developed to be optimal for Text transmission

    if ((ifile = fopen ("Code", "rb")) != NULL)
    {
          fseek (ifile, 0L, 0);
             fread (code, 2, 256, ifile);
          fread (code_length, 1, 256, ifile);
          fclose (ifile);
                //bulid the decompression tree here before decompressing
          build_decomp_tree ();
    }
    else
    {
           MessageBox(0,"Not able to load the code","Error!" ,MB_OK |MB_ICONSTOP);
    }

    cindex = 1;
    oldtime = GetTickCount();

    return( RegisterClass( &wndclass ) ) ;

} // end of InitApplication()

//---------------------------------------------------------------------------
//  HWND NEAR InitInstance( HANDLE hInstance, int nCmdShow )
//
//  Description:
//     Initializes instance specific information.
//
//  Parameters:
//     HANDLE hInstance
//        Handle to instance
//
//     int nCmdShow
//        How do we show the window?
//
//---------------------------------------------------------------------------

HWND NEAR InitInstance( HANDLE hInstance, int nCmdShow )
{
   HWND  hTTYWnd ;

   // load accelerators
   ghAccel = LoadAccelerators( hInstance, MAKEINTRESOURCE( TTYACCEL ) ) ;

   // create the TTY window
   hTTYWnd = CreateWindow( gszTTYClass, gszAppName,
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           CW_USEDEFAULT, CW_USEDEFAULT,
                           NULL, NULL, hInstance, NULL ) ;

   if (NULL == hTTYWnd)
      return ( NULL ) ;

   ShowWindow( hTTYWnd, nCmdShow ) ;
   UpdateWindow( hTTYWnd ) ;

   return ( hTTYWnd ) ;

} // end of InitInstance()
```

```
//---------------------------------------------------------------------------
//  LRESULT FAR PASCAL TTYWndProc( HWND hWnd, UINT uMsg,
//                                 WPARAM wParam, LPARAM lParam )
//
//  Description:
//     This is the TTY Window Proc.  This handles ALL messages
//     to the tty window.
//
//  Parameters:
//     As documented for Window procedures.
//
//  Win-32 Porting Issues:
//     - WM_HSCROLL and WM_VSCROLL packing is different under Win-32.
//     - Needed LOWORD() of wParam for WM_CHAR messages.
//
//---------------------------------------------------------------------------

LRESULT FAR PASCAL TTYWndProc( HWND hWnd, UINT uMsg,
                               WPARAM wParam, LPARAM lParam )
{

   DWORD dwBlockSize=512;
   char * szBuffer;

   switch (uMsg)
   {

      case WM_CREATE:
         return ( CreateTTYInfo( hWnd ) ) ;

      case WM_COMMAND:
      {
         switch ( LOWORD( wParam ) )
         {
                        case IDM_DIAL:
                              hwnd = hWnd;
                              DialogBox( GETHINST( hWnd ),
                           MAKEINTRESOURCE(DILAING_DIALOG),
                           hWnd,DialingDlgProc);
                              break;
                        case IDM_ATTENSION:
                              hwnd = hWnd;
                              DialogBox( GETHINST( hWnd ),
                           MAKEINTRESOURCE(ATTENSION_DIALOG),
                           hWnd,AttensionDlgProc);
                              break;

                        case IDM_HUFFMAN_CODING:
                              hwnd = hWnd;
                              if (Huffman_Coding){
                                    CheckMenuItem(GetMenu(hWnd),IDM_HUFFMAN_CODING
                                                            ,MF_UNCHECKED);
                                    Huffman_Coding = FALSE;
                              }
                              else
                              {
                                    CheckMenuItem(GetMenu(hWnd),IDM_HUFFMAN_CODING
                                                      ,MF_CHECKED);
                                    Huffman_Coding = TRUE;
                              }
                              break;

                  case IDM_OPEN:
                              {
                              OPENFILENAME ofn;
                              char szFileName[BUFSIZE];
                              /* prompt user for file to open */
```

```
                    ofn.lStructSize = sizeof(OPENFILENAME);
                    ofn.hwndOwner = hWnd;
                    ofn.hInstance = NULL;
                    ofn.lpstrFilter = NULL;
                    ofn.lpstrCustomFilter = NULL;
                    ofn.nMaxCustFilter = 0;
                    ofn.nFilterIndex = 0;
                    *szFileName = '\0';
                    ofn.lpstrFile = szFileName;
                    ofn.nMaxFile = BUFSIZE;
                    ofn.lpstrFileTitle = NULL;
                    ofn.nMaxFileTitle = 0;
                    ofn.lpstrInitialDir = NULL;
                    ofn.lpstrTitle = "Open";
                    ofn.Flags = OFN_FILEMUSTEXIST | OFN_PATHMUSTEXIST
                                        | OFN_HIDEREADONLY;
                    ofn.nFileOffset = 0;
                    ofn.nFileExtension = 0;
                    ofn.lpstrDefExt = NULL;
                    ofn.lCustData = 0;
                    ofn.lpfnHook = NULL;
                    ofn.lpTemplateName = NULL;
                    if(GetOpenFileName(&ofn)){
                            if (Huffman_Coding){
                                    compress_image(hWnd,ofn.lpstrFile);
                            }
                            else
                            {
                                    FILE * ifile;
                                    BYTE b;
                                    if ((ifile =
                                            fopen (ofn.lpstrFile, "rb")) != NULL){
                                            register unsigned long loop,file_size;
                                            fseek (ifile, 0L, 2);
                                            file_size =
                                                    (unsigned long) ftell (ifile);
                                            fseek (ifile, 0L, 0);
                                            for (loop = 0; loop < file_size;
                                                 loop++){
                                                    b = getc (ifile);
                                                    WriteCommBlock(hWnd,&b,1);
                                            }
                                            fclose (ifile);
                                    }
                            }
                    }
                    }
                     break;
case IDM_CONNECT:
   if (!OpenConnection( hWnd ))
      MessageBox( hWnd, "Connection failed.", gszAppName,
                  MB_ICONEXCLAMATION ) ;
   break ;

case IDM_DISCONNECT:
   KillTTYFocus( hWnd ) ;
   CloseConnection( hWnd ) ;
   break ;

case IDM_SETTINGS:
{
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;
   GoModalDialogBoxParam( GETHINST( hWnd ),
                          MAKEINTRESOURCE( SETTINGSDLGBOX )
                            , hWnd,(DLGPROC) SettingsDlgProc,
```

```
                                (LPARAM) (LPSTR) npTTYInfo ) ;

            // if fConnected, set new COM parameters

            if (CONNECTED( npTTYInfo ))
            {
               if (!SetupConnection( hWnd ))
                  MessageBox( hWnd, "Settings failed!", gszAppName,
                              MB_ICONEXCLAMATION ) ;
            }
         }
         break ;

         case IDM_SENDBLOCK:
            // This function is very useful to test how
            // your comm application handles large blocks
            // of data

                     szBuffer = malloc(dwBlockSize);
                     memset(szBuffer, 'X', dwBlockSize);
                     WriteCommBlock( hWnd, szBuffer,dwBlockSize);
                     free(szBuffer);
                  break;

         case IDM_ABOUT:
            GoModalDialogBoxParam ( GETHINST( hWnd ),
                                    MAKEINTRESOURCE( ABOUTDLGBOX ),
                                    hWnd,
                                    (DLGPROC) AboutDlgProc, 0L ) ;
            break;

         case IDM_EXIT:
            PostMessage( hWnd, WM_CLOSE, 0, 0L ) ;
            break ;
      }
   }
   break ;

   case WM_PAINT:
      PaintTTY( hWnd ) ;
      break ;

   case WM_SIZE:
      SizeTTY( hWnd, HIWORD( lParam ), LOWORD( lParam ) ) ;
      break ;

   case WM_HSCROLL:
      ScrollTTYHorz( hWnd, LOWORD( wParam ), HIWORD( wParam ) ) ;
      break ;

   case WM_VSCROLL:
      ScrollTTYVert( hWnd, LOWORD( wParam ), HIWORD( wParam ) ) ;
      break ;

   case WM_CHAR:
      ProcessTTYCharacter( hWnd, LOBYTE( LOWORD( wParam ) ) ) ;
      break ;

   case WM_SETFOCUS:
      SetTTYFocus( hWnd ) ;
      break ;

   case WM_KILLFOCUS:
      KillTTYFocus( hWnd ) ;
      break ;

   case WM_DESTROY:
      DestroyTTYInfo( hWnd ) ;
```

```
            PostQuitMessage( 0 ) ;
            break ;

         case WM_CLOSE:
            if (IDOK != MessageBox( hWnd, "OK to close window?",
                                    "Huffman Coding",
                                      MB_ICONQUESTION | MB_OKCANCEL ))
            break ;

            // fall through

         default:
            return( DefWindowProc( hWnd, uMsg, wParam, lParam ) ) ;
      }
      return 0L ;

} // end of TTYWndProc()

//---------------------------------------------------------------------------
//  LRESULT NEAR CreateTTYInfo( HWND hWnd )
//
//  Description:
//     Creates the tty information structure and sets
//     menu option availability.  Returns -1 if unsuccessful.
//
//  Parameters:
//     HWND  hWnd
//        Handle to main window.
//
//  Win-32 Porting Issues:
//     - Needed to initialize TERMWND( npTTYInfo ) for secondary thread.
//     - Needed to create/initialize overlapped structures used in reads &
//       writes to COMM device.
//
//---------------------------------------------------------------------------

LRESULT NEAR CreateTTYInfo( HWND hWnd )
{
   HMENU       hMenu ;
   NPTTYINFO   npTTYInfo ;

   if (NULL == (npTTYInfo =
                  (NPTTYINFO) LocalAlloc( LPTR, sizeof( TTYINFO ) )))
      return ( (LRESULT) -1 ) ;

   // initialize TTY info structure

   COMDEV( npTTYInfo )        = 0 ;
   CONNECTED( npTTYInfo )     = FALSE ;
   CURSORSTATE( npTTYInfo )   = CS_HIDE ;
   LOCALECHO( npTTYInfo )     = FALSE ;
   AUTOWRAP( npTTYInfo )      = TRUE ;
   PORT( npTTYInfo )          = 1 ;
   BAUDRATE( npTTYInfo )      = CBR_9600 ;
   BYTESIZE( npTTYInfo )      = 8 ;
   //FLOWCTRL( npTTYInfo )      = FC_RTSCTS ;
   FLOWCTRL( npTTYInfo )      = FC_XONXOFF ;
   PARITY( npTTYInfo )        = NOPARITY ;
   STOPBITS( npTTYInfo )      = ONESTOPBIT ;
   XONXOFF( npTTYInfo )       = FALSE ;
   XSIZE( npTTYInfo )         = 0 ;
   YSIZE( npTTYInfo )         = 0 ;
   XSCROLL( npTTYInfo )       = 0 ;
   YSCROLL( npTTYInfo )       = 0 ;
   XOFFSET( npTTYInfo )       = 0 ;
   YOFFSET( npTTYInfo )       = 0 ;
   COLUMN( npTTYInfo )        = 0 ;
   ROW( npTTYInfo )           = 0 ;
```

```
HTTYFONT( npTTYInfo )       = NULL ;
FGCOLOR( npTTYInfo )        = RGB( 0, 0, 0 ) ;
USECNRECEIVE( npTTYInfo )   = TRUE ;
DISPLAYERRORS( npTTYInfo )  = TRUE ;
WRITE_OS( npTTYInfo ).Offset = 0 ;
WRITE_OS( npTTYInfo ).OffsetHigh = 0 ;
READ_OS( npTTYInfo ).Offset = 0 ;
READ_OS( npTTYInfo ).OffsetHigh = 0 ;
TERMWND( npTTYInfo ) =        hWnd ;

// create I/O event used for overlapped reads / writes

READ_OS( npTTYInfo ).hEvent = CreateEvent( NULL,     // no security
                                           TRUE,     // explicit reset req
                                           FALSE,    // initial event reset
                                           NULL ) ; // no name

if (READ_OS( npTTYInfo ).hEvent == NULL)
{
   LocalFree( npTTYInfo ) ;
   return ( -1 ) ;
}
WRITE_OS( npTTYInfo ).hEvent = CreateEvent( NULL,     // no security
                                            TRUE,     // explicit reset req
                                            FALSE,    // initial event reset
                                            NULL ) ; // no name

if (NULL == WRITE_OS( npTTYInfo ).hEvent)
{
   CloseHandle( READ_OS( npTTYInfo ).hEvent ) ;
   LocalFree( npTTYInfo ) ;
   return ( -1 ) ;
}

// clear screen space

_fmemset( SCREEN( npTTYInfo ), ' ', MAXROWS * MAXCOLS ) ;

// setup default font information

LFTTYFONT( npTTYInfo ).lfHeight =         9 ;
LFTTYFONT( npTTYInfo ).lfWidth =          0 ;
LFTTYFONT( npTTYInfo ).lfEscapement =     0 ;
LFTTYFONT( npTTYInfo ).lfOrientation =    0 ;
LFTTYFONT( npTTYInfo ).lfWeight =         0 ;
LFTTYFONT( npTTYInfo ).lfItalic =         0 ;
LFTTYFONT( npTTYInfo ).lfUnderline =      0 ;
LFTTYFONT( npTTYInfo ).lfStrikeOut =      0 ;
LFTTYFONT( npTTYInfo ).lfCharSet =        OEM_CHARSET ;
LFTTYFONT( npTTYInfo ).lfOutPrecision =   OUT_DEFAULT_PRECIS ;
LFTTYFONT( npTTYInfo ).lfClipPrecision =  CLIP_DEFAULT_PRECIS ;
LFTTYFONT( npTTYInfo ).lfQuality =        DEFAULT_QUALITY ;
LFTTYFONT( npTTYInfo ).lfPitchAndFamily = FIXED_PITCH | FF_MODERN ;
lstrcpy( LFTTYFONT( npTTYInfo ).lfFaceName, "FixedSys" ) ;

// set TTYInfo handle before any further message processing.

SETNPTTYINFO( hWnd, npTTYInfo ) ;

// reset the character information, etc.

ResetTTYScreen( hWnd, npTTYInfo ) ;

hMenu = GetMenu( hWnd ) ;
EnableMenuItem( hMenu, IDM_DISCONNECT,
                MF_GRAYED | MF_DISABLED | MF_BYCOMMAND ) ;
EnableMenuItem( hMenu, IDM_CONNECT, MF_ENABLED | MF_BYCOMMAND ) ;

return ( (LRESULT) TRUE ) ;
```

```
} // end of CreateTTYInfo()

//---------------------------------------------------------------------------
//  BOOL NEAR DestroyTTYInfo( HWND hWnd )
//
//  Description:
//     Destroys block associated with TTY window handle.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//  Win-32 Porting Issues:
//     - Needed to clean up event objects created during initialization.
//
//---------------------------------------------------------------------------

BOOL NEAR DestroyTTYInfo( HWND hWnd )
{
   NPTTYINFO npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   // force connection closed (if not already closed)

   if (CONNECTED( npTTYInfo ))
      CloseConnection( hWnd ) ;

   // clean up event objects

   CloseHandle( READ_OS( npTTYInfo ).hEvent ) ;
   CloseHandle( WRITE_OS( npTTYInfo ).hEvent ) ;
   CloseHandle( POSTEVENT( npTTYInfo ) ) ;

   DeleteObject( HTTYFONT( npTTYInfo ) ) ;

   LocalFree( npTTYInfo ) ;
   return ( TRUE ) ;

} // end of DestroyTTYInfo()
```

```
//---------------------------------------------------------------------------
//  BOOL NEAR ResetTTYScreen( HWND hWnd, NPTTYINFO npTTYInfo )
//
//  Description:
//     Resets the TTY character information and causes the
//     screen to resize to update the scroll information.
//
//  Parameters:
//     NPTTYINFO  npTTYInfo
//        pointer to TTY info structure
//
//---------------------------------------------------------------------------

BOOL NEAR ResetTTYScreen( HWND hWnd, NPTTYINFO npTTYInfo )
{
   HDC         hDC ;
   TEXTMETRIC  tm ;
   RECT        rcWindow ;

   if (NULL == npTTYInfo)
      return ( FALSE ) ;

   if (NULL != HTTYFONT( npTTYInfo ))
      DeleteObject( HTTYFONT( npTTYInfo ) ) ;

   HTTYFONT( npTTYInfo ) = CreateFontIndirect( &LFTTYFONT( npTTYInfo ) ) ;

   hDC = GetDC( hWnd ) ;
   SelectObject( hDC, HTTYFONT( npTTYInfo ) ) ;
   GetTextMetrics( hDC, &tm ) ;
   ReleaseDC( hWnd, hDC ) ;

   XCHAR( npTTYInfo ) = tm.tmAveCharWidth  ;
   YCHAR( npTTYInfo ) = tm.tmHeight + tm.tmExternalLeading ;

   // a slimy hack to force the scroll position, region to
   // be recalculated based on the new character sizes

   GetWindowRect( hWnd, &rcWindow ) ;
   SendMessage( hWnd, WM_SIZE, SIZENORMAL,
               (LPARAM) MAKELONG( rcWindow.right - rcWindow.left,
                                  rcWindow.bottom - rcWindow.top ) ) ;

   return ( TRUE ) ;

} // end of ResetTTYScreen()

//---------------------------------------------------------------------------
//  BOOL NEAR PaintTTY( HWND hWnd )
//
//  Description:
//     Paints the rectangle determined by the paint struct of
//     the DC.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window (as always)
//
//---------------------------------------------------------------------------

BOOL NEAR PaintTTY( HWND hWnd )
{
   int          nRow, nCol, nEndRow, nEndCol, nCount, nHorzPos, nVertPos ;
```

```
    HDC          hDC ;
    HFONT        hOldFont ;
    NPTTYINFO    npTTYInfo ;
    PAINTSTRUCT  ps ;
    RECT         rect ;

    if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
       return ( FALSE ) ;

    hDC = BeginPaint( hWnd, &ps ) ;
    hOldFont = SelectObject( hDC, HTTYFONT( npTTYInfo ) ) ;
    SetTextColor( hDC, FGCOLOR( npTTYInfo ) ) ;
    SetBkColor( hDC, GetSysColor( COLOR_WINDOW ) ) ;
    rect = ps.rcPaint ;
    nRow =
       min( MAXROWS - 1,
            max( 0, (rect.top + YOFFSET( npTTYInfo )) / YCHAR( npTTYInfo ) ) ) ;
    nEndRow =
       min( MAXROWS - 1,
            ((rect.bottom + YOFFSET( npTTYInfo ) - 1) / YCHAR( npTTYInfo ) ) ) ;
    nCol =
       min( MAXCOLS - 1,
            max( 0, (rect.left + XOFFSET( npTTYInfo )) / XCHAR( npTTYInfo ) ) ) ;
    nEndCol =
       min( MAXCOLS - 1,
            ((rect.right + XOFFSET( npTTYInfo ) - 1) / XCHAR( npTTYInfo ) ) ) ;
    nCount = nEndCol - nCol + 1 ;
    for (; nRow <= nEndRow; nRow++)
    {
       nVertPos = (nRow * YCHAR( npTTYInfo )) - YOFFSET( npTTYInfo ) ;
       nHorzPos = (nCol * XCHAR( npTTYInfo )) - XOFFSET( npTTYInfo ) ;
       rect.top = nVertPos ;
       rect.bottom = nVertPos + YCHAR( npTTYInfo ) ;
       rect.left = nHorzPos ;
       rect.right = nHorzPos + XCHAR( npTTYInfo ) * nCount ;
       SetBkMode( hDC, OPAQUE ) ;
       ExtTextOut( hDC, nHorzPos, nVertPos, ETO_OPAQUE | ETO_CLIPPED, &rect,
                   (LPSTR)( SCREEN( npTTYInfo ) + nRow * MAXCOLS + nCol ),
                   nCount, NULL ) ;
    }
    SelectObject( hDC, hOldFont ) ;
    EndPaint( hWnd, &ps ) ;
    MoveTTYCursor( hWnd ) ;
    return ( TRUE ) ;

} // end of PaintTTY()

//---------------------------------------------------------------------------
//  BOOL NEAR SizeTTY( HWND hWnd, WORD wVertSize, WORD wHorzSize )
//
//  Description:
//     Sizes TTY and sets up scrolling regions.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     WORD wVertSize
//        new vertical size
//
//     WORD wHorzSize
//        new horizontal size
//
//---------------------------------------------------------------------------

BOOL NEAR SizeTTY( HWND hWnd, WORD wVertSize, WORD wHorzSize )
{
    int        nScrollAmt ;
```

```
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   YSIZE( npTTYInfo ) = (int) wVertSize ;
   YSCROLL( npTTYInfo ) = max( 0, (MAXROWS * YCHAR( npTTYInfo )) -
                               YSIZE( npTTYInfo ) ) ;
   nScrollAmt = min( YSCROLL( npTTYInfo ), YOFFSET( npTTYInfo ) ) -
                YOFFSET( npTTYInfo ) ;
   ScrollWindow( hWnd, 0, -nScrollAmt, NULL, NULL ) ;

   YOFFSET( npTTYInfo ) = YOFFSET( npTTYInfo ) + nScrollAmt ;
   SetScrollPos( hWnd, SB_VERT, YOFFSET( npTTYInfo ), FALSE ) ;
   SetScrollRange( hWnd, SB_VERT, 0, YSCROLL( npTTYInfo ), TRUE ) ;

   XSIZE( npTTYInfo ) = (int) wHorzSize ;
   XSCROLL( npTTYInfo ) = max( 0, (MAXCOLS * XCHAR( npTTYInfo )) -
                               XSIZE( npTTYInfo ) ) ;
   nScrollAmt = min( XSCROLL( npTTYInfo ), XOFFSET( npTTYInfo )) -
                XOFFSET( npTTYInfo ) ;
   ScrollWindow( hWnd, 0, -nScrollAmt, NULL, NULL ) ;
   XOFFSET( npTTYInfo ) = XOFFSET( npTTYInfo ) + nScrollAmt ;
   SetScrollPos( hWnd, SB_HORZ, XOFFSET( npTTYInfo ), FALSE ) ;
   SetScrollRange( hWnd, SB_HORZ, 0, XSCROLL( npTTYInfo ), TRUE ) ;

   InvalidateRect( hWnd, NULL, TRUE ) ;

   return ( TRUE ) ;

} // end of SizeTTY()

//---------------------------------------------------------------------------
//  BOOL NEAR ScrollTTYVert( HWND hWnd, WORD wScrollCmd, WORD wScrollPos )
//
//  Description:
//     Scrolls TTY window vertically.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     WORD wScrollCmd
//        type of scrolling we're doing
//
//     WORD wScrollPos
//        scroll position
//
//---------------------------------------------------------------------------

BOOL NEAR ScrollTTYVert( HWND hWnd, WORD wScrollCmd, WORD wScrollPos )
{
   int        nScrollAmt ;
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   switch (wScrollCmd)
   {
      case SB_TOP:
         nScrollAmt = -YOFFSET( npTTYInfo ) ;
         break ;

      case SB_BOTTOM:
         nScrollAmt = YSCROLL( npTTYInfo ) - YOFFSET( npTTYInfo ) ;
         break ;
```

```
      case SB_PAGEUP:
         nScrollAmt = -YSIZE( npTTYInfo ) ;
         break ;

      case SB_PAGEDOWN:
         nScrollAmt = YSIZE( npTTYInfo ) ;
         break ;

      case SB_LINEUP:
         nScrollAmt = -YCHAR( npTTYInfo ) ;
         break ;

      case SB_LINEDOWN:
         nScrollAmt = YCHAR( npTTYInfo ) ;
         break ;

      case SB_THUMBPOSITION:
         nScrollAmt = wScrollPos - YOFFSET( npTTYInfo ) ;
         break ;

      default:
         return ( FALSE ) ;
   }
   if ((YOFFSET( npTTYInfo ) + nScrollAmt) > YSCROLL( npTTYInfo ))
      nScrollAmt = YSCROLL( npTTYInfo ) - YOFFSET( npTTYInfo ) ;
   if ((YOFFSET( npTTYInfo ) + nScrollAmt) < 0)
      nScrollAmt = -YOFFSET( npTTYInfo ) ;
   ScrollWindow( hWnd, 0, -nScrollAmt, NULL, NULL ) ;
   YOFFSET( npTTYInfo ) = YOFFSET( npTTYInfo ) + nScrollAmt ;
   SetScrollPos( hWnd, SB_VERT, YOFFSET( npTTYInfo ), TRUE ) ;

   return ( TRUE ) ;

} // end of ScrollTTYVert()

//---------------------------------------------------------------------------
//  BOOL NEAR ScrollTTYHorz( HWND hWnd, WORD wScrollCmd, WORD wScrollPos )
//
//  Description:
//     Scrolls TTY window horizontally.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     WORD wScrollCmd
//        type of scrolling we're doing
//
//     WORD wScrollPos
//        scroll position
//
//---------------------------------------------------------------------------

BOOL NEAR ScrollTTYHorz( HWND hWnd, WORD wScrollCmd, WORD wScrollPos )
{
   int        nScrollAmt ;
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   switch (wScrollCmd)
   {
      case SB_TOP:
         nScrollAmt = -XOFFSET( npTTYInfo ) ;
         break ;

      case SB_BOTTOM:
```

```
            nScrollAmt = XSCROLL( npTTYInfo ) - XOFFSET( npTTYInfo ) ;
            break ;

         case SB_PAGEUP:
            nScrollAmt = -XSIZE( npTTYInfo ) ;
            break ;

         case SB_PAGEDOWN:
            nScrollAmt = XSIZE( npTTYInfo ) ;
            break ;

         case SB_LINEUP:
            nScrollAmt = -XCHAR( npTTYInfo ) ;
            break ;

         case SB_LINEDOWN:
            nScrollAmt = XCHAR( npTTYInfo ) ;
            break ;

         case SB_THUMBPOSITION:
            nScrollAmt = wScrollPos - XOFFSET( npTTYInfo ) ;
            break ;

         default:
            return ( FALSE ) ;
      }
      if ((XOFFSET( npTTYInfo ) + nScrollAmt) > XSCROLL( npTTYInfo ))
         nScrollAmt = XSCROLL( npTTYInfo ) - XOFFSET( npTTYInfo ) ;
      if ((XOFFSET( npTTYInfo ) + nScrollAmt) < 0)
         nScrollAmt = -XOFFSET( npTTYInfo ) ;
      ScrollWindow( hWnd, -nScrollAmt, 0, NULL, NULL ) ;
      XOFFSET( npTTYInfo ) = XOFFSET( npTTYInfo ) + nScrollAmt ;
      SetScrollPos( hWnd, SB_HORZ, XOFFSET( npTTYInfo ), TRUE ) ;

      return ( TRUE ) ;

} // end of ScrollTTYHorz()

//---------------------------------------------------------------------------
//  BOOL NEAR SetTTYFocus( HWND hWnd )
//
//  Description:
//     Sets the focus to the TTY window also creates caret.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//---------------------------------------------------------------------------

BOOL NEAR SetTTYFocus( HWND hWnd )
{
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   if (CONNECTED( npTTYInfo ) && (CURSORSTATE( npTTYInfo ) != CS_SHOW))
   {
      CreateCaret( hWnd, NULL, XCHAR( npTTYInfo ), YCHAR( npTTYInfo ) ) ;
      ShowCaret( hWnd ) ;
      CURSORSTATE( npTTYInfo ) = CS_SHOW ;
   }
   MoveTTYCursor( hWnd ) ;
   return ( TRUE ) ;

} // end of SetTTYFocus()
```

```
//---------------------------------------------------------------------------
//  BOOL NEAR KillTTYFocus( HWND hWnd )
//
//  Description:
//     Kills TTY focus and destroys the caret.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//---------------------------------------------------------------------------

BOOL NEAR KillTTYFocus( HWND hWnd )
{
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   if (CONNECTED( npTTYInfo ) && (CURSORSTATE( npTTYInfo ) != CS_HIDE))
   {
      HideCaret( hWnd ) ;
      DestroyCaret() ;
      CURSORSTATE( npTTYInfo ) = CS_HIDE ;
   }
   return ( TRUE ) ;

} // end of KillTTYFocus()

//---------------------------------------------------------------------------
//  BOOL NEAR MoveTTYCursor( HWND hWnd )
//
//  Description:
//     Moves caret to current position.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//---------------------------------------------------------------------------

BOOL NEAR MoveTTYCursor( HWND hWnd )
{
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   if (CONNECTED( npTTYInfo ) && (CURSORSTATE( npTTYInfo ) & CS_SHOW))
      SetCaretPos( (COLUMN( npTTYInfo ) * XCHAR( npTTYInfo )) -
                   XOFFSET( npTTYInfo ),
                   (ROW( npTTYInfo ) * YCHAR( npTTYInfo )) -
                   YOFFSET( npTTYInfo ) ) ;

   return ( TRUE ) ;

} // end of MoveTTYCursor()
```

```
//---------------------------------------------------------------------------
//  BOOL NEAR ProcessTTYCharacter( HWND hWnd, BYTE bOut )
//
//  Description:
//     This simply writes a character to the port and echos it
//     to the TTY screen if fLocalEcho is set.  Some minor
//     keyboard mapping could be performed here.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     BYTE bOut
//        byte from keyboard
//
//---------------------------------------------------------------------------

BOOL NEAR ProcessTTYCharacter( HWND hWnd, BYTE bOut )
{
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   if (!CONNECTED( npTTYInfo ))
      return ( FALSE ) ;

   // a robust app would take appropriate steps if WriteCommBlock failed
   if (Huffman_Coding)
         compress_char (hWnd, bOut);
   else
      WriteCommBlock( hWnd, &bOut, 1 ) ;
   if (LOCALECHO( npTTYInfo ))
      WriteTTYBlock( hWnd, &bOut, 1 ) ;

   return ( TRUE ) ;

} // end of ProcessTTYCharacter()

//---------------------------------------------------------------------------
//  BOOL NEAR OpenConnection( HWND hWnd )
//
//  Description:
//     Opens communication port specified in the TTYINFO struct.
//     It also sets the CommState and notifies the window via
//     the fConnected flag in the TTYINFO struct.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//  Win-32 Porting Issues:
//     - OpenComm() is not supported under Win-32.  Use CreateFile()
//       and setup for OVERLAPPED_IO.
//     - Win-32 has specific communication timeout parameters.
//     - Created the secondary thread for event notification.
//
//---------------------------------------------------------------------------

BOOL NEAR OpenConnection( HWND hWnd )
{
   char       szPort[ 15 ], szTemp[ 10 ] ;
   BOOL       fRetVal ;
   HCURSOR    hOldCursor, hWaitCursor ;
   HMENU      hMenu ;
```

```
NPTTYINFO  npTTYInfo ;

HANDLE        hCommWatchThread ;
DWORD         dwThreadID ;
COMMTIMEOUTS  CommTimeOuts ;

if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
   return ( FALSE ) ;

// show the hourglass cursor
hWaitCursor = LoadCursor( NULL, IDC_WAIT ) ;
hOldCursor = SetCursor( hWaitCursor ) ;

// load the COM prefix string and append port number

LoadString( GETHINST( hWnd ), IDS_COMPREFIX, szTemp, sizeof( szTemp ) ) ;
wsprintf( szPort, "%s%d", (LPSTR) szTemp, PORT( npTTYInfo ) ) ;

// open COMM device

if ((COMDEV( npTTYInfo ) =
   CreateFile( szPort, GENERIC_READ | GENERIC_WRITE,
               0,                    // exclusive access
               NULL,                 // no security attrs
               OPEN_EXISTING,
               FILE_ATTRIBUTE_NORMAL |
               FILE_FLAG_OVERLAPPED, // overlapped I/O
               NULL )) == (HANDLE) -1 )
   return ( FALSE ) ;
else
{
   // get any early notifications

   SetCommMask( COMDEV( npTTYInfo ), EV_RXCHAR ) ;

   // setup device buffers

   SetupComm( COMDEV( npTTYInfo ), 4096, 4096 ) ;

   // purge any information in the buffer

   PurgeComm( COMDEV( npTTYInfo ), PURGE_TXABORT | PURGE_RXABORT |
                                   PURGE_TXCLEAR | PURGE_RXCLEAR ) ;

   // set up for overlapped I/O

   CommTimeOuts.ReadIntervalTimeout = 0xFFFFFFFF ;
   CommTimeOuts.ReadTotalTimeoutMultiplier = 0 ;
   CommTimeOuts.ReadTotalTimeoutConstant = 1000 ;
   // CBR_9600 is approximately 1byte/ms. For our purposes, allow
   // double the expected time per character for a fudge factor.
   CommTimeOuts.WriteTotalTimeoutMultiplier =
                                 2*CBR_9600/BAUDRATE( npTTYInfo ) ;
   CommTimeOuts.WriteTotalTimeoutConstant = 0 ;
   SetCommTimeouts( COMDEV( npTTYInfo ), &CommTimeOuts ) ;
}

fRetVal = SetupConnection( hWnd ) ;

if (fRetVal)
{
   CONNECTED( npTTYInfo ) = TRUE ;

   // Create a secondary thread
   // to watch for an event.

   if (NULL == (hCommWatchThread =
                  CreateThread( (LPSECURITY_ATTRIBUTES) NULL,
```

```
                                    0,
                                    (LPTHREAD_START_ROUTINE) CommWatchProc,
                                    (LPVOID) npTTYInfo,
                                    0, &dwThreadID )))
      {
         CONNECTED( npTTYInfo ) = FALSE ;
         CloseHandle( COMDEV( npTTYInfo ) ) ;
         fRetVal = FALSE ;
      }
      else
      {
         THREADID( npTTYInfo ) = dwThreadID ;
         HTHREAD( npTTYInfo ) = hCommWatchThread ;

         // assert DTR

         EscapeCommFunction( COMDEV( npTTYInfo ), SETDTR ) ;

         SetTTYFocus( hWnd ) ;

         hMenu = GetMenu( hWnd ) ;
         EnableMenuItem( hMenu, IDM_DISCONNECT,
                         MF_ENABLED | MF_BYCOMMAND ) ;
         EnableMenuItem( hMenu, IDM_CONNECT,
                         MF_GRAYED | MF_DISABLED | MF_BYCOMMAND ) ;

      }
   }
   else
   {
      CONNECTED( npTTYInfo ) = FALSE ;
      CloseHandle( COMDEV( npTTYInfo ) ) ;
   }

   // restore cursor

   SetCursor( hOldCursor ) ;

   return ( fRetVal ) ;

} // end of OpenConnection()

//---------------------------------------------------------------------------
//  BOOL NEAR SetupConnection( HWND hWnd )
//
//  Description:
//     This routines sets up the DCB based on settings in the
//     TTY info structure and performs a SetCommState().
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//  Win-32 Porting Issues:
//     - Win-32 requires a slightly different processing of the DCB.
//       Changes were made for configuration of the hardware handshaking
//       lines.
//
//---------------------------------------------------------------------------

BOOL NEAR SetupConnection( HWND hWnd )
{
   BOOL       fRetVal ;
   BYTE       bSet ;
   DCB        dcb ;
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
```

```
      return ( FALSE ) ;

   dcb.DCBlength = sizeof( DCB ) ;

   GetCommState( COMDEV( npTTYInfo ), &dcb ) ;

   dcb.BaudRate = BAUDRATE( npTTYInfo ) ;
   dcb.ByteSize = BYTESIZE( npTTYInfo ) ;
   dcb.Parity = PARITY( npTTYInfo ) ;
   dcb.StopBits = STOPBITS( npTTYInfo ) ;

   // setup hardware flow control

   bSet = (BYTE) ((FLOWCTRL( npTTYInfo ) & FC_DTRDSR) != 0) ;
   dcb.fOutxDsrFlow = bSet ;
   if (bSet)
      dcb.fDtrControl = DTR_CONTROL_HANDSHAKE ;
   else
      dcb.fDtrControl = DTR_CONTROL_ENABLE ;

   bSet = (BYTE) ((FLOWCTRL( npTTYInfo ) & FC_RTSCTS) != 0) ;
        dcb.fOutxCtsFlow = bSet ;
   if (bSet)
      dcb.fRtsControl = RTS_CONTROL_HANDSHAKE ;
   else
      dcb.fRtsControl = RTS_CONTROL_ENABLE ;

   // setup software flow control

   bSet = (BYTE) ((FLOWCTRL( npTTYInfo ) & FC_XONXOFF) != 0) ;

   dcb.fInX = dcb.fOutX = bSet ;
   dcb.XonChar = ASCII_XON ;
   dcb.XoffChar = ASCII_XOFF ;
   dcb.XonLim = 100 ;
   dcb.XoffLim = 100 ;

   // other various settings

   dcb.fBinary = TRUE ;
   dcb.fParity = TRUE ;

   fRetVal = SetCommState( COMDEV( npTTYInfo ), &dcb ) ;

   return ( fRetVal ) ;

} // end of SetupConnection()
```

```
//-------------------------------------------------------------------------
//  BOOL NEAR CloseConnection( HWND hWnd )
//
//  Description:
//     Closes the connection to the port.  Resets the connect flag
//     in the TTYINFO struct.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//  Win-32 Porting Issues:
//     - Needed to stop secondary thread.  SetCommMask() will signal the
//       WaitCommEvent() event and the thread will halt when the
//       CONNECTED() flag is clear.
//     - Use new PurgeComm() API to clear communications driver before
//       closing device.
//
//-------------------------------------------------------------------------

BOOL NEAR CloseConnection( HWND hWnd )
{
   HMENU      hMenu ;
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   // set connected flag to FALSE

   CONNECTED( npTTYInfo ) = FALSE ;

   // disable event notification and wait for thread
   // to halt

   SetCommMask( COMDEV( npTTYInfo ), 0 ) ;

   // block until thread has been halted

   while(THREADID(npTTYInfo) != 0);

   // kill the focus

   KillTTYFocus( hWnd ) ;

   // drop DTR

   EscapeCommFunction( COMDEV( npTTYInfo ), CLRDTR ) ;

   // purge any outstanding reads/writes and close device handle

   PurgeComm( COMDEV( npTTYInfo ), PURGE_TXABORT | PURGE_RXABORT |
                                   PURGE_TXCLEAR | PURGE_RXCLEAR ) ;
   CloseHandle( COMDEV( npTTYInfo ) ) ;

   // change the selectable items in the menu

   hMenu = GetMenu( hWnd ) ;
   EnableMenuItem( hMenu, IDM_DISCONNECT,
                   MF_GRAYED | MF_DISABLED | MF_BYCOMMAND ) ;
   EnableMenuItem( hMenu, IDM_CONNECT,
                   MF_ENABLED | MF_BYCOMMAND ) ;

   return ( TRUE ) ;

} // end of CloseConnection()
```

```
//-------------------------------------------------------------------------
//  int NEAR ReadCommBlock( HWND hWnd, LPSTR lpszBlock, int nMaxLength )
//
//  Description:
//     Reads a block from the COM port and stuffs it into
//     the provided buffer.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     LPSTR lpszBlock
//        block used for storage
//
//     int nMaxLength
//        max length of block to read
//
//  Win-32 Porting Issues:
//     - ReadComm() has been replaced by ReadFile() in Win-32.
//     - Overlapped I/O has been implemented.
//
//-------------------------------------------------------------------------

int NEAR ReadCommBlock( HWND hWnd, LPSTR lpszBlock, int nMaxLength )
{
   BOOL       fReadStat ;
   COMSTAT    ComStat ;
   DWORD      dwErrorFlags;
   DWORD      dwLength;
   DWORD      dwError;
   char       szError[ 10 ] ;
   NPTTYINFO  npTTYInfo ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   // only try to read number of bytes in queue
   ClearCommError( COMDEV( npTTYInfo ), &dwErrorFlags, &ComStat ) ;
   dwLength = min( (DWORD) nMaxLength, ComStat.cbInQue ) ;

   if (dwLength > 0)
   {
      fReadStat = ReadFile( COMDEV( npTTYInfo ), lpszBlock,
                            dwLength, &dwLength, &READ_OS( npTTYInfo ) ) ;
      if (!fReadStat)
      {
         if (GetLastError() == ERROR_IO_PENDING)
         {
            OutputDebugString("\n\rIO Pending");
            // We have to wait for read to complete.
            // This function will timeout according to the
            // CommTimeOuts.ReadTotalTimeoutConstant variable
            // Every time it times out, check for port errors
            while(!GetOverlappedResult( COMDEV( npTTYInfo ),
               &READ_OS( npTTYInfo ), &dwLength, TRUE ))
            {
               dwError = GetLastError();
               if(dwError == ERROR_IO_INCOMPLETE)
                  // normal result if not finished
                  continue;
               else
               {
                  // an error occurred, try to recover
                  wsprintf( szError, "<CE-%u>", dwError ) ;
                  //WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
                  ClearCommError( COMDEV( npTTYInfo ), &dwErrorFlags, &ComStat ) ;
                  if ((dwErrorFlags > 0) && DISPLAYERRORS( npTTYInfo ))
                  {
```

```
                        wsprintf( szError, "<CE-%u>", dwErrorFlags ) ;
                        //WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
                    }
                    break;
                }

            }

             }
        else
        {
            // some other error occurred
            dwLength = 0 ;
            ClearCommError( COMDEV( npTTYInfo ), &dwErrorFlags, &ComStat ) ;
            if ((dwErrorFlags > 0) && DISPLAYERRORS( npTTYInfo ))
            {
                    wsprintf( szError, "<CE-%u>", dwErrorFlags ) ;
                    //WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
            }
        }
    }
}

    return ( dwLength ) ;

} // end of ReadCommBlock()

//---------------------------------------------------------------------------
//  BOOL NEAR WriteCommBlock( HWND hWnd, BYTE *pByte )
//
//  Description:
//     Writes a block of data to the COM port specified in the associated
//     TTY info structure.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     BYTE *pByte
//        pointer to data to write to port
//
//  Win-32 Porting Issues:
//     - WriteComm() has been replaced by WriteFile() in Win-32.
//     - Overlapped I/O has been implemented.
//
//---------------------------------------------------------------------------

BOOL NEAR WriteCommBlock( HWND hWnd, LPSTR lpByte , DWORD dwBytesToWrite)
{

    BOOL        fWriteStat ;
    DWORD       dwBytesWritten ;
    NPTTYINFO   npTTYInfo ;
    DWORD       dwErrorFlags;
    DWORD        dwError;
    DWORD       dwBytesSent=0;
    COMSTAT     ComStat;
    char        szError[ 128 ] ;


    if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
       return ( FALSE ) ;

    fWriteStat = WriteFile( COMDEV( npTTYInfo ), lpByte, dwBytesToWrite,
                            &dwBytesWritten, &WRITE_OS( npTTYInfo ) ) ;

    // Note that normally the code will not execute the following
    // because the driver caches write operations. Small I/O requests
```

```
// (up to several thousand bytes) will normally be accepted
// immediately and WriteFile will return true even though an
// overlapped operation was specified

if (!fWriteStat)
{
   if(GetLastError() == ERROR_IO_PENDING)
   {
      // We should wait for the completion of the write operation
      // so we know if it worked or not

      // This is only one way to do this. It might be beneficial to
      // place the write operation in a separate thread
      // so that blocking on completion will not negatively
      // affect the responsiveness of the UI

      // If the write takes too long to complete, this
      // function will timeout according to the
      // CommTimeOuts.WriteTotalTimeoutMultiplier variable.
      // This code logs the timeout but does not retry
      // the write.

      while(!GetOverlappedResult( COMDEV( npTTYInfo ),
         &WRITE_OS( npTTYInfo ), &dwBytesWritten, TRUE ))
      {
         dwError = GetLastError();
         if(dwError == ERROR_IO_INCOMPLETE)
         {
            // normal result if not finished
            dwBytesSent += dwBytesWritten;
            continue;
         }
         else
         {
            // an error occurred, try to recover
            wsprintf( szError, "<CE-%u>", dwError ) ;
            WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
            ClearCommError( COMDEV( npTTYInfo ), &dwErrorFlags, &ComStat ) ;
            if ((dwErrorFlags > 0) && DISPLAYERRORS( npTTYInfo ))
            {
               wsprintf( szError, "<CE-%u>", dwErrorFlags ) ;
               WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
            }
            break;
         }
      }

      dwBytesSent += dwBytesWritten;

      if( dwBytesSent != dwBytesToWrite )
          wsprintf(szError,"\nProbable Write Timeout: Total of %ld bytes sent"
                                    , dwBytesSent);
      else
          wsprintf(szError,"\n%ld bytes written", dwBytesSent);

      OutputDebugString(szError);

   }
   else
   {
      // some other error occurred
      ClearCommError( COMDEV( npTTYInfo ), &dwErrorFlags, &ComStat ) ;
      if ((dwErrorFlags > 0) && DISPLAYERRORS( npTTYInfo ))
      {
         wsprintf( szError, "<CE-%u>", dwErrorFlags ) ;
         WriteTTYBlock( hWnd, szError, lstrlen( szError ) ) ;
      }
      return ( FALSE );
```

```
      }
   }
   return ( TRUE ) ;

} // end of WriteCommBlock()

//---------------------------------------------------------------------------
//  BOOL NEAR WriteTTYBlock( HWND hWnd, LPSTR lpBlock, int nLength )
//
//  Description:
//     Writes block to TTY screen.  Nothing fancy - just
//     straight TTY.
//
//  Parameters:
//     HWND hWnd
//        handle to TTY window
//
//     LPSTR lpBlock
//        far pointer to block of data
//
//     int nLength
//        length of block
//
//---------------------------------------------------------------------------

BOOL NEAR WriteTTYBlock( HWND hWnd, LPSTR lpBlock, int nLength )
{
   int        i ;
   NPTTYINFO  npTTYInfo ;
   RECT       rect ;

   if (NULL == (npTTYInfo = GETNPTTYINFO( hWnd )))
      return ( FALSE ) ;

   for (i = 0 ; i < nLength; i++)
   {
      switch (lpBlock[ i ])
      {
         case ASCII_BEL:
            // Bell
            MessageBeep( 0 ) ;
            break ;

         case ASCII_BS:
            // Backspace
            if (COLUMN( npTTYInfo ) > 0)
               COLUMN( npTTYInfo ) -- ;
            MoveTTYCursor( hWnd ) ;
            break ;

         case ASCII_CR:
            // Carriage return
            COLUMN( npTTYInfo ) = 0 ;
            MoveTTYCursor( hWnd ) ;
            if (!NEWLINE( npTTYInfo ))
               break;

            // fall through

         case ASCII_LF:
            // Line feed
            if (ROW( npTTYInfo )++ == MAXROWS - 1)
            {
               _fmemmove( (LPSTR) (SCREEN( npTTYInfo )),
                          (LPSTR) (SCREEN( npTTYInfo ) + MAXCOLS),
                          (MAXROWS - 1) * MAXCOLS ) ;
               _fmemset( (LPSTR) (SCREEN( npTTYInfo ) +
                                  (MAXROWS - 1) * MAXCOLS),' ', MAXCOLS ) ;
```

```
                InvalidateRect( hWnd, NULL, FALSE ) ;
                ROW( npTTYInfo )-- ;
             }
             MoveTTYCursor( hWnd ) ;
             break ;

          default:
             *(SCREEN( npTTYInfo ) + ROW( npTTYInfo ) * MAXCOLS +
                 COLUMN( npTTYInfo )) = lpBlock[ i ] ;
             rect.left = (COLUMN( npTTYInfo ) * XCHAR( npTTYInfo )) -
                         XOFFSET( npTTYInfo ) ;
             rect.right = rect.left + XCHAR( npTTYInfo ) ;
             rect.top = (ROW( npTTYInfo ) * YCHAR( npTTYInfo )) -
                        YOFFSET( npTTYInfo ) ;
             rect.bottom = rect.top + YCHAR( npTTYInfo ) ;
             InvalidateRect( hWnd, &rect, FALSE ) ;

             // Line wrap
             if (COLUMN( npTTYInfo ) < MAXCOLS - 1)
                COLUMN( npTTYInfo )++ ;
             else if (AUTOWRAP( npTTYInfo ))
                WriteTTYBlock( hWnd, "\r\n", 2 ) ;
             break;
       }
    }
    return ( TRUE ) ;

} // end of WriteTTYBlock()

//---------------------------------------------------------------------------
//  VOID NEAR GoModalDialogBoxParam( HINSTANCE hInstance,
//                                   LPCSTR lpszTemplate, HWND hWnd,
//                                   DLGPROC lpDlgProc, LPARAM lParam )
//
//  Description:
//     It is a simple utility function that simply performs the
//     MPI and invokes the dialog box with a DWORD paramter.
//
//  Parameters:
//     similar to that of DialogBoxParam() with the exception
//     that the lpDlgProc is not a procedure instance
//
//---------------------------------------------------------------------------

VOID NEAR GoModalDialogBoxParam( HINSTANCE hInstance, LPCSTR lpszTemplate,
                                 HWND hWnd, DLGPROC lpDlgProc, LPARAM lParam )
{
   DLGPROC  lpProcInstance ;

   lpProcInstance = (DLGPROC) MakeProcInstance( (FARPROC) lpDlgProc,
                                                hInstance ) ;
   DialogBoxParam( hInstance, lpszTemplate, hWnd, lpProcInstance, lParam ) ;
   FreeProcInstance( (FARPROC) lpProcInstance ) ;

} // end of GoModalDialogBoxParam()
```

```
//---------------------------------------------------------------------------
//  BOOL FAR PASCAL AboutDlgProc( HWND hDlg, UINT uMsg,
//                                WPARAM wParam, LPARAM lParam )
//
//  Description:
//     Simulates the Windows System Dialog Box.
//
//  Parameters:
//     Same as standard dialog procedures.
//
//---------------------------------------------------------------------------

BOOL FAR PASCAL AboutDlgProc( HWND hDlg, UINT uMsg,
                              WPARAM wParam, LPARAM lParam )
{
   switch (uMsg)
   {
      case WM_INITDIALOG:
      {
         char         szBuffer[ MAXLEN_TEMPSTR ], szTemp[ MAXLEN_TEMPSTR ];
         WORD         wRevision, wVersion ;

#ifdef ABOUTDLG_USEBITMAP
         // if we are using the bitmap, hide the icon

         ShowWindow( GetDlgItem( hDlg, IDD_ABOUTICON ), SW_HIDE ) ;
#endif
         // sets up the version number for Windows

         wVersion = LOWORD( GetVersion() ) ;
         wRevision = HIBYTE( wVersion ) ;
         wVersion = LOBYTE( wVersion ) ;

         GetDlgItemText( hDlg, IDD_TITLELINE, szTemp, sizeof( szTemp ) ) ;
         wsprintf( szBuffer, szTemp, wVersion, wRevision ) ;
         SetDlgItemText( hDlg, IDD_TITLELINE, szBuffer ) ;

         // sets up version number for TTY

         GetDlgItemText( hDlg, IDD_VERSION, szTemp, sizeof( szTemp ) ) ;
         wsprintf( szBuffer, szTemp, VER_MAJOR, VER_MINOR, VER_BUILD ) ;
         SetDlgItemText( hDlg, IDD_VERSION, (LPSTR) szBuffer ) ;

         // get by-line

         LoadString( GETHINST( hDlg ), IDS_BYLINE, szBuffer,
                     sizeof( szBuffer ) ) ;
         SetDlgItemText( hDlg, IDD_BYLINE, szBuffer ) ;

         SetDlgItemText( hDlg, IDD_WINDOWSMODE, "NT Mode" ) ;

      }
      return ( TRUE ) ;

#ifdef ABOUTDLG_USEBITMAP
      // used to paint the bitmap

      case WM_PAINT:
      {
         HBITMAP      hBitMap ;
         HDC          hDC, hMemDC ;
         PAINTSTRUCT  ps ;

         // load bitmap and display it

         hDC = BeginPaint( hDlg, &ps ) ;
         if (NULL != (hMemDC = CreateCompatibleDC( hDC )))
         {
```

```
            hBitMap = LoadBitmap( GETHINST( hDlg ),
                                  MAKEINTRESOURCE( TTYBITMAP ) ) ;
            hBitMap = SelectObject( hMemDC, hBitMap ) ;
            BitBlt( hDC, 10, 10, 64, 64, hMemDC, 0, 0, SRCCOPY ) ;
            DeleteObject( SelectObject( hMemDC, hBitMap ) ) ;
            DeleteDC( hMemDC ) ;
         }
         EndPaint( hDlg, &ps ) ;
      }
      break ;
#endif

      case WM_COMMAND:
         if (LOWORD( wParam ) == IDD_OK)
         {
            EndDialog( hDlg, TRUE ) ;
            return ( TRUE ) ;
         }
         break;
   }
   return ( FALSE ) ;

} // end of AboutDlgProc()

//---------------------------------------------------------------------------
//  VOID NEAR FillComboBox( HINSTANCE hInstance, HWND hCtrlWnd, int nIDString,
//                          WORD NEAR *npTable, WORD wTableLen,
//                          WORD wCurrentSetting )
//
//  Description:
//     Fills the given combo box with strings from the resource
//     table starting at nIDString.  Associated items are
//     added from given table.  The combo box is notified of
//     the current setting.
//
//  Parameters:
//     HINSTANCE hInstance
//        handle to application instance
//
//     HWND hCtrlWnd
//        handle to combo box control
//
//     int nIDString
//        first resource string id
//
//     DWORD NEAR *npTable
//        near point to table of associated values
//
//     WORD wTableLen
//        length of table
//
//     DWORD dwCurrentSetting
//        current setting (for combo box selection)
//
//---------------------------------------------------------------------------

VOID NEAR FillComboBox( HINSTANCE hInstance, HWND hCtrlWnd, int nIDString,
                        DWORD NEAR *npTable, WORD wTableLen,
                        DWORD dwCurrentSetting )
{
   char  szBuffer[ MAXLEN_TEMPSTR ] ;
   WORD  wCount, wPosition ;

   for (wCount = 0; wCount < wTableLen; wCount++)
   {
      // load the string from the string resources and
      // add it to the combo box
```

```
      LoadString( hInstance, nIDString + wCount, szBuffer, sizeof( szBuffer ) ) ;
      wPosition = LOWORD( SendMessage( hCtrlWnd, CB_ADDSTRING, 0,
                                       (LPARAM) (LPSTR) szBuffer ) ) ;

      // use item data to store the actual table value

      SendMessage( hCtrlWnd, CB_SETITEMDATA, (WPARAM) wPosition,
                   (LPARAM) *(npTable + wCount) ) ;

      // if this is our current setting, select it

      if (*(npTable + wCount) == dwCurrentSetting)
         SendMessage( hCtrlWnd, CB_SETCURSEL, (WPARAM) wPosition, 0L ) ;
   }

} // end of FillComboBox()

//---------------------------------------------------------------------------
//  BOOL NEAR SettingsDlgInit( HWND hDlg )
//
//  Description:
//     Puts current settings into dialog box (via CheckRadioButton() etc.)
//
//  Parameters:
//     HWND hDlg
//        handle to dialog box
//
//  Win-32 Porting Issues:
//     - Constants require DWORD arrays for baud rate table, etc.
//     - There is no "MAXCOM" function in Win-32.  Number of COM ports
//       is assumed to be 4.
//
//---------------------------------------------------------------------------

BOOL NEAR SettingsDlgInit( HWND hDlg )
{
   char       szBuffer[ MAXLEN_TEMPSTR ], szTemp[ MAXLEN_TEMPSTR ] ;
   NPTTYINFO  npTTYInfo ;
   WORD       wCount, wMaxCOM, wPosition ;

   if (NULL == (npTTYInfo = (NPTTYINFO) GET_PROP( hDlg, ATOM_TTYINFO )))
      return ( FALSE ) ;

   wMaxCOM = MAXPORTS ;

   // load the COM prefix from resources

   LoadString( GETHINST( hDlg ), IDS_COMPREFIX, szTemp, sizeof( szTemp ) ) ;

   // fill port combo box and make initial selection

   for (wCount = 0; wCount < wMaxCOM; wCount++)
   {
      wsprintf( szBuffer, "%s%d", (LPSTR) szTemp, wCount + 1 ) ;
      SendDlgItemMessage( hDlg, IDD_PORTCB, CB_ADDSTRING, 0,
                          (LPARAM) (LPSTR) szBuffer ) ;
   }

   SendDlgItemMessage( hDlg, IDD_PORTCB, CB_SETCURSEL,
                       (WPARAM) (PORT( npTTYInfo ) - 1), 0L ) ;

   // disable COM port combo box if connection has already been
   // established (e.g. OpenComm() already successful)

   EnableWindow( GetDlgItem( hDlg, IDD_PORTCB ), !CONNECTED( npTTYInfo ) ) ;

   // fill baud combo box and make initial selection
```

```
FillComboBox( GETHINST( hDlg ), GetDlgItem( hDlg, IDD_BAUDCB ),
              IDS_BAUD110, BaudTable,
              sizeof( BaudTable ) / sizeof( BaudTable[ 0 ] ),
              BAUDRATE( npTTYInfo ) ) ;

// fill data bits combo box and make initial selection

for (wCount = 5; wCount < 9; wCount++)
{
   wsprintf( szBuffer, "%d", wCount ) ;
   wPosition = LOWORD( SendDlgItemMessage( hDlg, IDD_DATABITSCB,
                                           CB_ADDSTRING, 0,
                                           (LPARAM) (LPSTR) szBuffer ) ) ;

   // if current selection, tell the combo box

   if (wCount == BYTESIZE( npTTYInfo ))
      SendDlgItemMessage( hDlg, IDD_DATABITSCB, CB_SETCURSEL,
                          (WPARAM) wPosition, 0L ) ;
}

// fill parity combo box and make initial selection

FillComboBox( GETHINST( hDlg ), GetDlgItem( hDlg, IDD_PARITYCB ),
              IDS_PARITYNONE, ParityTable,
              sizeof( ParityTable ) / sizeof( ParityTable[ 0 ] ),
              PARITY( npTTYInfo ) ) ;

// fill stop bits combo box and make initial selection

FillComboBox( GETHINST( hDlg ), GetDlgItem( hDlg, IDD_STOPBITSCB ),
              IDS_ONESTOPBIT, StopBitsTable,
              sizeof( StopBitsTable ) / sizeof ( StopBitsTable[ 0 ] ),
              STOPBITS( npTTYInfo ) ) ;

// initalize the flow control settings

CheckDlgButton( hDlg, IDD_DTRDSR,
                (FLOWCTRL( npTTYInfo ) & FC_DTRDSR) > 0 ) ;
CheckDlgButton( hDlg, IDD_RTSCTS,
                (FLOWCTRL( npTTYInfo ) & FC_RTSCTS) > 0 ) ;
CheckDlgButton( hDlg, IDD_XONXOFF,
                (FLOWCTRL( npTTYInfo ) & FC_XONXOFF) > 0 ) ;

// other TTY settings

CheckDlgButton( hDlg, IDD_AUTOWRAP, AUTOWRAP( npTTYInfo ) ) ;
CheckDlgButton( hDlg, IDD_NEWLINE, NEWLINE( npTTYInfo ) ) ;
CheckDlgButton( hDlg, IDD_LOCALECHO, LOCALECHO( npTTYInfo ) ) ;

// control options

// "Use CN_RECEIVE" is not valid under Win-32

EnableWindow( GetDlgItem( hDlg, IDD_USECNRECEIVE ), FALSE ) ;

CheckDlgButton( hDlg, IDD_DISPLAYERRORS, DISPLAYERRORS( npTTYInfo ) ) ;

return ( TRUE ) ;

} // end of SettingsDlgInit()
```

```
//---------------------------------------------------------------------------
//  BOOL NEAR SelectTTYFont( HWND hDlg )
//
//  Description:
//     Selects the current font for the TTY screen.
//     Uses the Common Dialog ChooseFont() API.
//
//  Parameters:
//     HWND hDlg
//        handle to settings dialog
//
//---------------------------------------------------------------------------

BOOL NEAR SelectTTYFont( HWND hDlg )
{
   CHOOSEFONT  cfTTYFont ;
   NPTTYINFO   npTTYInfo ;

   if (NULL == (npTTYInfo = (NPTTYINFO) GET_PROP( hDlg, ATOM_TTYINFO )))
      return ( FALSE ) ;

   cfTTYFont.lStructSize    = sizeof( CHOOSEFONT ) ;
   cfTTYFont.hwndOwner      = hDlg ;
   cfTTYFont.hDC            = NULL ;
   cfTTYFont.rgbColors      = FGCOLOR( npTTYInfo ) ;
   cfTTYFont.lpLogFont      = &LFTTYFONT( npTTYInfo ) ;
   cfTTYFont.Flags          = CF_SCREENFONTS | CF_FIXEDPITCHONLY |
                              CF_EFFECTS | CF_INITTOLOGFONTSTRUCT ;
   cfTTYFont.lCustData      = 0 ;
   cfTTYFont.lpfnHook       = NULL ;
   cfTTYFont.lpTemplateName = NULL ;
   cfTTYFont.hInstance      = GETHINST( hDlg ) ;

   if (ChooseFont( &cfTTYFont ))
   {
     FGCOLOR( npTTYInfo ) = cfTTYFont.rgbColors ;
     ResetTTYScreen( GetParent( hDlg ), npTTYInfo ) ;
   }

   return ( TRUE ) ;

} // end of SelectTTYFont()

//---------------------------------------------------------------------------
//  BOOL NEAR SettingsDlgTerm( HWND hDlg )
//
//  Description:
//     Puts dialog contents into TTY info structure.
//
//  Parameters:
//     HWND hDlg
//        handle to settings dialog
//
//  Win-32 Porting Issues:
//     - Baud rate requires DWORD values.
//
//---------------------------------------------------------------------------

BOOL NEAR SettingsDlgTerm( HWND hDlg )
{
   NPTTYINFO  npTTYInfo ;
   WORD       wSelection ;

   if (NULL == (npTTYInfo = (NPTTYINFO) GET_PROP( hDlg, ATOM_TTYINFO )))
      return ( FALSE ) ;

   // get port selection
```

```
   PORT( npTTYInfo ) =
      LOBYTE( LOWORD( SendDlgItemMessage( hDlg, IDD_PORTCB,
                                          CB_GETCURSEL,
                                          0, 0L ) ) + 1 ) ;
   // get baud rate selection

   wSelection =
      LOWORD( SendDlgItemMessage( hDlg, IDD_BAUDCB, CB_GETCURSEL,
                                  0, 0L ) ) ;
   BAUDRATE( npTTYInfo ) =
      SendDlgItemMessage( hDlg, IDD_BAUDCB, CB_GETITEMDATA,
                          (WPARAM) wSelection, 0L ) ;

   // get data bits selection

   BYTESIZE( npTTYInfo ) =
      LOBYTE( LOWORD( SendDlgItemMessage( hDlg, IDD_DATABITSCB,
                                          CB_GETCURSEL,
                                          0, 0L ) ) + 5 ) ;


   // get parity selection

   wSelection =
      LOWORD( SendDlgItemMessage( hDlg, IDD_PARITYCB, CB_GETCURSEL,
                                  0, 0L ) ) ;
   PARITY( npTTYInfo ) =
      LOBYTE( LOWORD( SendDlgItemMessage( hDlg, IDD_PARITYCB,
                                          CB_GETITEMDATA,
                                          (WPARAM) wSelection,
                                          0L ) ) ) ;

   // get stop bits selection

   wSelection =
      LOWORD( SendDlgItemMessage( hDlg, IDD_STOPBITSCB, CB_GETCURSEL,
                                  0, 0L ) ) ;
   STOPBITS( npTTYInfo ) =
      LOBYTE( LOWORD( SendDlgItemMessage( hDlg, IDD_STOPBITSCB,
                                          CB_GETITEMDATA,
                                          (WPARAM) wSelection, 0L ) ) ) ;

   // get flow control settings

   FLOWCTRL( npTTYInfo ) = 0 ;
   if (IsDlgButtonChecked( hDlg, IDD_DTRDSR ))
      FLOWCTRL( npTTYInfo ) |= FC_DTRDSR ;
   if (IsDlgButtonChecked( hDlg, IDD_RTSCTS ))
      FLOWCTRL( npTTYInfo ) |= FC_RTSCTS ;
   if (IsDlgButtonChecked( hDlg, IDD_XONXOFF ))
      FLOWCTRL( npTTYInfo ) |= FC_XONXOFF ;

   // get other various settings

   AUTOWRAP( npTTYInfo ) = IsDlgButtonChecked( hDlg, IDD_AUTOWRAP ) ;
   NEWLINE( npTTYInfo ) = IsDlgButtonChecked( hDlg, IDD_NEWLINE ) ;
   LOCALECHO( npTTYInfo ) = IsDlgButtonChecked( hDlg, IDD_LOCALECHO ) ;

   // control options

   USECNRECEIVE( npTTYInfo ) = IsDlgButtonChecked( hDlg, IDD_USECNRECEIVE ) ;
   DISPLAYERRORS( npTTYInfo ) = IsDlgButtonChecked( hDlg, IDD_DISPLAYERRORS ) ;

   return ( TRUE ) ;

} // end of SettingsDlgTerm()
```

```
//---------------------------------------------------------------------------
//  BOOL FAR PASCAL SettingsDlgProc( HWND hDlg, UINT uMsg,
//                                   WPARAM wParam, LPARAM lParam )
//
//  Description:
//     This handles all of the user preference settings for
//     the TTY.
//
//  Parameters:
//     same as all dialog procedures
//
//  Win-32 Porting Issues:
//     - npTTYInfo is a DWORD in Win-32.
//
//---------------------------------------------------------------------------

BOOL FAR PASCAL SettingsDlgProc( HWND hDlg, UINT uMsg,
                                 WPARAM wParam, LPARAM lParam )
{
   switch (uMsg)
   {
      case WM_INITDIALOG:
      {
         NPTTYINFO  npTTYInfo ;

         // get & save pointer to TTY info structure

         npTTYInfo = (NPTTYINFO) lParam ;

         SET_PROP( hDlg, ATOM_TTYINFO, (HANDLE) npTTYInfo ) ;

         return ( SettingsDlgInit( hDlg ) ) ;
      }

      case WM_COMMAND:
         switch ( LOWORD( wParam ))
         {
            case IDD_FONT:
               return ( SelectTTYFont( hDlg ) ) ;

            case IDD_OK:
               // Copy stuff into structure
               SettingsDlgTerm( hDlg ) ;
               EndDialog( hDlg, TRUE ) ;
               return ( TRUE ) ;

            case IDD_CANCEL:
               // Just end
               EndDialog( hDlg, TRUE ) ;
               return ( TRUE ) ;
         }
         break;

      case WM_DESTROY:
         REMOVE_PROP( hDlg, ATOM_TTYINFO ) ;
         break ;
   }
   return ( FALSE ) ;

} // end of SettingsDlgProc()
```

```
//*********************************************************************
//BOOL CALLBACK DialingDlgProc( HWND hDlg, UINT uMsg,
//                              WPARAM wParam, LPARAM lParam )
// Description:
//     This Dialog box handels dialing of phone number
//
//  Parameters:
//     same as all dialog procedures
//
//  Returns:
//     TRUE if message has been processed, else FALS
//*********************************************************************
BOOL CALLBACK DialingDlgProc( HWND hDlg, UINT uMsg,
                              WPARAM wParam, LPARAM lParam )
{
        char                    Text[80];
        unsigned int            i;
        BYTE                    b;
        int                     NoCopied;

    switch (uMsg)
   {
            case WM_COMMAND:
                    switch(LOWORD(wParam))
                    {
                       case ID_OK:
                                  {
                                     for (i=0;i<80;i++)
                                          Text[i] = 0;
                                         NoCopied =
                                            GetDlgItemText(hDlg,IDC_NUMBER,Text,80);
                                      b=43;
                                      for(i=0;i<3;i++)
                                                WriteCommBlock( hwnd, &b,1);
                                      b = 65;
                                      WriteCommBlock( hwnd, &b,1);
                                      b = 84;
                                      WriteCommBlock( hwnd, &b,1);
                                      b = 68;
                                      WriteCommBlock( hwnd, &b,1);
                                      b = 80;
                                      WriteCommBlock( hwnd, &b,1);

                                      for (i=0;i<strlen(Text);i++)
                                                WriteCommBlock( hwnd, &Text[i],1);
                                       b = 13;
                                       WriteCommBlock( hwnd, &b,1);

                                       EndDialog(hDlg,TRUE);
                                       return TRUE;
                                  }
                 case ID_CANCEL:
                                  EndDialog(hDlg,TRUE);
                                  return TRUE;
                          }
                }

        return (FALSE);

 }
```

```
//*************************************************************************
//BOOL CALLBACK InformationDlgProc( HWND hDlg, UINT uMsg,
//                                  WPARAM wParam, LPARAM lParam )
// Description:
//     This Dialog box displays information tranfer rate obtained
//
//  Parameters:
//     same as all dialog procedures
//
//  Returns:
//     TRUE if message has been processed, else FALS
//*************************************************************************
BOOL CALLBACK InformationDlgProc( HWND hDlg, UINT uMsg,
                                  WPARAM wParam, LPARAM lParam )
{

        char TextFileSize[80],TextCompressed[80],TextBaudRate[80];
        char TextDataReduction[80],TextInformationRate[80];

        long DataReduction,InformationRate;

        ldiv_t divt;

        NPTTYINFO   npTTYInfo ;

        npTTYInfo = GETNPTTYINFO( hwnd );

        wsprintf( TextFileSize, "%ld", file_size );
       wsprintf( TextCompressed, "%ld", compressed_count);
        wsprintf( TextBaudRate, "%ld", BAUDRATE( npTTYInfo ));

        divt = ldiv((100*(file_size-compressed_count)),file_size);

        DataReduction = divt.quot;

        wsprintf( TextDataReduction, "%ld", DataReduction);

        divt = ldiv((BAUDRATE( npTTYInfo )*file_size),(compressed_count));

        InformationRate = divt.quot;

        wsprintf( TextInformationRate, "%ld", InformationRate);


    switch (uMsg)
   {
              case WM_INITDIALOG:
                     {
                            SetDlgItemText(hDlg,IDC_RATE,TextFileSize,
                                                   strlen(TextFileSize));
                            SetDlgItemText(hDlg,IDC_COMPRESSED,TextCompressed,
                                                   strlen(TextCompressed));
                            SetDlgItemText(hDlg,IDC_MODEM_RATE,TextBaudRate,
                                                   strlen(TextBaudRate));
                            SetDlgItemText(hDlg,IDC_DATA_REDUCTION,TextDataReduction,
                                                   strlen(TextDataReduction));
                            SetDlgItemText(hDlg,IDC_INFORMATION_RATE,
                                                   TextInformationRate,
                                                   strlen(TextInformationRate));
                            break;
                     }
           case WM_COMMAND:
                     {

                      EndDialog(hDlg,TRUE);
                      return TRUE;
                     }
```

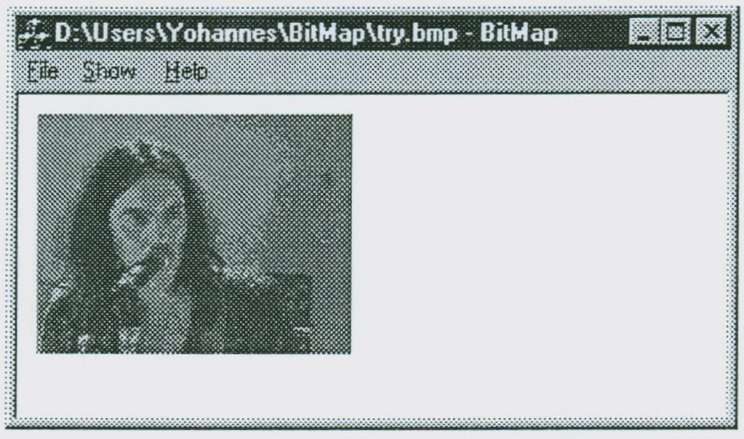
```
            }

        return (FALSE);

}


//***************************************************************************
//BOOL CALLBACK AttensionDlgProc( HWND hDlg, UINT uMsg,
//                                WPARAM wParam, LPARAM lParam )
// Description:
//     This Dialog box sends attension command to the modem
//
//  Parameters:
//     same as all dialog procedures
//
//  Returns:
//     TRUE if message has been processed, else FALS
//***************************************************************************
BOOL CALLBACK AttensionDlgProc( HWND hDlg, UINT uMsg,
                                WPARAM wParam, LPARAM lParam )
{
        char                    Text[80];
        unsigned int            i;
        BYTE                    b;

    switch (uMsg)
   {
            case WM_COMMAND:
                        switch(LOWORD(wParam))
                        {
                           case ID_OK:
                                        {
                                           for (i=0;i<80;i++)
                                               Text[i] = 0;
                                            GetDlgItemText(hDlg,IDC_NUMBER,Text,80);
                                            b=43;
                                            for(i=0;i<3;i++)
                                               WriteCommBlock( hwnd, &b,1);
                                            b = 65;
                                            WriteCommBlock( hwnd, &b,1);
                                            b = 84;
                                            WriteCommBlock( hwnd, &b,1);

                                            for (i=0;i<strlen(Text);i++)
                                              WriteCommBlock( hwnd, &Text[i],1);
                                            b = 13;
                                            WriteCommBlock( hwnd, &b,1);

                                            EndDialog(hDlg,TRUE);
                                            return TRUE;
                                        }
                 case ID_CANCEL:
                                        EndDialog(hDlg,TRUE);
                                        return TRUE;
                        }
                }

        return (FALSE);

}
```

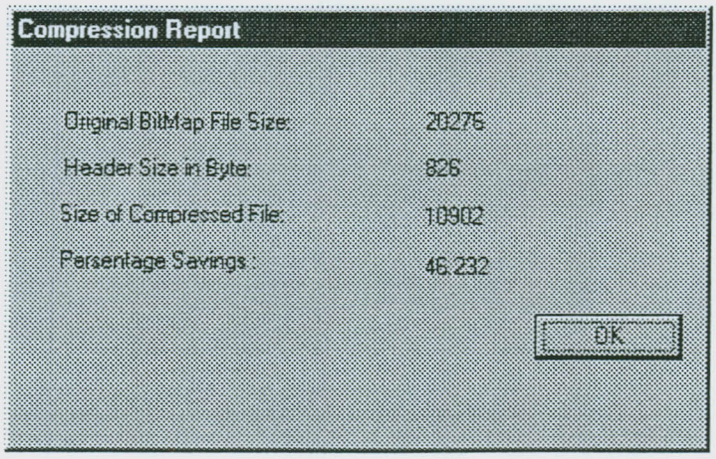
```
//************************************************************************
//  DWORD FAR PASCAL CommWatchProc( LPSTR lpData )
//
//  Description:
//     A secondary thread that will watch for COMM events.
//
//  Parameters:
//     LPSTR lpData
//        32-bit pointer argument
//
//  Win-32 Porting Issues:
//     - Added this thread to watch the communications device and
//       post notifications to the associated window.
//
//************************************************************************

DWORD FAR PASCAL CommWatchProc( LPSTR lpData )
{
   DWORD       dwEvtMask ;
   NPTTYINFO   npTTYInfo = (NPTTYINFO) lpData ;
   OVERLAPPED  os ;
       int         nLength ;
   BYTE        abIn[ MAXBLOCK + 1] ;

   memset( &os, 0, sizeof( OVERLAPPED ) ) ;

   // create I/O event used for overlapped read

   os.hEvent = CreateEvent( NULL,    // no security
                            TRUE,    // explicit reset req
                            FALSE,   // initial event reset
                            NULL ) ; // no name
   if (os.hEvent == NULL)
   {
      MessageBox( NULL, "Failed to create event for thread!",
                                          "Text Huffman Error!",
                                           MB_ICONEXCLAMATION | MB_OK ) ;
      return ( FALSE ) ;
   }

   if (!SetCommMask( COMDEV( npTTYInfo ), EV_RXCHAR ))
      return ( FALSE ) ;

   while ( CONNECTED( npTTYInfo ) )
   {
      dwEvtMask = 0 ;

      WaitCommEvent( COMDEV( npTTYInfo ), &dwEvtMask, NULL );

      if ((dwEvtMask & EV_RXCHAR) == EV_RXCHAR)
      {
         do
         {
            if (nLength = ReadCommBlock( hTTYWnd, abIn, MAXBLOCK ))
            {
                           if (Huffman_Coding){
                                  //With Huffman Coding applied
                                  decompress_image (hTTYWnd,
                                        (LPBYTE)abIn, nLength );
                           }
                           else
                           {
                                  //No Huffman Coding is applied
                                  WriteTTYBlock( hTTYWnd, (LPSTR) abIn, nLength ) ;
                           }

               // force a paint
```

```
                UpdateWindow( hTTYWnd ) ;
              }
            }
            while ( nLength > 0 ) ;
             }

    }

    // get rid of event handle

    CloseHandle( os.hEvent ) ;

    // clear information in structure (kind of a "we're done flag")

    THREADID( npTTYInfo ) = 0 ;
    HTHREAD( npTTYInfo ) = NULL ;

    return( TRUE ) ;

} // end of CommWatchProc()

//---------------------------------------------------------------------------
//  End of File: Huffman.c
//---------------------------------------------------------------------------
```

APPENDIX 7

Bitmap Graphics Compressor Program





1. Header for DIB Class

```
////////////////////////////////////////////////////////////
// CDIB.H: Header file for the DIB class.
////////////////////////////////////////////////////////////
#ifndef __CDIB_H
#define __CDIB_H
class CDib : public CObject
{
protected:
    LPBITMAPFILEHEADER m_pBmFileHeader;
    LPBITMAPINFO m_pBmInfo;
    LPBITMAPINFOHEADER m_pBmInfoHeader;
    RGBQUAD* m_pRGBTable;
    BYTE* m_pDibBits;
    UINT m_numColors;
    short           father[512],decomp_tree[512];
    unsigned short  code[256], heap_length;
    unsigned long   compress_charcount, heap[257];
    unsigned char   code_length[256];
    long            frequency_count[512];
public:
    float           savings;
    unsigned short  header_charcount;
    unsigned long   output_characters,file_size;
    BOOLEAN         Color24;
public:
    CDib(const char* fileName);
    ~CDib();
    DWORD GetDibSizeImage();
    UINT GetDibWidth();
    UINT GetDibHeight();
    UINT GetDibNumColors();
    LPBITMAPINFOHEADER GetDibInfoHeaderPtr();
    LPBITMAPINFO GetDibInfoPtr();
    LPRGBQUAD GetDibRGBTablePtr();
    BYTE* GetDibBitsPtr();
    BITMAPFILEHEADER bmFileHeaderMain;
    void get_frequency_count (const char* fileName);
    void reheap (unsigned short heap_entry);
    void build_initial_heap ();
    void build_code_tree ();
    unsigned short  generate_code_table ();
    void  compression_report ();
    unsigned short compress_image (const char* fileName,const char *fileDest);
    void build_decomp_tree ();
    void decompress_image (const char* filename, BYTE* PDib);
    BITMAPFILEHEADER Get_BitMap_FileHeader();
protected:
    void LoadBitmapFile(const char* fileName);
};
#endif
```

2. Implementation of the DIB Class

```
////////////////////////////////////////////////////////////
// CDIB.CPP: Implementation file for the DIB class.
////////////////////////////////////////////////////////////
#include "stdafx.h"
#include "cdib.h"
#include "windowsx.h"
////////////////////////////////////////////////////////////
// CDib::CDib()
////////////////////////////////////////////////////////////
CDib::CDib(const char* fileName)
{
    // Load the bitmap and initialize
    // the class's data members.
    LoadBitmapFile(fileName);
}
////////////////////////////////////////////////////////////
// CDib::~CDib()
////////////////////////////////////////////////////////////
CDib::~CDib()
{
    // Free the memory assigned to the bitmap.
    GlobalFreePtr(m_pBmInfo);
}
////////////////////////////////////////////////////////////
// CDib::LoadBitmapFile()
//
// This function loads a DIB from disk into memory. It
// also initializes the various class data members.
////////////////////////////////////////////////////////////
void CDib::LoadBitmapFile
    (const char* fileName)
{
    // Construct and open a file object.
    CFile file(fileName, CFile::modeRead);

    // Read the bitmap's file header into memory.
    //BITMAPFILEHEADER bmFileHeader;
    BITMAPINFO       bmFileInfo;
    BYTE* pDib;
    unsigned long file_size;
    file.Read((void*)&bmFileHeaderMain, sizeof(bmFileHeaderMain));
    // Check whether the file is really a bitmap.
    if (bmFileHeaderMain.bfType != 0x4d42)
    {
        AfxMessageBox("Not a bitmap file");
        m_pBmFileHeader = 0;
        m_pBmInfo = 0;
        m_pBmInfoHeader = 0;
        m_pRGBTable = 0;
        m_pDibBits = 0;
        m_numColors = 0;
    }
    // If the file checks out okay, continue loading.
    else
    {
        // Calculate the size of the DIB, which is the
        // file size minus the size of the file header.
        if (bmFileHeaderMain.bfReserved1 != 10) {
            DWORD fileLength = file.GetLength();
            DWORD dibSize = fileLength - sizeof(BITMAPFILEHEADER);
        // Allocate enough memory to fit the bitmap.
```

```
            pDib =
                (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE, dibSize);

            // Read the bitmap into memory and close the file.
            file.Read((void*)pDib, dibSize);
            file.Close();
        }
        else
        {
            file.Read((void*)&bmFileInfo,sizeof(bmFileInfo));
            file.Read((void*)&file_size,sizeof(file_size));
            file.Close();

            pDib =
                (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE, file_size-
                                      sizeof(bmFileHeaderMain));
            decompress_image(fileName,pDib);

        }
        // Initialize pointers to the bitmap's BITMAPINFO
        // and BITMAPINFOHEADER structures.
        m_pBmInfo = (LPBITMAPINFO) pDib;
        m_pBmInfoHeader = (LPBITMAPINFOHEADER) pDib;

        // Calculate a pointer to the bitmap's color table.
        m_pRGBTable =
            (RGBQUAD*)(pDib + m_pBmInfoHeader->biSize);
        // Get the number of colors in the bitmap.
        int m_numColors = GetDibNumColors();
        // Calculate the bitmap image's size.
        m_pBmInfoHeader->biSizeImage =
            GetDibSizeImage();

        // Make sure the biClrUsed field
        // is initialized properly.
        if (m_pBmInfoHeader->biClrUsed == 0)
            m_pBmInfoHeader->biClrUsed = m_numColors;
        // Calculate a pointer to the bitmap's actual data.
        DWORD clrTableSize = m_numColors * sizeof(RGBQUAD);
        m_pDibBits =
            pDib + m_pBmInfoHeader->biSize + clrTableSize;

    }
}
////////////////////////////////////////////////////////////
// CDib::GetDibSizeImage()
//
// This function calculates and returns the size of the
// bitmap's image in bytes.
////////////////////////////////////////////////////////////
DWORD CDib::GetDibSizeImage()
{
    // If the bitmap's biSizeImage field contains
    // invalid information, calculate the correct size.
    if (m_pBmInfoHeader->biSizeImage == 0)
    {
        // Get the width in bytes of a single row.
        DWORD byteWidth = (DWORD) GetDibWidth();

        // Get the height of the bitmap.
        DWORD height = (DWORD) GetDibHeight();
```

```
        // Multiply the byte width by the number of rows.
        DWORD imageSize = byteWidth * height;

        return imageSize;
    }
    // Otherwise, just return the size stored in
    // the BITMAPINFOHEADER structure.
    else
        return m_pBmInfoHeader->biSizeImage;
}
////////////////////////////////////////////////////////////
// CDib::GetDibWidth()
//
// This function returns the width in bytes of a single
// row in the bitmap.
////////////////////////////////////////////////////////////
UINT CDib::GetDibWidth()
{
    return (UINT) m_pBmInfoHeader->biWidth;
}

////////////////////////////////////////////////////////////
// CDib::GetDibHeight()
//
// This function returns the bitmap's height in pixels.
////////////////////////////////////////////////////////////
UINT CDib::GetDibHeight()
{
    return (UINT) m_pBmInfoHeader->biHeight;
}

////////////////////////////////////////////////////////////
// CDib::GetDibNumColors()
//
// This function returns the number of colors in the
// bitmap.
////////////////////////////////////////////////////////////
UINT CDib::GetDibNumColors()
{
    if ((m_pBmInfoHeader->biClrUsed == 0) &&
          (m_pBmInfoHeader->biBitCount < 9))
        return (1 << m_pBmInfoHeader->biBitCount);
    else
        return (int) m_pBmInfoHeader->biClrUsed;
}

////////////////////////////////////////////////////////////
// CDib::GetDibInfoHeaderPtr()
//
// This function returns a pointer to the bitmap's
// BITMAPINFOHEADER structure.
////////////////////////////////////////////////////////////
LPBITMAPINFOHEADER CDib::GetDibInfoHeaderPtr()
{
    return m_pBmInfoHeader;
}
////////////////////////////////////////////////////////////
// CDib::GetDibInfoPtr()
//
// This function returns a pointer to the bitmap's
```

```
// BITMAPINFO structure.
////////////////////////////////////////////////////////////
LPBITMAPINFO CDib::GetDibInfoPtr()
{
    return m_pBmInfo;
}
////////////////////////////////////////////////////////////
// CDib::GetDibRGBTablePtr()
//
// This function returns a pointer to the bitmap's
// color table.
////////////////////////////////////////////////////////////
LPRGBQUAD CDib::GetDibRGBTablePtr()
{
    return m_pRGBTable;
}
////////////////////////////////////////////////////////////
// CDib::GetDibBitsPtr()
//
// This function returns a pointer to the bitmap's
// actual image data.
////////////////////////////////////////////////////////////
BYTE* CDib::GetDibBitsPtr()
{
    return m_pDibBits;
}


////////////////////////////////////////////////////////////////////
// This function performs the actual data compression.
////////////////////////////////////////////////////////////////////

unsigned short CDib::compress_image (const char* fileName,const char *fileDest)
{
   register unsigned int    thebyte = 0;
   register short           loop1;
   register unsigned short  current_code;
   register unsigned long   counter;
   unsigned short           current_length, dvalue;
   unsigned long            curbyte = 0;
   short                    curbit = 7;
   BYTE                     Outbuff[20000];
   register unsigned long   loop;
   unsigned long            offset;

   //BITMAPFILEHEADER   bmFileHeader;
   BITMAPINFO      bmFileInfo;


   //Get Frequency distribution of each character
   get_frequency_count (fileName);

   //build initial heap
   build_initial_heap();

   //build code tree
   build_code_tree();

   //generate_code_table

   if (!generate_code_table()){
```

```
    AfxMessageBox("Code Value Out of Range . Sorry Cannot Compress !");
}
else
{

    //Construct and open a file objects for compression.
    CFile file(fileName, CFile::modeRead),fileD(fileDest,CFile::modeCreate
              | CFile::modeWrite);

    //get the size of the file
    file_size = file.GetLength();

    file.Read((void*)&bmFileHeaderMain,sizeof(bmFileHeaderMain));
     //code for a bitmap structure employing Huffman coding is 10
    if(bmFileHeaderMain.bfReserved1 == 10) {return (0); }
    else{bmFileHeaderMain.bfReserved1 = 10; }
    fileD.Write((void*)&bmFileHeaderMain,sizeof(bmFileHeaderMain));
    file.Read((void*)&bmFileInfo,sizeof(bmFileInfo));

    //check if the bitmap color content is greter than 256 colors

    if (bmFileInfo.bmiHeader.biBitCount >16)
    {
        file.Seek(-1*(long)sizeof(bmFileInfo), CFile::current);
        file.Read((void*)&bmFileInfo,sizeof(BITMAPINFO));
        fileD.Write((void*)&bmFileInfo,sizeof(BITMAPINFO));
        Color24 = TRUE;
    }
    else
    {
        fileD.Write((void*)&bmFileInfo,sizeof(bmFileInfo));
        Color24 = FALSE;
    }

    //save headers for the Huffman compression coding

    fileD.Write((void*)&file_size,sizeof(file_size));
    fileD.Write(code,sizeof(code));
    fileD.Write(code_length,sizeof(code_length));

    BYTE *p = (BYTE *) GlobalAllocPtr(GMEM_MOVEABLE,file_size);
    BYTE *pnew;
    pnew =p;
    file.SeekToBegin();
    file.Read((void*)p,file_size);
    file.Close();
    counter = 0;
    if (bmFileInfo.bmiHeader.biBitCount >16)
    {
        pnew =pnew+sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
        offset =sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
    }
    else
    {
        pnew =pnew+sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
        offset =sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
    }

    for (loop = 0; loop < file_size-offset; loop++)
    {
         dvalue         = (unsigned short) *pnew;
```

```
            current_code   = code[dvalue];
            current_length = (unsigned short) code_length[dvalue];

            for (loop1 = current_length-1; loop1 >= 0; --loop1)
            {
                if ((current_code >> loop1) & 1)
                    thebyte |= (char) (1 << curbit);

                if (--curbit < 0)
                {
                    Outbuff[counter] = (BYTE) thebyte;
                    if(counter >18000) {
                      fileD.Write(Outbuff,counter+1);
                      counter =0;
                    }
                    thebyte = 0;
                    curbyte++;
                    curbit = 7;
                    counter++;
                }
            }
            pnew++;
        }
        fileD.Write(Outbuff,counter);
        Outbuff[0] = (BYTE) thebyte;
        fileD.Write(Outbuff,1);
        compress_charcount = ++curbyte;
        GlobalFreePtr(p);
        fileD.Close();
    }
    return(1);
}


////////////////////////////////////////////////////////////////////
//This function displays the results of the compression sequence.
////////////////////////////////////////////////////////////////////
void  CDib::compression_report ()
{
    if (Color24 = TRUE)
        header_charcount   = sizeof(BITMAPFILEHEADER) +sizeof(BITMAPINFO)+
                           768 + sizeof (file_size);
    else
        header_charcount   = sizeof(BITMAPFILEHEADER) +sizeof(BITMAPINFO)+
                           768 + sizeof (file_size);

    output_characters  = (unsigned long) header_charcount +
             compress_charcount;
    savings = 100 - ((float) output_characters / (float) file_size) * 100;
}


////////////////////////////////////////////////////////
//This function generates the compression code table.
////////////////////////////////////////////////////////

unsigned short  CDib::generate_code_table ()
{
    register unsigned short  loop;
    register unsigned short  current_length;
    register unsigned short  current_bit;
```

```
    unsigned short  bitcode;
    short           parent;


    for (loop = 0; loop < 256; loop++)
       if (frequency_count[loop])
       {
          current_length = bitcode = 0;
          current_bit = 1;
          parent = father[loop];

          while (parent)
          {
             if (parent < 0)
             {
                bitcode += current_bit;
                parent = -parent;
             }
             parent = father[parent];
             current_bit <<= 1;
             current_length++;
          }

          code[loop] = bitcode;

          if (current_length > 16)
             return (0);
          else
             code_length[loop] = (unsigned char) current_length;
       }
       else
          code[loop] = code_length[loop] = 0;

    return (1);
}

////////////////////////////////////////////////////
// This function builds the compression code tree.
////////////////////////////////////////////////////

void CDib::build_code_tree ()
{
    register unsigned short  findex;
    register unsigned long   heap_value;


    while (heap_length != 1)
    {
       heap_value = heap[1];
       heap[1]    = heap[heap_length--];

       reheap (1);
       findex = heap_length + 255;

       frequency_count[findex] = frequency_count[heap[1]] +
                                 frequency_count[heap_value];
       father[heap_value] =  findex;
       father[heap[1]]    = -findex;
       heap[1]            =  findex;
```

```
      reheap (1);
   }

   father[256] = 0;
}

////////////////////////////////////////////////////////////////////////////
//This function creates a "legal" heap from the current heap tree structure.
////////////////////////////////////////////////////////////////////////////

void CDib::reheap (unsigned short heap_entry)

{
   register unsigned short  index;
   register unsigned short  flag = 1;

   unsigned long   heap_value;


   heap_value = heap[heap_entry];

   while ((heap_entry <= (heap_length >> 1)) && (flag))
   {
      index = heap_entry << 1;

      if (index < heap_length)
         if (frequency_count[heap[index]] >= frequency_count[heap[index+1]])
            index++;

      if (frequency_count[heap_value] < frequency_count[heap[index]])
     flag--;
      else
      {
         heap[heap_entry] = heap[index];
         heap_entry       = index;
      }
   }

   heap[heap_entry] = heap_value;
}

////////////////////////////////////////////////////////////////////////////
// This function builds a heap from the initial frequency count data.
////////////////////////////////////////////////////////////////////////////

void CDib::build_initial_heap ()
{
   register unsigned short  loop;


   heap_length = 0;

   for (loop = 0; loop < 256; loop++)
      if (frequency_count[loop])
         heap[++heap_length] = (unsigned long) loop;

   for (loop = heap_length; loop > 0; loop--)
      reheap (loop);
}
```

```
////////////////////////////////////////////////////////////////////////
//This function counts the number of occurrences of each byte in the data
//that are to be compressed.
////////////////////////////////////////////////////////////////////////

void CDib::get_frequency_count (const char * fileName)
{
    register unsigned long loop;
    for(loop=0;loop<512;loop++){
       father[loop]=0;
       decomp_tree[loop]=0;
       frequency_count[loop]=0;
    };

    for(loop=0;loop<256;loop++){
       code[loop]=0;
       code_length[loop]=0;
    };

    for(loop=0;loop<257;loop++){
        heap[loop]=0;
    }

    //Definitions for the bitmap file Header
    BITMAPFILEHEADER  FileHeader;
    BITMAPINFO        FileInfo;
    long              offset;

    //Construct and open a file object.
    CFile file(fileName, CFile::modeRead);
    DWORD Size = file.GetLength();
    BYTE *p =(BYTE*)GlobalAllocPtr(GMEM_MOVEABLE,Size);
    file.Read((void*)p,Size);

    file.SeekToBegin();
    file.Read((void*)&FileHeader,sizeof(FileHeader));
    file.Read((void*)&FileInfo,sizeof(FileInfo));
    file.Close();

    BYTE *pnew;
    pnew =p;
    if (FileInfo.bmiHeader.biBitCount > 16)
    {
        pnew =pnew+sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
        offset =sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
    }
    else
    {
        pnew =pnew+sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
        offset=sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO);
    }


    for (loop = 0; loop < Size-(offset); loop++){
      frequency_count[*pnew]++;
      pnew++;
    }
    GlobalFreePtr(p);
}

////////////////////////////////////////////////////////////////////////
```

```
//This function builds the decompression tree.
////////////////////////////////////////////////////////////////////////

void  CDib::build_decomp_tree ()
{
   register unsigned short  loop1;
   register unsigned short  current_index;

   unsigned short  loop;
   unsigned short  current_node = 1;


   decomp_tree[1] = 1;

   for (loop = 0; loop < 256; loop++)
   {
      if (code_length[loop])
      {
     current_index = 1;
     for (loop1 = code_length[loop] - 1; loop1 > 0; loop1--)
     {
        current_index = (decomp_tree[current_index] << 1) +
               ((code[loop] >> loop1) & 1);
        if (!(decomp_tree[current_index]))
           decomp_tree[current_index] = ++current_node;
     }
     decomp_tree[(decomp_tree[current_index] << 1) +
       (code[loop] & 1)] = -loop;
      }
   }
}

////////////////////////////////////////////////////////////////////////////////
//This function decompresses the compressed image.
////////////////////////////////////////////////////////////////////////////////

void  CDib::decompress_image (const char* fileName, BYTE* pDib)
{
   register unsigned short  cindex = 1;
   register char            curchar;
   register short           bitshift;
   BYTE*                    pLocal;
   BYTE*                    pFileLocal;
   BITMAPINFO               bmFileInfo;
   unsigned long            Size;
   register unsigned long   counter;
   register unsigned long   loop;
   unsigned long            offset1;

   for(loop=0;loop<256;loop++){
       code[loop]=0;
       code_length[loop]=0;
    };
   for(loop=0;loop<511;loop++){
       decomp_tree[loop]=0;
   }

   // Save the pointer
   pLocal = pDib;

   // Construct and open a file object for decompression.
```

```
CFile file(fileName, CFile::modeRead);

file.Read((void*)&bmFileHeaderMain, sizeof(bmFileHeaderMain));

file.Read((void*)&bmFileInfo, sizeof(BITMAPINFO));

if (bmFileInfo.bmiHeader.biBitCount >16)
{
     file.Seek(-1*(long)sizeof(bmFileInfo), CFile::current);
     file.Read((void*)&bmFileInfo, sizeof(BITMAPINFO));
}

file.Read((void*)&file_size, sizeof(file_size));
file.Read((void*)code, sizeof(code));
file.Read((void*)code_length, sizeof(code_length));

//build the decompression tree
build_decomp_tree();

file.SeekToBegin();
Size = file.GetLength();
if (bmFileInfo.bmiHeader.biBitCount >16)
{
    compress_charcount = Size-(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO)
                              +sizeof(file_size) + 768);
}
else
{
    compress_charcount = Size-(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO)
                              +sizeof(file_size) + 768);
}

BYTE* pFile = (BYTE*)GlobalAllocPtr(GMEM_MOVEABLE,Size);
file.Read((void*)pFile,Size);
file.Close();

pFileLocal= pFile;
pFileLocal= pFileLocal + sizeof(BITMAPFILEHEADER);

if (bmFileInfo.bmiHeader.biBitCount >16)
{
    offset1 = sizeof(BITMAPINFO);
}
else
{
    offset1 = sizeof(BITMAPINFO);
}

for(counter=0;counter < offset1;counter ++)
{
    *pLocal =*pFileLocal;
    pLocal++;pFileLocal++;
}


pFileLocal = pFileLocal+ sizeof(file_size)+ sizeof(code) + sizeof(code_length)
;


unsigned long  charcount = 0;
```

```
if (bmFileInfo.bmiHeader.biBitCount >16)
{
    file_size=file_size-(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO))-1;
}
else
{
    file_size=file_size-(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO))-1;
}

while (charcount < file_size)
{
   curchar = (char) *pFileLocal;

   for (bitshift = 7; bitshift >= 0; --bitshift)
   {
       cindex = (cindex << 1) + ((curchar >> bitshift) & 1);

      if (decomp_tree[cindex] <= 0)
      {
        *pLocal = (int) (-decomp_tree[cindex]);
        if ((++charcount) == file_size)
            bitshift = 0;
        else
            cindex = 1;
        pLocal++;
      }
      else
       cindex = decomp_tree[cindex];
   }
   pFileLocal++;
}

if (bmFileInfo.bmiHeader.biBitCount >16)
{
    file_size=file_size+(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO))-1;
}
else
{
    file_size=file_size+(sizeof(BITMAPFILEHEADER) + sizeof(BITMAPINFO))-1;
}


  GlobalFreePtr(pFile);

}

////////////////////////////////////////////////////////////
// This function returns pointer the BITMAPFILEHEADER.
////////////////////////////////////////////////////////////

BITMAPFILEHEADER CDib::Get_BitMap_FileHeader()
{
    return bmFileHeaderMain;
}
```

REFERENCES

1. Robert G. Galler, *Information Theory and Reliable Communication*, John Wiley &Sons, 1968.

2. Steven Roman, *Introduction to Coding and Information Theory*, Springer, 1997.

3. Richard W. Hamming, *Coding and Information Theory*, Printice-Hall, Inc., Englewood Cliffs, N. J. 07632, 1980.

4. Gilbert Held, *Data and Image Compression Tools and Techniques*, John Wiley &Sons, 1996.

5. Ulrich Gunthere and Radu Nicolescu, *A Low Cost Decoder for Arbitrary Binary Variable Length Codes*, Computer Science Department of the University of Auckland CITR at Tamaki Campus, September 1997.

6. Robert L. Hummel, *Data and Fax Communications*, Ziff-Davis ZD Press.

7. Charles Mirho and Andre Terrisse, *Communications Programming for Windows 95*, Microsoft Press, 1996.

DECLARATION

I, the undersigned, hereby declare that this thesis is my original work carried out under the supervision of Dr.-Ing Hailu Ayele, has not been presented as a thesis for a degree program in any other university and that all sources used for the thesis are duly acknowledged.

Yohannes Kassahun