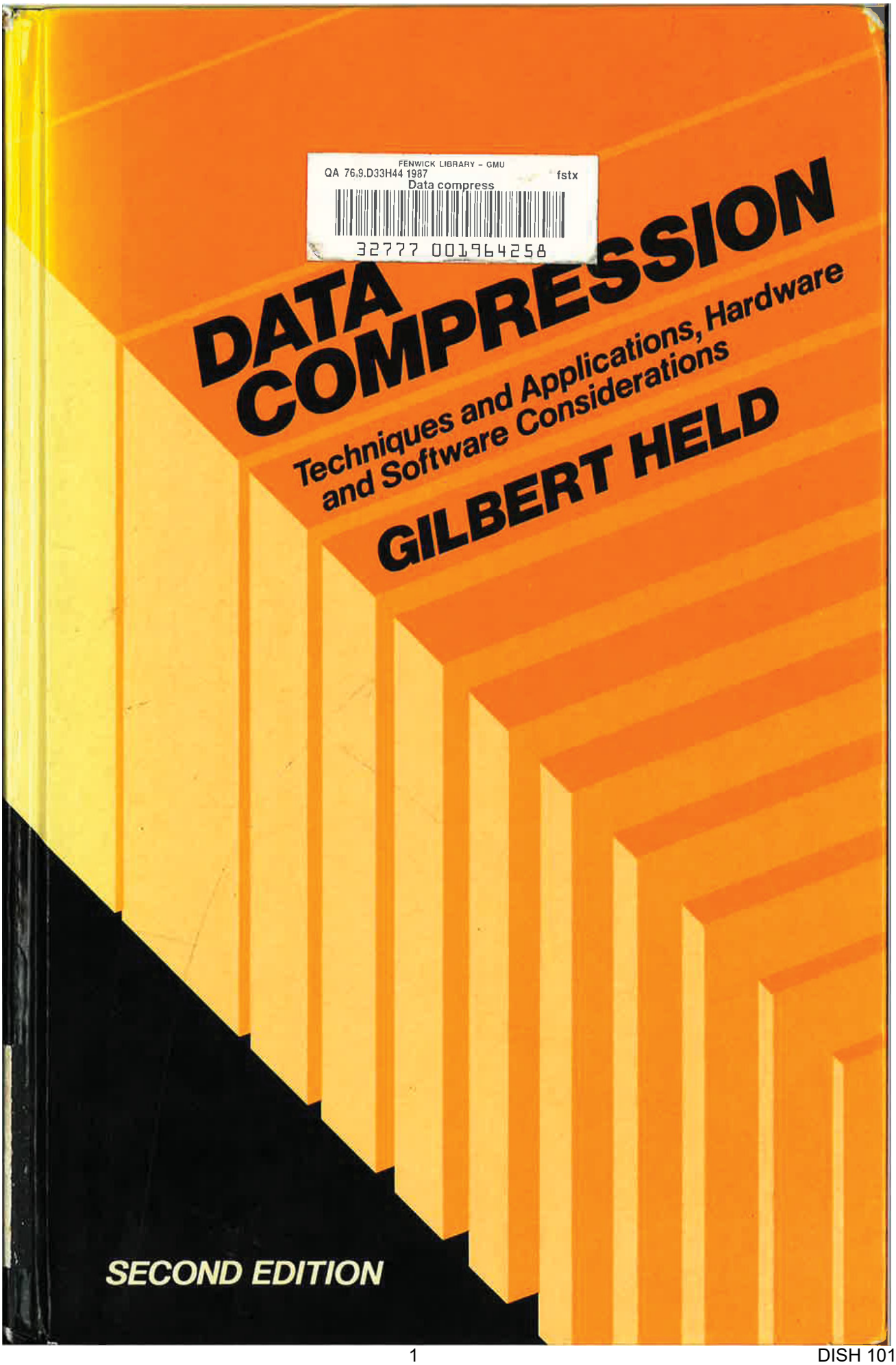

FENWICK LIBRARY - GMU
QA 76.9.D33H44 1987
Data compress
fstx
32777 001964258
DATA COMPRESSION
Techniques and Applications, Hardware and Software Considerations
GILBERT HELD
SECOND EDITION

DATA COMPRESSION

Techniques and Applications
Hardware and Software Considerations

Second Edition

GILBERT HELD
4-Degree Consulting,
Macon, Georgia, USA

and

THOMAS R. MARSHALL
(software author)

John Wiley & Sons
Chichester · New York · Brisbane · Toronto · Singapore

Library of Congress Cataloging-in-Publication Data:
Held, Gilbert, 1943–
Data compression.
Bibliography: p.
Includes index.
1. Data compression (Computer science) I. Marshall, Thomas (Thomas R.) II. Title.
QA76.9.D33H44 1987 005.74'6 86-18942
ISBN 0 471 91280 8

British Library Cataloguing in Publication Data:
Held, Gilbert
Data compression: techniques and applications hardware and software considerations. —
2nd ed.
1. Data compression (Computer science)
I. Title II. Marshall, Thomas
005.74'6 QA76.9.D33
ISBN 0 471 91280 8

Typeset by Photo·Graphics, Honiton, Devon
Printed in Great Britain

CHAPTER ONE

Rationale and Utilization

In the chronology of computer development, large-scale information transfer by remote computing and the development of massive information storage and retrieval systems have witnessed a tremendous growth. Concurrent with this growth, several problem areas have developed which can result in major, but unnecessary, economic expenditures.

One problem is the so-called 'run-away database'. Here the size of the database used by an organization for its information storage and retrieval programs becomes larger and larger, requiring additional disc drives for online systems and reels of magnetic tapes for those systems that can be processed in a batch environment.

Accompanying the growth in the size of databases has been a large increase in the number of users and duration of usage by personnel at remote locations. These factors result in tremendous amounts of data being transferred between computers and remote terminals. To provide transmission facilities for the required data transfers, communications lines and auxiliary devices, such as modems and multiplexers, have been continuously upgraded by many organizations to permit higher data-transfer capability.

Although the obvious solutions to these problems of data storage and information transfer are to install additional storage devices and expand existing communications facilities, to do so requires an additional increase in an organization's equipment and operating costs. One method that can be employed to alleviate a portion of data storage and information transfer problems is through the representation of data by more efficient codes. If one examines an organization's database or monitors a transmission line, there is an excellent chance that the individual characters that both make up the database and the transmission sequence could be encoded more effectively. Two techniques that can result in a more effective encoded data representation are logical and physical data compression.

1.1 LOGICAL COMPRESSION

When a database is designed, one of the first steps of the analyst is to obtain as much data reduction as possible. This data reduction results from the

elimination of redundant fields of information while representing the data elements in the remaining fields with as few logical indicators as is feasible.

Although logical compression is data dependent and the method employed can vary based upon the analyst's foresight, the following two examples will illustrate the ease of implementation and benefits of this compression technique.

One simple example of logical data compression is the occupational field on a personnel database. Suppose 30 alphanumeric positions are allocated to this field. If the field is fixed, occupations such as the 10-character occupational description 'DISHWASHER' have 20 blanks inserted into the remainder of the field. Then, 30 million characters of storage would be required for the occupational field of 1 million workers. Suppose at most there were 32 768 distinct occupations. Instead of indicating the occupational title, one could encode the equivalent 5-digit data code, eliminating 25 character positions per field. The size of the field could be reduced further by allocating the binary value of 1 or more characters to the occupational code. As an example, an 8-bit character could represent $2^8 - 1$ or 255 distinct values or occupational codes. Linking two 8-bit characters through appropriate software would provide $2^{16}-1$ or 65 535 distinct codes. This would reduce the field size from 30 to 2 characters, saving 28 million characters of storage. If our counting begins at zero instead of a conventional starting place of 1, an 8-bit character could represent 256 codes while a 16-bit character could be employed to represent 65 536 distinct values.

A second example of logical compression is a date field. This type of field frequently occurs in databases. Normally, the numeric equivalents of the subfields representing day, month and year are used in place of longhand notation. Thus, 01 04 81 would represent 1 April 1981. While this logical compression results in 6 numeric characters of storage, additional data reduction can result from storing the date as a binary value. Since the day will never exceed 31, 5 bits would suffice to represent the date field. Similarly, 4 bits could be used to represent the month value while 7 bits could represent 127 years, permitting a relative year ranging from 1900 to 2027.

Logical compression using numerical and binary representation is illustrated in Figure 1.1 for the preceding date field example. It is interesting to note that employing binary representation reduces the date field to 16 binary digits or two 8-bit concatenated characters of storage. As discussed, many logical compression methods can be considered by an analyst during the database design process. Each method may result in a distinct degree of data storage reduction. Correspondingly, when logically compressed databases or portions of such databases are transmitted between locations, transmission time is reduced since fewer data characters are transmitted.

While logical compression can be an effective tool in minimizing the size of a database, it only reduces transmission time when logically compressed data is transmitted. Thus, the transmission of inquiry and response data

Longhand	DAY	MONTH	YEAR
Example	1	APRIL	1981
Logical compression using numerical representation			
Example	01	04	81
Logical compression using binary representation			
Example	00001	0100	1010001

Figure 1.1 Logical compression methods. Logical compression can result from alphanumeric, numeric or binary representation of data in a shorthand notation

which are typically encoded as separate and distinct entities in the appropriate bit representation of the code for each character is not normally affected. Similarly, the occurrence of repeating patterns and groups of characters which are normally contained in reports transmitted from computer systems to terminal devices would not be affected. For such situations, a reduction in data transmission time depends upon the physical compression of the data as it is encountered.

1.2 PHYSICAL COMPRESSION

Physical compression can be viewed as the process of reducing the quantity of data prior to it entering a transmission medium and the expansion of such data into its original format upon receipt at a distant location. Although both physical and logical compression can result in reduced transmission time, distinct application differences exist between the two techniques. Logical compression is normally used to represent databases more efficiently and does not consider the frequency of occurrence of characters or groups of characters. Physical compression takes advantage of the fact that when data is encoded as separate and distinct entities, the probabilities of occurrence of characters and groups of characters differ. Since frequently occurring characters are encoded into as many bits as those characters that only rarely occur, data reduction becomes possible by encoding frequently occurring characters into short bit codes while representing infrequently occurring characters by longer bit codes. Like logical compression, many physical compression techniques exist. Some techniques replace repeating strings of characters by a special compression indicator character and a quantity count character. Other techniques replace frequently occurring characters with a short binary code while infrequently encountered characters are replaced by longer binary codes. In Chapter 2, 10 distinct physical compression methods are covered in detail. For the remainder of this book we will focus our attention upon physical data compression.

1.3 COMPRESSION BENEFITS

When data compression is used to reduce storage requirements, overall program execution time may be reduced. This is because the reduction in storage will result in a reduction of disc-access attempts, while the encoding and decoding required by the compression technique employed will result in additional program instructions being executed. Since the execution time of a group of program instructions is normally significantly less than the time required to access and transfer data to a peripheral device, overall program execution time may be reduced.

With respect to the transmission of data, compression provides the network planner with several benefits in addition to the potential cost savings associated with sending less data over the switched telephone network where the cost of the call is usually based upon its duration. First, compression can reduce the probability of transmission errors occurring since fewer characters are transmitted when data is compressed while the probability of an error occurring remains constant. Second, since compression increases efficiency, it may reduce or even eliminate extra workshifts. Finally, by converting text that is represented by a conventional code such as standard ASCII into a different code, compression algorithms may provide a level of security against illicit monitoring.

For data communications, the transfer of compressed data over a medium results in an increase in the effective rate of information transfer, even though the actual data transfer rate expressed in bits per second remains the same. Data compression can be implemented on most existing hardware by software or through the use of a special hardware device that incorporates one or more compression techniques.

In Figure 1.2, a basic data-compression block diagram is illustrated. Shown as a black box, compression and decompression may occur within the user's processor to include personal computers, intelligent terminals or in a device foreign to the processor, such as a specialized communications component. Foremost among these components are data concentrators and statistical multiplexers.

To examine in some detail a portion of the benefits that may result from the employment of one or more compression techniques requires a review of some fundamental compression terminology.

1.4 TERMINOLOGY

As illustrated in Figure 1.2, an original data stream is operated upon according to a particular algorithm to produce a compressed data stream. This compression of the original data stream is sometimes referred to as an encoding process with the result that the compressed data stream is also called an encoded data stream. Reversing the process, the compressed data stream is decompressed to reproduce the original data stream. Since this

Figure 1.2 Basic data-compression block diagram. An original data stream operated upon according to one or more compression algorithms results in the generation of a compressed data stream

decompression process results in the decoding of the compressed data stream, the result is sometimes referred to as the decoded data stream. We will use the term original data stream and decoded data stream synonymously, as well as the terms compressed data stream and encoded data stream.

The degree of data reduction obtained as a result of the compression process is known as the compression ratio. This ratio measures the quantity of compressed data in comparison to the quantity of original data, such that (Ruth and Krentzler, 1972):

$$\text{Compression ratio} = \frac{\text{Length of original data string}}{\text{Length of compressed data string}}$$

From the above equation, it is obvious that the higher the compression ratio the more effective the compression technique employed. Another term used when talking about compression is the figure of merit, where:

$$\text{Figure of merit} = \frac{\text{Length of compressed data string}}{\text{Length of original data string}}$$

The figure of merit is the reciprocal of the compression ratio and must always be less than unity for the compression process to be effective. The fraction of data reduction is one minus the figure of merit. 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 the fraction of data reduction would be 0.66.

1.5 COMMUNICATIONS APPLICATIONS

To obtain an overview of some of the communications benefits available through the incorporation of data compression, we can consider a typical data communications application. As illustrated in the top portion of Figure 1.3, a remote batch terminal is connected to a central computer with trans-

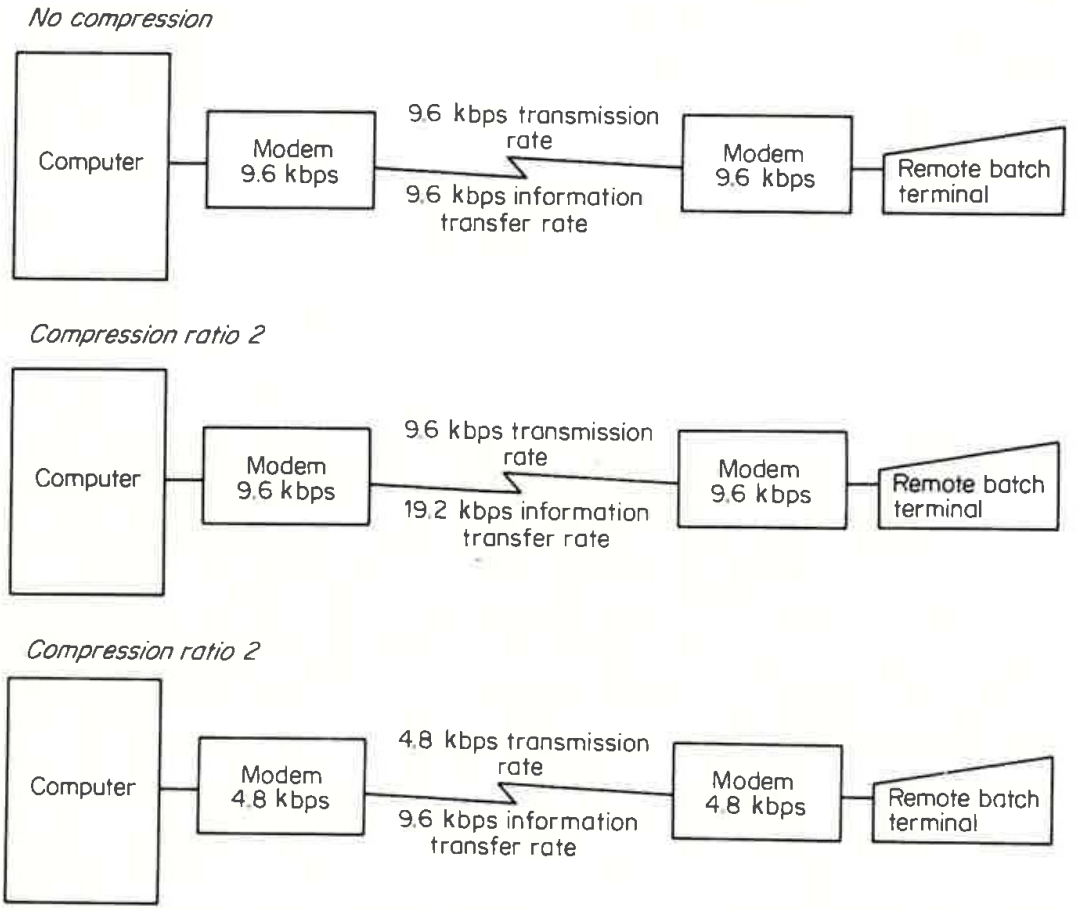


Figure 1.3 Data compression affects the information transfer ratio (ITR), through the use of data compression, the methodology and structure of one's data communications facility may be changed

mission occurring at a 9.6 kbps data rate. Let us assume that the data to be transmitted has not been compressed. If through the programming of one or more compression algorithms or the installation of a hardware compression device a compression ratio of 2 is obtained, several alternatives may be available with respect to one's data communications methodology. First, our data transmission time is reduced since the effective information transfer rate has increased to approximately 19.2 kbps as shown in the middle portion of Figure 1.3. Ignoring communications software overhead, the data transmission time is halved. Thus, one may now consider using the remote batch terminal for other remote processing applications or perhaps an expensive after-hours shift or portion of such a shift can be alleviated. In the lower portion of Figure 1.3 another user option is illustrated. Here the transmission rate may be reduced to 4800 bps. With a compression ratio of 2, this is equivalent to an information transfer rate of 9600 bps. By lowering the data transmission rate, more expensive 9600 bps modems may be replaced by 4800 bps modems and line conditioning which is normally required when transmitting data at 9600 bps may be removed, resulting in an additional cost reduction.

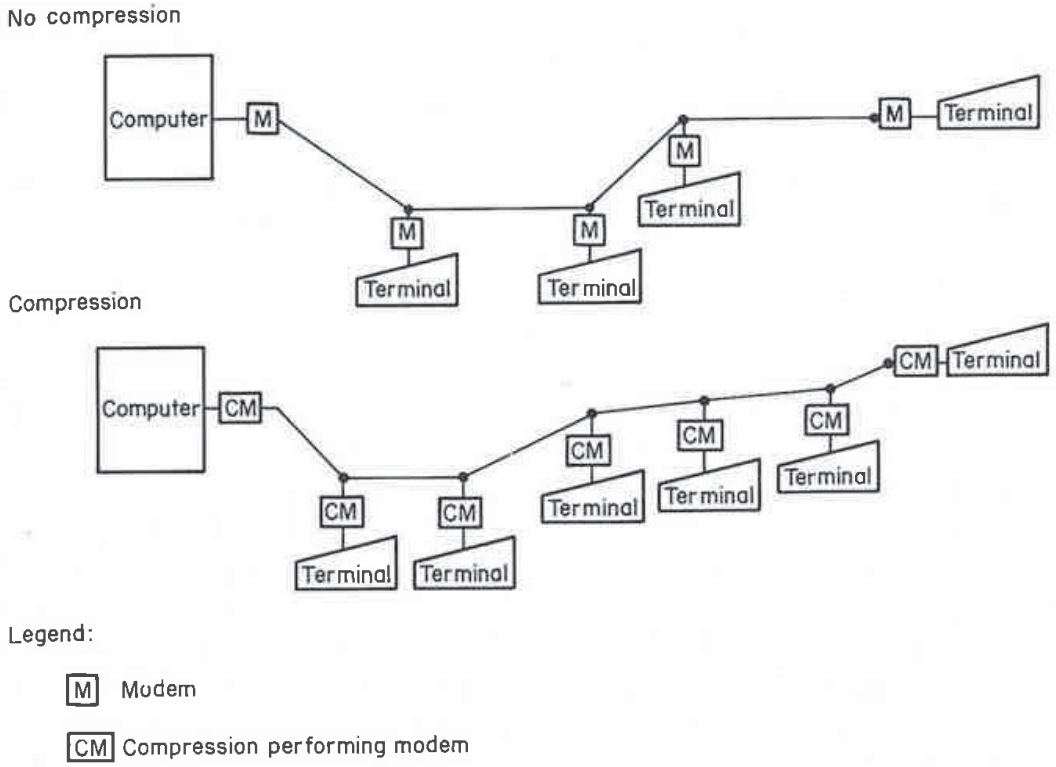


Figure 1.4 Data compression on a multidrop line reduces the flow of data on the line, permitting additional terminals to be serviced

A second type of communications application that can benefit from the utilization of data compression is illustrated in Figure 1.4. A typical multidrop network is illustrated in the top portion of Figure 1.4, connecting terminals at diverse geographical locations via a common leased line to a computer site. Typically, the transmission activity of the terminals is the governing factor that limits the multidrop line to a maximum number of drops. In the bottom portion of Figure 1.4, it is assumed that compression performing modems were substituted for the conventional modems used in the original multidrop configuration. Since data compression on a multidrop line reduces the flow of data on the line, its utilization will normally enable additional drops to be added to the line prior to the occurrence of throughput delays that affect the response time of the terminals attached to each drop. In this particular example, it is assumed that the use of compression performing modems permitted an increase in the number of line drops from 4 to 6.

1.6 DATA COMPRESSION AND INFORMATION TRANSFER

When data is 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 transmitted over a transmission medium must be converted into an acceptable format for that medium. When digital data is transmitted over analogue telephone lines, modems must be

employed to convert the digital pulses of the business machine into a modulated signal acceptable for transmission on the analogue telephone circuit. The time between the first bit entering the modem and the first modulated signal produced by the device is known as 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 is known as the circuit or propagation delay time. This is the time required for the signal to be propagated or transferred down the line to the distant end. Propagation delay time can be approximated by equating 1 millisecond for every 150 circuit miles and adding 12 milliseconds to the total.

Once data is received at the distant end it must be acted upon, resulting in a processing 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 time 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 at the associated modem to computer and modem to terminal interface change. The time required to switch control signals to change the direction of transmission is known as line turnaround time and can result in delays up to 250 or more milliseconds, depending upon the transmission protocol employed. We can denote the effect of data compression by examining the transmission protocol commonly known as BISYNC communications and a few of its derivations.

BISNYC communications

One of the most commonly employed transmission protocols is the Binary Synchronous Communications (BISNYC) communications control structure. This line control structure was introduced in 1966 by International Business Machine Corporation and is used for transmission by many medium-speed and high-speed devices to include terminal and computer systems. BISNYC provides a set of rules which govern the synchronous transmission of binary-coded data. While this protocol can be used with a variety of transmission codes, it is limited to the half duplex transmission mode and requires the acknowledgement of the receipt of every block of transmitted data. In an evolutionary process, a number of synchronous protocols have been developed to supplement or serve as a replacement to BISNYC, the most prominent being the high level data link control (HDLC) protocol defined by the International Standard Organization (ISO).

The key difference between BISYNC and HDLC protocols is that BISYNC is a half duplex, character-oriented transmission control structure while

HDLC is a bit-oriented, full duplex transmission control structure. We can investigate the efficiency of these basic transmission control structures and the effect of data compression upon their information transfer efficiency. To do so, an examination of some typical error control procedures is first required.

Error control

The most commonly employed error-control procedure is known as automatic request for repeat (ARQ). In this type of control procedure, upon detection of an error a request is made by the receiving station to the sending station to retransmit the message. Two types of ARQ procedures have been developed: 'stop and wait ARQ' and 'go back *n* ARQ', which is sometimes called continuous ARQ.

'Stop and wait ARQ' is a simple type of error-control procedure. Here the transmitting station stops at the end of each block and waits for a reply from the receiving terminal pertaining to the block's accuracy (ACK) or error (NAK) prior to transmitting the next block. This type of error-control procedure is illustrated in Figure 1.5. Here the time between transmitted blocks is referred to as dead time which acts to reduce the effective data rate on the circuit. When the transmission mode is half duplex, the circuit must be turned around twice for each block transmitted, once to receive the reply (ACK or NAK) and once again to resume transmitting. These line turnarounds, as well as such factors as the propagation delay time, station message processing time and the modem internal delay time, all contribute to what is shown as the cumulative delay factors.

When the 'go back *n* ARQ' type of error control procedure is employed, the dead time can be substantially reduced to the point where it may be

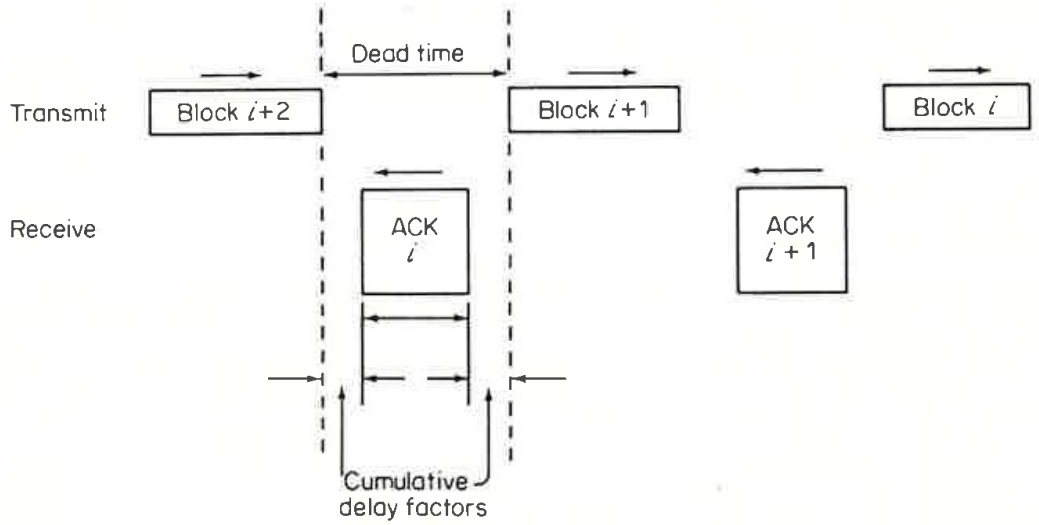


Figure 1.5 Stop and wait ARQ. In this type of error control procedure, the receiver transmits an acknowledgement after each block. This can result in a significant amount of cumulative delay time between data blocks

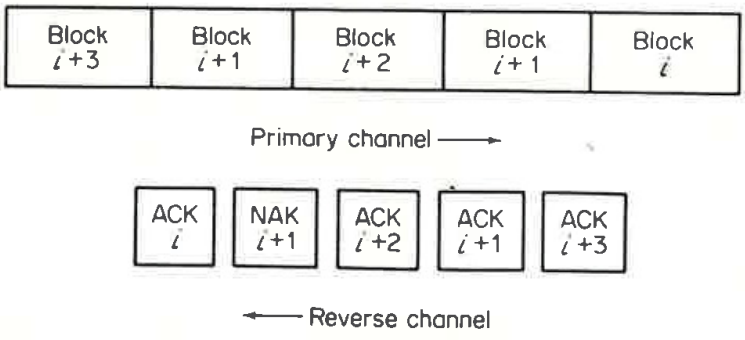


Figure 1.6 Go back N ARQ. In a 'go back n ARQ' error-control procedure, the transmitter continuously sends messages until the receiver detects an error. The receiver then transmits a negative acknowledgement on the reverse channel and the transmitter retransmits the block received in error. Some versions of this technique require blocks sent before the error indication was encountered to be retransmitted in addition to the block received in error

insignificant. One way to implement this type of error control procedure is by the utilization of a simultaneous reverse channel for acknowledgement signalling as illustrated in Figure 1.6. In this type of operating mode, the receiving station sends back the ACK or NAK response on the reverse channel for each transmitted block. If the primary channel operates at a much higher data rate than the reverse channel, many blocks may have been received prior to the transmitting station receiving the NAK in response to a block at the receiving station being found in error. The number of blocks one may go back to request a transmission, n, is a function of the block size and buffer area available in the business machines and terminals at the transmitting and receiving stations, the ratio of the data transfer rates of the primary and reverse channels and the processing time required to compute the block check character and transmit an acknowledgement. For the latter, this time is shown as small gaps between the ACK and NAK blocks in Figure 1.6.

Half duplex throughout model

When a message block is transmitted in the BISYNC control structure, a number of control characters are contained in that block in addition to the message text. If the variable C is assigned to represent the number of control characters per block and the variable D is used to represent the number of data characters, then the total block length is $C + D$. If the data transfer rate expressed in bps is denoted as T_R and the number of bits per character is denoted as B_C, then the transmission time for one character is equal to B_C/T_R which can be denoted as T_C. Since $D + C$ characters are contained in a message block, the time required to transmit the block will become $T_C{}^*(D + C)$. Once the block is received, it must be acknowledged. To do

so, the receiving station is required to first compute a block check character (BCC) and compare it with the transmitted BCC character appended to the end of the transmitted block. Although the BCC character is computed as the data is received, a comparison is performed after the entire block is received and only then can an acknowledgement be transmitted. The time to check the transmitted and computed BCC characters and form and transmit the acknowledgement is known as the processing and acknowledgement time (T_{PA}).

When transmission is half duplex, the line turnaround time (T_L) required to reverse the transmission direction of the line must be added. Normally, this time includes the request-to-send/clear-to-send (RTS/CTS) modem delay time as well as each of the modems' internal delay time. For the acknowledgement to reach its destination, it must propagate down the circuit and this propagation delay time, denoted as T_p, must also be considered. If the acknowledgement message contains A characters then, when transmitted on the primary channel, A^*B_C/T_R seconds are required to send the acknowledgement.

Once the original transmitting station receives the acknowledgement it must determine if it is required to retransmit the previously sent message block. This time is similar to the processing and acknowledgement time previously discussed. To transmit either a new message block or repeat the previously sent mesasge block, the line must be turned around again and the message block will require time to propagate down the line to the receiving station. Thus, the total time to transmit a message block and receive an acknowledgement, denoted as T_B, becomes:

$$T_B = T_C{}^*(D + C) + 2^* (T_{PA} + T_L + T_p) + (A^*B_C/T_R). \tag{1.1}$$

Since efficiency is the data-transfer rate divided by the theoretical data-transfer rate, the transmission control structure efficiency (E_{TCS}) becomes:

$$E_{TCS} = \frac{B_C{}^*D^*(1-P)}{T_R{}^*T_B}. \tag{1.2}$$

Here P is the probability that one or more bits in the block are in error, causing a retransmission to occur.

Although the preceding is a measurement of the transmission control structure efficiency, it does not consider the data code efficiency which is the ratio of information bits to total bits per character. When the data code efficiency is included, we obtain a measurement of the information transfer efficiency. We can call this ratio the information transfer ratio (ITR) which will provide us with a measurement of the protocol's information transfer efficiency. This results in:

$$\text{ITR} = \frac{B_{IC}{}^*E_{TCS}}{B_C} \tag{1.3}$$

where:

ITR = Information transfer ratio
B_{IC} = Information bits per character
B_C = Total bits per character
D = Data characters per message block
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 turnaround time
T_P = Propagation delay time
p = Probability of one or more errors in block.

From the preceding, the information transfer ratio provides us with a measurement of the efficiency of the transmission control structure without considering the effect of compression. When compression is considered we obtain a new term which we will denote as the effective information transfer ratio (EITR).

When data is compressed, the original data stream will be reduced in size prior to transmission, the actual reduction being dependent upon the compression algorithms employed as well as the composition of the data acted upon. In general, we can assume that the compression ratio considers the number of characters in the compressed data stream to include special control characters required to indicate one or more compression algorithms. This reasonable assumption simplifies the effect of considering data compression when examining a particular protocol. As an example, consider a 160-character data block compressed into 78 data characters plus 2 compression indicator characters. Here the compression ratio would be 160/(78 + 2) or 2. The effect upon the previously developed equation to compute the information transfer ratio would be to change D in the numerator to the non-compressed string length of 160 characters while D in the denominator would be the actual 78 compressed data characters plus the two additional special characters required to indicate data compression, resulting in a total of 80 characters. If the total number of control characters framing the data block is relatively small, the effective information transfer ratio can be approximated by multiplying the information transfer ratio by the compression ratio.

Computation examples

We will assume that our data transmission rate is 4800 bps and we will transmit information using a BISYNC transmission control structure employing a 'stop and wait ARQ' error control procedure. Furthermore, let us assume the following parameters:

A = 4 characters per acknowledgement
B_{IC} = 8 bits per character
B_C = 8 bits per character
D = 80 data characters per block
C = 10 control characters per block
T_R = 4800 bps
T_C = 8/4800 = 0.00166 seconds (s) per character
T_{PA} = 20 milliseconds = 0.02 s
T_L = 100 milliseconds = 0.10 s
T_p = 30 milliseconds = 0.03 s
P = 0.01

Then:

$$\text{ITR} = \frac{8 \cdot 80 \cdot (1-0.01)}{4800*[0.00166(80+10)+2*(0.02+0.03+0.1)+4*8/4800)]} = 0.2861.$$

Since the transfer rate of information in bits (TRIB) is equal to the product of the data transfer rate and the information transfer ratio, we obtain:

$$\text{TRIB} = \text{ITR}*T_R = 0.2861*4800 = 1373 \text{ bps.}$$

For the preceding example, approximately 28 per cent of the data transfer rate is effectively used.

Let us now examine the effect of doubling the text size to 160 characters while the remaining parameters except P continue as before. Since the block size has doubled, P approximately doubles, resulting in the ITR becoming:

$$\text{ITR} = \frac{8*160*(1-0.02)}{4800*[0.00166(160+10)+2*(0.02+0.03+0.1)+(4*8/4800)]} = 0.4339.$$

With an ITR of 0.4339 the TRIB now becomes:

$$\text{TRIB} = \text{ITR}*T_R = 0.4339*4800 = 2083 \text{ bps.}$$

Here, doubling the block size raises the percentage of the data transfer rate effectively used to 43.39 per cent.

Compression effect

Suppose one or more data-compression algorithms are employed which result in a compression ratio of 2. What effect would this have upon the effective information transfer ratio?

The effective information transfer ratio (EITR) can be obtained by modifying equation (1.1) as follows:

$$\text{EITR} = \frac{B_{IC}{}^{*}D_1{}^{*}(1-P)}{T_R{}^{*}[T_C{}^{*}(D_2+C)+2{}^{*}(T_{PA}+T_L+T_p)+(A{}^{*}B_C/T_R)]} \qquad (1.4)$$

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.

If on the average 160 data characters are transmitted in a compressed format of 80 characters we obtain:

$$\text{EITR} = \frac{8{}^{*}160{}^{*}(1-0.01)}{4800{}^{*}[0.00166(80+10)+2{}^{*}(0.02+0.03+0.1)+(4{}^{*}8/4800)]} = 0.5788.$$

As previously discussed, the effective information transfer ratio can be approximated as follows:

$$\text{EITR} \simeq \text{ITR}{}^{*}\text{CR}. \qquad (1.5)$$

Substituting, we obtain:

$$\text{EITR} \simeq 0.2861{}^{*}2 \simeq 0.5722.$$

Since the transfer rate of information in bits (TRIB) is the product of the effective information transfer ratio and the operating data rate, we obtain:

$$\text{TRIB} = 0.5788{}^{*}4800 = 2778 \text{ bps}.$$

In Table 1.1, the reader will find a comparison of the variations in the ITR, EITR and TRIB when non-compressed and compressed data are transmitted for two different block sizes.

From Table 1.1, it is apparent that two methods can be employed to increase one's transmission efficiency. First, one may alter the protocol or transmission control sequence by varying the size of the data blocks transmitted. Alternatively, one can compress data prior to transmitting a block of information. Both methods can result in more information passing over a transmission line per unit time.

In Table 1.2, the reader will find a tabulation of the execution of a computer program which calculated the ITR as the block size varied from 40 to 2480 characters in increments of 40. In examining this table one should note that the maximum ITR of 0.6459 is obtained when the block size is 720 characters. This indicates that as the block size increases with a constant error rate, a certain point is reached where the time to retransmit a long block every so often negates the enlargement of the block size. For the

Table 1.1 Compression effect comparison

	Non-compressed data		Compressed data	
Block size (characters)	80	160	80	160
ITR (dimensionless)	0.2861	0.4339	N/A	N/A
EITR (dimensionless)	N/A	N/A	0.5788	0.8678
TRIB (bps)	1373	2083	2778	4165

Table 1.2 Information transfer ratio and block size.
Probability of block error = 0.01

ITR	Block size	ITR	Block size
0.169	40	0.590	1280
0.286	80	0.534	1320
0.370	120	0.577	1360
0.433	160	0.570	1400
0.482	200	0.564	1440
0.519	240	0.556	1480
0.549	280	0.549	1520
0.572	320	0.542	1560
0.591	360	0.535	1600
0.606	400		
0.617	440	0.527	1640
0.626	480	0.519	1680
0.633	520	0.512	1720
0.638	560	0.504	1760
0.642	600	0.495	1800
0.644	640	0.488	1840
0.645	680	0.480	1880
0.645	720	0.472	1920
		0.464	1960
0.645	760	0.455	2000
0.643	800	0.447	2040
0.641	840	0.439	2080
0.639	880	0.430	2120
0.635	920	0.422	2160
0.632	960	0.413	2200
0.628	1000	0.404	2240
0.623	1040	0.396	2280
0.618	1080	0.337	2320
0.613	1120	0.378	2360
0.608	1160	0.370	2400
0.602	1200	0.361	2440
0.596	1240	0.352	2480

parameters considered, the optimum block size is 720 characters. Only for the ideal situation, where $P = 0$ would a continuous increase in block size produce additional efficiencies.

In Figure 1.7 the ITR is plotted as a function of block size for the error-free condition and 0.01 probability of error conditions. The 0.01 probability of error condition per 40 character block was held constant by incrementing the error rate in proportion to the increase in the block size. Since an error-free line is not something a transmission engineer can reasonably expect, 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. Once again, data compression can become an effective mechanism for increasing transmission efficiency while keeping data buffer requirements within an acceptable level.

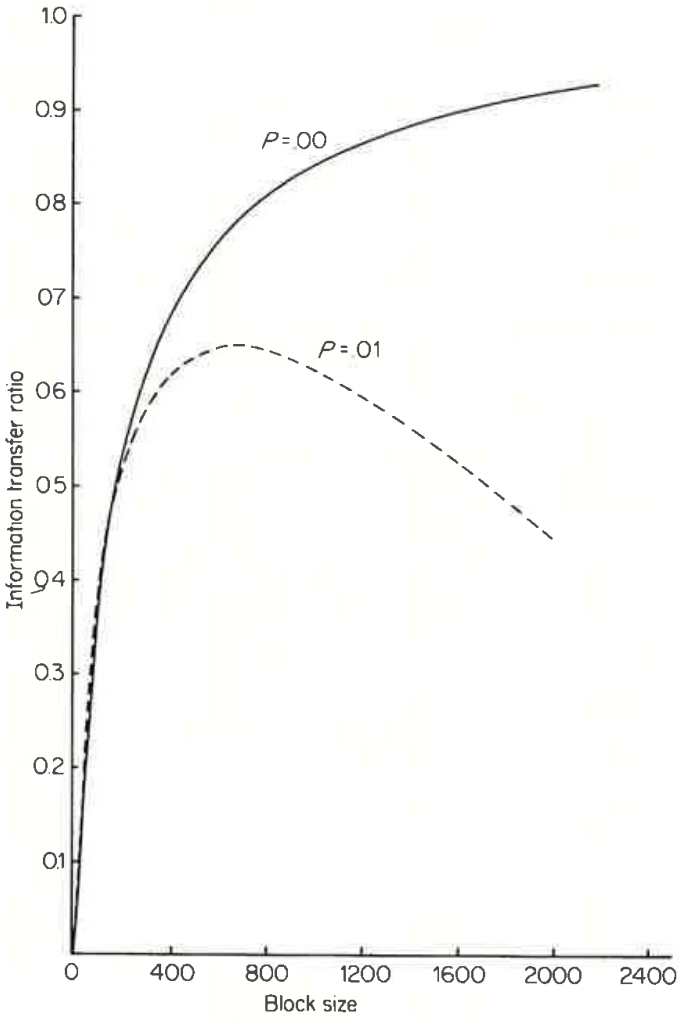


Figure 1.7 ITR and error rate

Return channel model

Consider a 'stop and wait ARQ' error control procedure where a return channel is available for the transmission of acknowledgements. The use of this return channel eliminates the necessity of line turnarounds; however, transmission is still half duplex since an acknowledgement is only transmitted after each received message block is processed.

When the message block is sent to the receiving station, both propagation delay and processing delay are encountered. When the acknowledgement is returned, one additional propagation delay and processing delay results. In addition to these delays, one must also consider the time required to transmit the acknowledgement message. If A denotes the length in characters of the acknowledgement message and T_S is the reverse channel data rate in bps, then the transmission time for the acknowledgement becomes $(A^*B_c)/T_S$. The total delay time due to the propagation and processing as well as the acknowledgement transmission time becomes:

$$2^*(T_{PA}+T_p)+\frac{A^*B_C}{T_S}.$$

Thus, the information transfer ratio becomes:

$$\text{ITR}=\frac{B_{IC}{}^*D_1(1-P)}{T_R{}^*[T_C{}^*(D_2+C)+2^*(T_{PA}+T_p)+A^*B_C/T_S]} \tag{1.6}$$

Let us examine the effect of this modified transmission procedure on the previous example where data was packed 80 characters per block. Let us assume that a 75 bps reverse channel is available and our acknowledgement message is comprised of four 8-bit characters. Then:

$$\text{ITR}=\frac{8^*80^*(1-0.01)}{4800[0.00166(80+10)+2^*(0.02+0.03)+4^*8/75]}=0.1953.$$

Note that the ITR actually decreased. This was caused by the slowness of the reverse channel where it took 0.4266 (4*8/75) seconds to transmit an acknowledgement. In comparison, the two-line turnarounds that were eliminated only required 0.2 s when the acknowledgement was sent at 4800 bps on the primary channel. This modified procedure is basically effectively when the line turnaround time exceeds the transmission time of the acknowledgement on the return channel. This situation normally occurs when the primary data transfer rate is 2400 bps or less. If the data is compressed prior to transmission and a compression ratio of 2 results in 160 data characters transmitted as a block of 80 compressed characters, the EITR can be computed as follows:

$$\text{EITR}=\frac{8^*160^*(1-0.01)}{4800[0.00166(80+10)+2^*(0.02+0.03)+4^*8/75]}=0.39.$$

In comparing the effect of compression, note that the transfer rate of information in bits (TRIB) rises from 0.1953*4800 or 937 to 0.39*4800 or 1872 bits per second. Thus, a compression ratio of 2 can be expected to approximately double throughout.

Full duplex model

A much greater throughout efficiency with the 'stop and wait ARQ' error control procedure can be obtained when a full duplex mode of transmission is employed. Although this requires a four-wire private circuit, the modems and line do not have to be reversed. This permits an acknowledgement to be transmitted at the same data rate as the message block but in the reverse direction without the line turnaround. Thus, the information transfer ratio becomes:

$$\text{ITR} = \frac{B_{IC}{}^{*}D_1{}^{*}(1-P)}{T_R{}^{*}[T_C{}^{*}(D_2+C)+2^{*}(T_{PA}+T_p)]}. \tag{1.7}$$

Again, returning to the original 80-character block example, we obtain:

$$\text{ITR} = \frac{8^{*}80^{*}(1-0.01)}{4800[0.00166^{*}(80+10)+2^{*}(0.02+0.03)]} = 0.5293$$

When data compression results in a compression ratio of 2, we obtain:

$$\text{EITR} = \frac{8^{*}160^{*}(1-0.01)}{4800[0.00166^{*}(80+10)+2^{*}(0.02+0.03)]} = 1.06$$

With an EITR greater than unity this means that the bits of information per unit time (in compressed format) exceed the data transmission rate of the equipment connected to the line. This illustrates the value of data compression, permitting one to obtain very high effective data-transfer rates without requiring additional communications facilities.

A second variation of the full duplex model results if a 'go back *n* ARQ' error control procedure is employed. In this situation, only the block received in error is retransmitted. Here, the information transfer ratio becomes:

$$\text{ITR} = \frac{B_{IC}{}^{*}D_1{}^{*}(1-P)}{T_R{}^{*}[T_C{}^{*}(D_2+C)]}. \tag{1.8}$$

Again, substituting values from the original example we obtain:

$$\text{ITR} = \frac{8^{*}80(1-0.01)}{4800[0.00166(80+10)]} = 0.8835.$$

This is obviously the most efficient technique since the line turnaround is eliminated and the processing and acknowledgement time (T_{PA}) and propagation delay time (T_p) in each direction are nullified due to simultaneous message block transmission and acknowledgement response. If we consider the effect of a compression ratio of 2, the effective information transfer ratio can be computed as follows:

$$\text{EITR} = \frac{8*160(1-0.01)}{4800[0.00166(80+10)]} = 1.767.$$

For this example, the TRIB becomes 1.767*4800 bps or 8482 bps. Here, compression and protocol structure permit an effective information transfer of 8482 bits/s on a 4800 bps data path. In effect, the selection of an appropriate protocol coupled with effective data compression algorithms can result in a very effective data transfer. This will result in data transfers normally associated with wideband facilities occurring over conventional voice data-transmission facilities.

CHAPTER TWO

Data-Compression Techniques

The tremendous growth in remote computing during the last decade has focused the interest of communications personnel upon data compression techniques. Originally brought to data-processing user attention during the 1960s as a mechanisms for increasing the capacity of mass storage devices, compression is now being applied to the data communications field. Here, compression results in the transfer of data in shorter time periods than if such data was transmitted without the employment of a compression technique.

In this chapter, 10 distinct methods that can be employed to compress data are covered. In addition, various combinations of techniques are discussed with emphasis placed upon their utilization and efficiency. Some of the techniques covered in this chapter require a careful analysis of current or projected data traffic to be effective. None of the techniques presented requires more than a moderate level of difficulty in developing software to conduct the encoding and decoding algorithms. Most of the techniques in this chapter should be easy for end-users to implement and their implementation may result in a high degree of data reduction for a minimal amount of effort.

By the application of one or more compression techniques, operational efficiencies may be increased or transmission costs reduced. For the former, data compression will permit an increase in information transferred over a data link per unit time interval. Concerning the latter, reducing the amount of data to be physically transferred may make the employment of a lower speed data link permissible, resulting in a reduction in cost in comparison with the expense of a data link operated at a higher data rate.

2.1 NULL SUPPRESSION

Null or blank suppression was one of the earliest data-compression techniques developed. Today, this simplistic technique is employed in the commonly used IBM 3780 BISYNC transmission protocol.

Technique overview

As the name implies, null suppression is a data-compression technique that scans a data stream for repeated blanks or nulls. Upon encountering such a sequence, the blank or null characters are replaced by a special ordered pair of characters whose format is illustrated in Figure 2.1. First, a compression indicator character is employed to denote that null suppression has occurred. The second character is used to indicate the quantity of null characters that were encountered and replaced by the two-character sequence (Aronson, 1977; Ruth and Kreutzer, 1972).

A. Compression format

B. Data compression example

Original data stream XYZ␣␣␣␣␣QRX

Compressed data stream $XYZS_C5QRX$

where: S_C = special compression indicator character

C. Data scan process

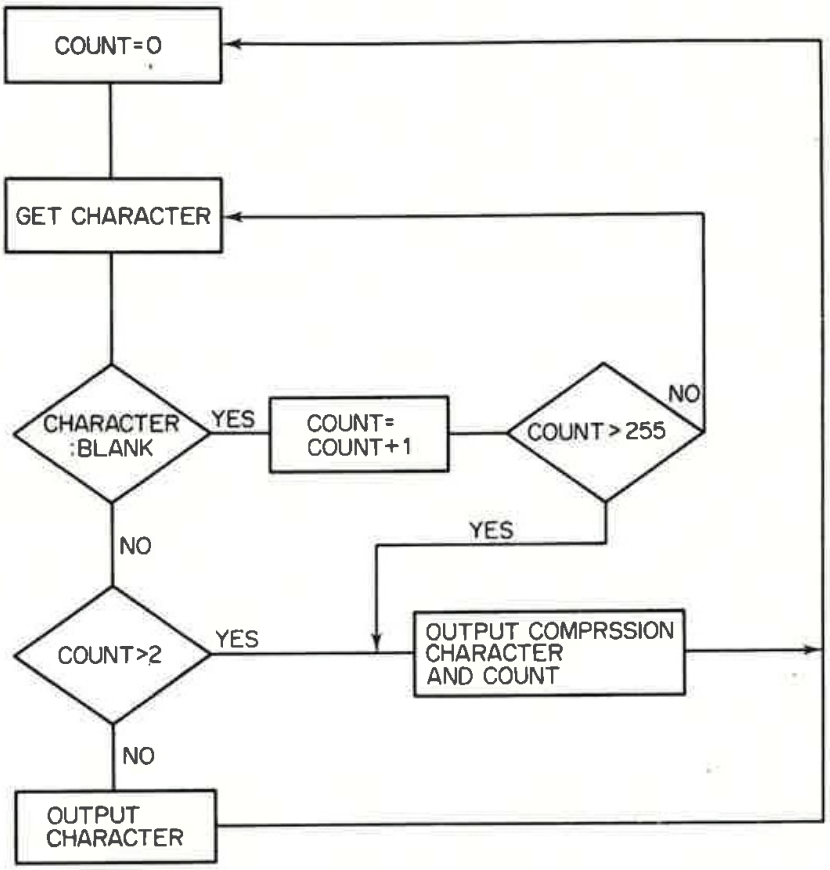


Figure 2.1 Null suppression

When the two-character sequence is transmitted within a data stream, the receiving device performs a search for the special character used to indicate null suppression. Upon detection of that character, the receiver knows that the next character contains the count of the number of nulls that were compressed. From this information, the original data stream can be reconstructed.

In the middle portion of Figure 2.1 is an example of the application of null suppression upon a data stream. Here, the character S_c indicates a special compression-indicating character, denoting that null suppression has transpired.

In the lower portion of Figure 2.1, a flow chart of the null suppression scanning process is illustrated. If we assume an 8-bit format for data characters, then the character counter can store values for up to 255 sequentially encountered nulls prior to overflowing if we start our numbering at 1, or 256 if our numbering commences by assuming a zero counter represents a value of 1.

Limitations

Since a 2-character compression sequence always results from the compression of up to 255 sequentially encountered nulls, no savings are possible unless 3 or more sequential nulls are found. Thus, a sequence of 2 nulls should not be placed into the null suppression compressed format. This is because no savings would result while the compression and decompression process requires a portion of processor time. In addition, if one is employing several data-compression techniques, we will see that 2 sequentially encountered nulls can be effectively compressed by the Diatomic encoding process. This compression technique results in a 100 per cent data reduction for the 2 null sequence situation where the null suppression technique is ineffective.

While null suppression is viewed as an elementary data-compression technique, it is very easy to implement and its payoff can be substantial. For a number of computer installations that switched from the 2780 bisynchronous transmission control sequence that does not compress data to the 3780 sequence that performs null suppression, throughput gains of between 30 and 50 per cent have been reported.

Technique variations

Two variations of null suppression can be used to compress portions of documents containing predefined or variable indentations. In one situation, it might be beneficial to reserve a group of characters from the character set to represent several predefined numbers of spaces or nulls. Thus, one character might then represent the indentation in a letter of 5 spaces, while a

second character could be used to represent 20 spaces required to tab over to the beginning of a column within a document.

Since predefined indentations represent tab stop positions, a second variation of null suppression is obtained from the employment of the tab character. If tab stops are predefined, one only has to replace a sequence of spaces or nulls by the tab character to signify that the next character begins in a particular column on the line, and all columns between the last character and the location where the next character begins are spaces or null characters. To illustrate this concept, let us assume that a portion of the document we wish to transmit is as follows:

> Now is the time to examine the relationship of defence expenditures upon the economy. For the years 1980 to 1984 our analysis shows:
>
> | Year | Guns | Butter |
> |---|---|---|
> | *xxxx* | *yyyy* | *zzzz* |

Note that there are four distinct tab stop locations in this document—the indentation of a paragraph and the three column positions. Thus, a tab stop followed by the character 'N' could be used to position the beginning of the paragraph into its appropriate location. Since the indentation occurs prior to the first column position, to position the 'Y' in year would require two tab stops to be issued. Similarly, the 'G' in guns would have to be preceded by three tab stop characters and so on. As the number of unique indentation and column location positions increases in a document or between different documents, the number of tab stop characters that may have to be issued to represent a predefined location could result in the expansion of data instead of its compression. To prevent such situations from occurring, as well as to eliminate the requirement of having prior knowledge about indentation and column locations, a variable tab stop procedure can be employed. In using variable tab stops one simply substitutes a tab stop character and the column position to tab to in place of the spacing between columns. Returning to the previous example, if 'year' began in column 15 while 'guns' and 'butter' began in columns 30 and 45 respectively, the line column heading labels could be replaced by the sequence Ts15 Year Ts30 Guns Ts45 Butter, where Ts represents the tab stop character.

2.2 BIT MAPPING

This compression technique is effective when the data to be operated upon consists of a high proportion of specific data types, such as numerics, or a large proportion of a specific character, such as blanks. As the name implies, a bit map is employed to indicate the presence or absence of data characters or the fact that certain data characters have been operated upon previously and must be operated upon again to return the data into its original format.

Encoding process

To examine the bit mapping technique and its applications, we will first see how it can be employed to implement a version of null suppression. In the left-hand portion of Figure 2.2, a portion of a data stream consisting of 3 data characters and 5 nulls is illustrated. Here, the 5 nulls represent $62\frac{1}{2}$ per cent of the content of the string and are spread throughout the data stream in a random sequence. Since null suppression is only effective when 3 or more sequential blanks are encountered, its use would only reduce the string from 8 to 7 characters in length.

Through the use of a bit map appended in front of the string, we can indicate the presence or absence of nulls and thereby reduce the size of the data string. In the lower portion of Figure 2.2, the employment of a bit map character is illustrated where all nulls are dropped from the data string and the bit which corresponds to the null position is set to zero while the bit position in the map which corresponds to a non-null or data character is set to one.

In comparing the compressed data string with the original data string, the 8 characters of data to include nulls have been reduced to 4 characters, 3 data characters and the bit map character. This results in a compression ratio of 2:1 for this particular application.

Hardware considerations

The bit map character illustrated in Figure 2.2 denotes non-null data character positions by location, from left to right. By reversing the bit map order, the data element positions can be indicated from right to left. Figure 2.3 indicates the two different methods of forming the bit map to represent the compressed data string. Using the bit map data element positioning technique

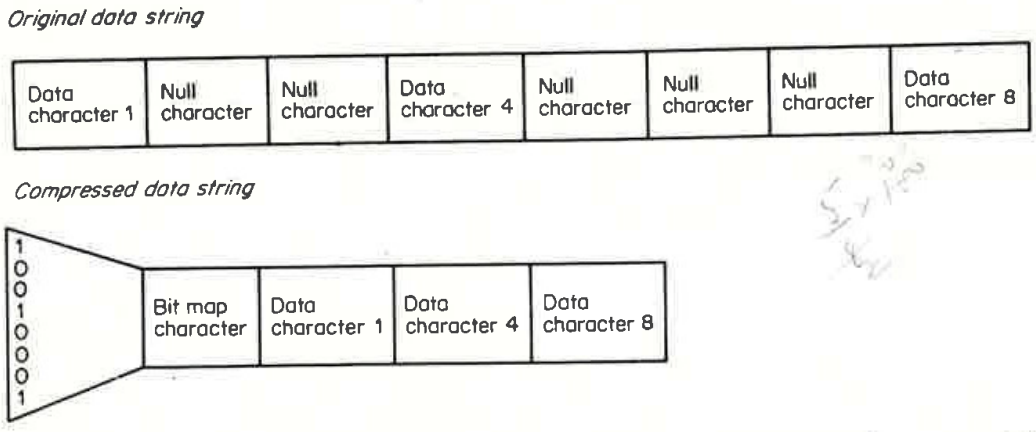


Figure 2.2 The bit mapping process. In a typical data stream, there is a high probability that one or more characters are repeated. Using one character to serve as a bit map can serve to eliminate the high frequency of occurrence of characters from a data stream

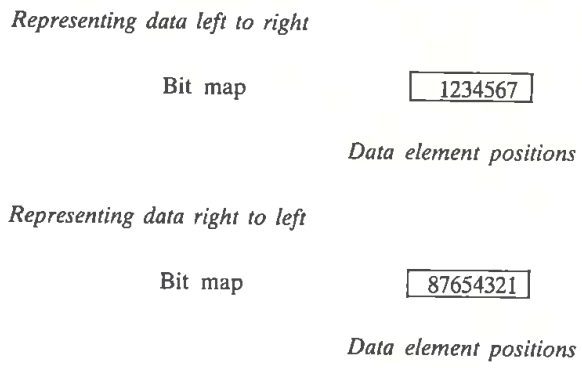


Figure 2.3 Bit map element positioning. Two methods can be employed to represent the compressed data string in the bit map—data represented left to right and right to left

illustrated in the lower portion of Figure 2.3, the bit map character resulting from the original data stream as illustrated in Figure 2.2 would become 10001001. The instruction set of the hardware device under consideration for peforming the bit map suppression technique will govern the method of bit map element positioning to be employed. This can be easily explained by first examining a flow chart of the functions that have to be performed on the original data string in order to construct the bit map and the compressed data string.

The bit map suppression process is illustrated functionally in Figure 2.4. The software routine to compress data must first initialize the bit map position counter (1), the bit map (2) and a character counter (3). After a character is obtained (4), the character counter is compared with eight (5). If a match occurs, eight incoming characters have been processed and we can exit from the routine (10). If no match occurs, the character counter is incremented (6) and the character under examination is compared with a null character (7). If the character under examination is not a null, the bit map position is set equal to a binary one (8). If the character is a null, this function (8) is bypassed. Next, the bit map position is either incremented or decremented (9) so that the bit map is prepared to be set to a zero in the following bit location if the next character examined is a null. Finally, after eight characters have been processed, the count equals eight (5) and the routine exit branch is taken (10).

From a hardware standpoint, the method used to perform the functions indicated in blocks (8) and (9) of Figure 2.4 depends upon the shift and logical instructions available for programmer utilization. This interrelationship can be viewed by denoting the effect on the bit map character as succeeding data characters are examined. In Figure 2.5, the effect on the bit map and 'mask' as a progression of data characters is examined is illustrated. Here, the mask is simply a binary one that is shifted through the 8-bit map positions and logically 'OR'd' with the bit map when the data character is not a null.

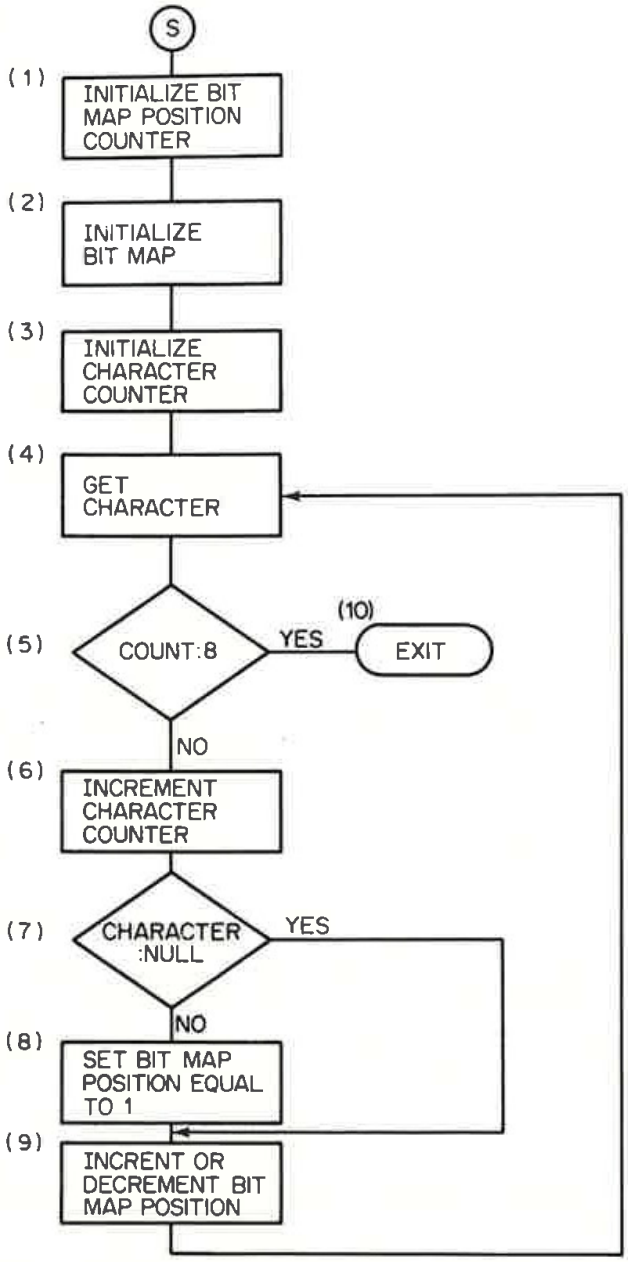


Figure 2.4 Bit map suppression function flow chart

In examining the mask, we can note that a logical or arithmetic left shift operation is required if we wish to position our bit map so that the right-hand bit indicates the presence or absence of a null character in the first element of the original data string. Thus, from a hardware viewpoint, the shift instruction available will be a governing factor with respect to how the bit map elements are positioned. Although most minicomputers and certainly all large computers have both left and right shift functions, a few microprocessors may have limited shifting capability. Such capability should be examined prior to attempting to implement this technique.

Data character	Initial bit map	Mask	Bit map+ mask (or bit map if null)
Data	00000000	00000001	00000001
Null	00000001	00000010	00000001
Data	00000001	00000100	00000101
Data	00000101	00001000	00001101
Null	00001101	00010000	00001101
Null	00001101	00100000	00001101
Null	00001101	01000000	00001101
Data	00001101	10000000	10001101

Figure 2.5 The bit map masking process. The mask character is a binary one shifted through all bit positions and logically OR'd with the bit map when the data character is not a null

Suppression efficiency

In the previous example, the bit map character contained 8 bits. While the example showed a 50 per cent reduction in characters from the original data string to the compressed data string, consider what happens to the compression efficiency when the percentage of nulls in the data string decreases. Table 2.1 shows the compression ratio based upon the percentage of nulls contained in the string for an 8-bit map. When there are no null characters, the resultant data string increases in size by 1 character as a result of the addition of the bit map character, producing a compression ratio of 0.888. This means that for the worst case situation where there are no null characters to be suppressed, an extra 12.5 percentage of data will result from the employment of this compression technique.

We can develop a mathematical model of suppression efficiency as follows. If p is the probability that any given character is a null, the expected number of nulls in a string of length S characters is Sp. Using null compression this will be encoded as a string of length

Table 2.1 Compression efficiency and null percentage

Null percentage	Resultant string size	Compression ratio
0.0	9	0.888
12.5	8	1.000
25.0	7	1.143
37.5	6	1.334
50.0	5	1.600
62.5	4	2.000
75.0	3	2.667
87.5	2	4.000
100.0	1	8.000

$$S^*(1-p)+\left[\frac{S}{8}\right]$$

and the compression ratio is then

$$\left(\frac{1}{S}\left\{S(1-p)+\left[\frac{S}{8}\right]\right\}\right)^{-1} \simeq \left((1-p)+\frac{1}{8}\right)^{-1}$$

for large value of S.

Bit map variations

In the previous discussion of the bit map procedure, we have assumed that either a null or another character appearing in large proportion to the remainder of the data is to be suppressed. For some applications, there is no particular character that is encountered more frequently than other characters; however, in certain cases one may encounter a situation where a specific type of data, such as numerics, frequently appears. One application where such a situation could exist is the process control area where numeric readings of various equipment are transmitted to a central site for processing and control signals are returned to the devices based upon certain predefined criteria. Depending upon the transmission code employed, certain economies may be obtained by the use of the bit map technique. If the data to be transmitted is in the extended binary-coded decimal interchange code (EBCDIC), then the first four bit positions of each numeric character are all ones. Thus, the bit map character could be employed to denote the number of packed characters in the compressed string, each character containing two digits with the leading four bit positions stripped. This technique is illustrated in Figure 2.6 and is quite similar to the half-byte packing technique that is covered later in this chapter.

Original data string

1
8
4
6
2
7
9
8
3
2

Compressed data string

00001010 BIT MAP	
8	1
6	4
7	2
8	9
2	3

Figure 2.6 Half-digit suppression. In the half-digit suppression technique, the contents of the bit map specify the number of digits that follow, packed two per character

Technique constraints

One key limitation of the bit map technique is that it is applicable to data having fixed size units, such as characters, bytes or words. When used to suppress a particular character, such as a null, the compression ratio of this technique is directly proportional to the percentage of occurrence of that character in the original data stream. Thus, if one character in a data string occurs 30 per cent of the time while the second most frequently encountered character occurs, say, 25 per cent of the time, this technique ignores the high percentage of occurrence of the second character or any other characters. As we shall see, a technique known as run-length encoding can be employed to take advantage of the adjacent redundancy of occurrence of all characters in a data stream.

2.3 RUN LENGTH

Run-length encoding is a data-compression method that will physically reduce any type of repeating character sequence, once the sequence of characters reaches a predefined level of occurrence. For the special situation where the null character is the repeated character, run-length compression can be viewed as a superset of null suppression (Rubin, 1976; Ruth and Kreutzer, 1972).

Operation

In a similar way to the method used to effect null suppression, the employment of run-length encoding requires the use of a special character to denote that this type of compression has occurred. This compression indicator character is normally followed by one of the repeating characters which was in the encountered string of repetitious characters. Finally, a count character signifies the number of times the repeated character occurred in the sequence.

When codes such as ASCII or EBCDIC are employed, a good choice for the special character is one that will not occur in the data string. For each of these codes there are numerous unassigned characters with unique bit representations that can be used. For situations where the character set contains no unused character, such as in the BAUDOT 5-level (bit) code, this technique may still be used by selecting a character that may not be used twice in succession, such as a letter shift or figure shift, to indicate that compression has occurred. The reader should refer to Appendix A (p. 00) for additional information concerning the selection and utilization of compression-indicating characters from different character codes.

Encoding process

The run-length compression process results in a string of repeated characters being converted into a compressed data string as shown in Figure 2.7 (Aronson, 1977). With three characters required to denote compression, run-length encoding is only effective when a data string contains a sequence of four or more repeated data characters.

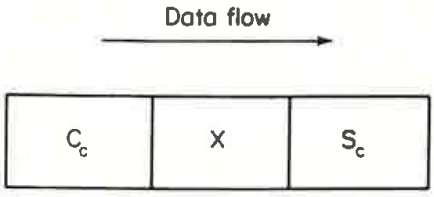


Figure 2.7 Run-length encoding, general compression format. In run-length encoding, a special character, repeated data character and character count character are required to indicate the compression parameters

S_c = Special character indicating compression follows.
X = Any repeated data character.
C_c = Character count. This count is the number of times the compressed character is to be repeated

Table 2.2 Applying run-length encoding

Original data string	Encoded data string
$******55.72	S_c$*655.72
----------	S$_c$-9
GunsbbbbbbbbbbButter	GunsS$_c$b1ϕButter

Three examples of the application of run-length encoding upon repeating character sequences are presented in Table 2.2. Note that S_c represents the special character used to indicate the occurrence of run-length encoding while the symbol ϕ is used to indicate the presence of a blank character.

With the null suppression format requiring two characters, employing run-length compression to suppress nulls always results in one additional character generated in the compressed data stream. While this is not significant when long strings of nulls are compressed, numerous short strings of nulls could result in an excess quantity of compressed data. This suggests that one should consider the use of a mixture of several algorithms to perform data compression.

The major steps in the run-length encoding process are shown in Figure 2.8 through the use of a systems flow chart. Initially, a character counter (1) and character repetition counter (2) are set to zero. After a character in the original data string is obtained (3), the character counter is incremented (4) by one. The character count is then compared with one (5). In the first cycle, this comparison always holds true and the character is then placed in a buffer (temporary storage) area (6) for later processing if the original data string is found to contain four or more repetitive data characters. For the second and subsequent cycles, the character obtained from the original data string (3) is compared with the character placed in storage (7). If the present character is equal to the character in storage, compression may be possible if four or more identical characters are encountered in sequence. Thus, when the character equals the stored character, the repeat counter (8) is incremented by one and another character is obtained from the original data string (3). If the present character under examination does not equal the character stored (7), the repeat counter is compared with four (9). If less than four, no compression is worthwhile since three characters must be used to encode compressed data. When the repeat counter is equal to or greater than four (9), the compression format (10) can now be set.

Special considerations

In the basic encoding flow chart illustrated in Figure 2.8, it was assumed that the repeat counter was capable of having an unlimited range of values. In reality, the maximum value that the repeat counter can contain is a function of the character code level employed. For an 8-level (8 bits per

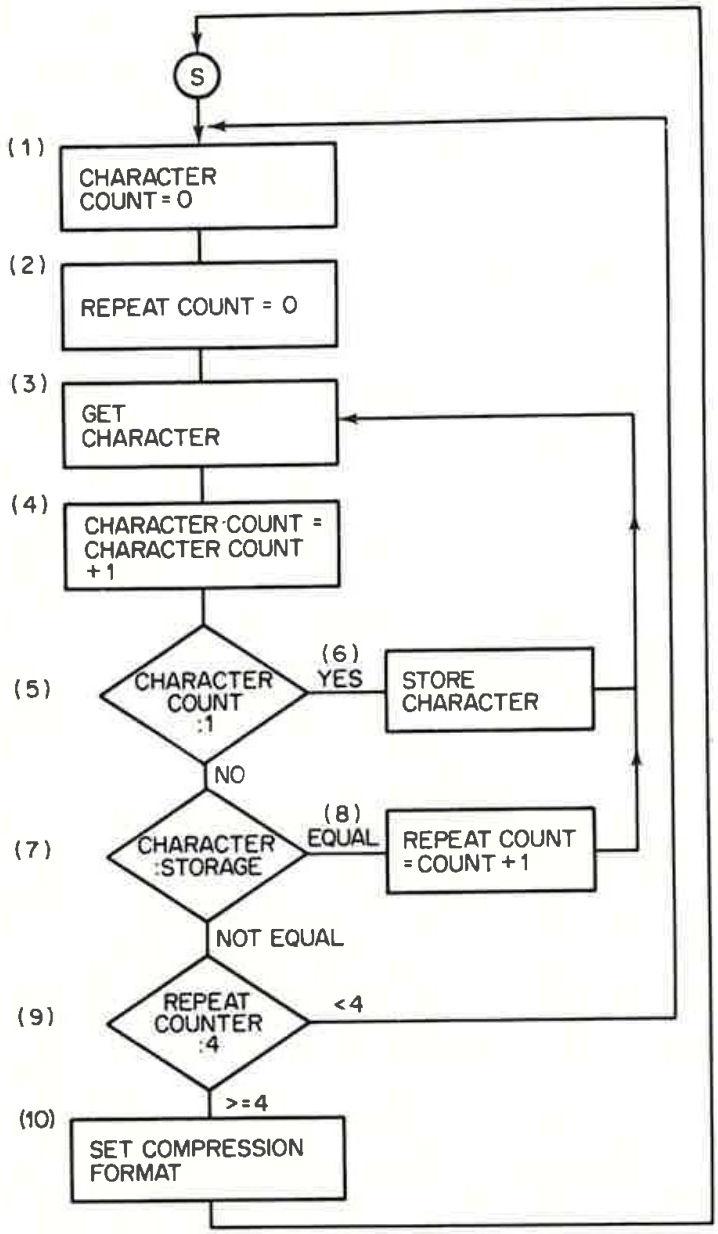


Figure 2.8 Basic run-length encoding process

character) character code, a maximum between 255 and 260 repetitive characters can be represented by the character counter. The exact value will depend upon how the character counter is employed. In most situations, the actual character counter value is used as the number of repetitive characters. In this mode, the counter's maximum value is $2^8 - 1$ or 255. Since the compression format illustrated in Figure 2.7 occurs only when 4 or more repetitive characters are encountered, the presence of a character count character in itself implies that 4 or more repetitive characters exist. Thus, a character counter of all bits zero can be used to indicate 4 repetitive characters while a character counter of all bits set to 1 would then indicate 260 repetitive characters. Once the method of employing the repeat counter is determined,

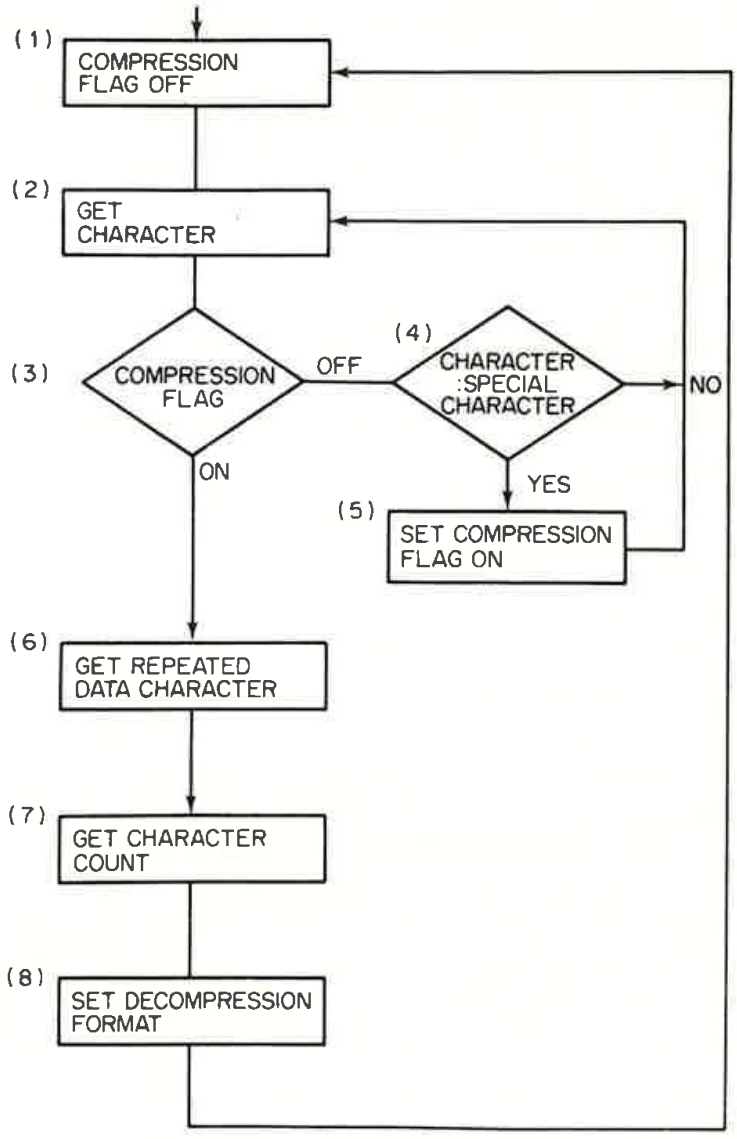


Figure 2.9 Run-length decoding process

the flow chart in Figure 2.8 must be modified to add an additional repeat counter comparison to test for the maximum value permitted to be stored in the character counter.

Decoding

The functions necessary to decompress data compressed according to the run-length encoding process are illustrated in Figure 2.9 in flow chart format. At the beginning of the decompression procedure, a compression flag is turned off (1) and a character is obtained from the compressed data string (2). Next, if the compression flag is off (3), the character is compared with the special run-length compression indicator character (4) to determine if run-length compression has occurred. If the character is not the special character, the next character is obtained (2). If the character is the run-

length compression indicator character, the compression flag is turned on (5) and the next character is obtained (2). On the next pass, since the compression flag is on (3), the following character obtained (6) is the repeated data character while the next character (7) contains the character count. Once these characters are obtained, the decompression format can be initiated (8).

Utilization

The most popular utilization of run-length encoding is the subset known as null suppression. This compression technique is primarily encountered in the IBM 3780 BISYNC protocol. Space compression is a standard feature of this protocol when the 3780 device is operating in the line mode with non-transparent data. Here, each group of 2 or more consecutive space characters, up to 63, is replaced by an IGS character if the transmission code is EBCDIC or a GS character if the code is ASCII. Either character is followed by a space count character that defines the number of spaces removed. For the situation where 64 or more consecutive space characters occur, an additional IGS or GS character and space-count character are inserted.

On Honeywell systems, a version of run-length compression is used in their general remote terminal system (GRTS) software on front-end processors communicating with remote terminals under the remote computer (RC) protocol. In addition, the same type of compression is used on the Honeywell stand alone tape-to-tape system (SATTS) for the transmission of reels of magnetic tape between locations. In this version of run-length encoding, a record is examined for a series of three or more occurrences of the same data character. When such a situation occurs, the series is compressed and a string of repeated characters is formed as illustrated in Figure 2.10.

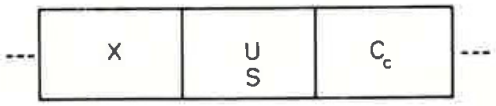


Figure 2.10 Honeywell version of run-length encoding. Run-length encoding as implemented on Honeywell systems differs slightly from most other computer manufacturers

X = Any repeated data character.
US = The ASCII character (0011111).
C_c = A 6-bit binary count. The BCD character represented by the binary count must be translated to ASCII for transmission to the communications subsystem. This count is the number of times the compressed character is to be repeated (maximum 63)

Efficiency

Run-length encoding efficiency depends upon the number of repeated character occurrences in the data to be compressed, the average repeated character length and the technique employed to perform compression. In Table 2.3, the reader will find a listing of the results of the execution of a computer program written to compute the overall compression ratio based upon a varied number of repeated character occurrences in a string of 1000 data characters. Here, the number of repeated character occurrences was varied from 10 to 50 while the average repeated character length was varied from 4 to 10. It was assumed that three characters were used for the compressed data format. The computed compression ratios listed in Table 2.3 ranged from a low of 1.0101 to a high of 1.5384. Table 2.3 is a synthetic representation due to the wide divergence of actual text. Since this table covers most common compressible occurrences, it provides a handy tabular reference for readers to determine the effect of run-length encoding.

Programming examples

To illustrate the programming required to implement run length encoding and other compression techniques in this book we have developed several BASIC language coding examples. Each of these small program segments were written in the BASICA version of the BASIC programming language which operates on the IBM PC and compatible computers. Although a different programming language, such as assembler, Pascal or C, would be more efficient, our utilization of BASICA was based upon its wide acceptance as a programming language and the ability to use the language as a learning tool for a maximum number of readers to follow. For optimum usage of the programming examples presented in this book, we suggest that one should either employ a BASIC compiler to speed up the execution of the examples or rewrite each program segment using a more optimum programming language.

In its internal operation, the IBM PC uses an 8-bit extended ASCII character code. This extended character code results in the assignment of distinct characters to ASCII values 128 through 255. Since every character from ASCII value 0 through 255 is defined and can occur when transmitting data from an IBM PC to another computer system, one might normally employ the ASCII SO (shift out) and SI (shift in) characters in developing a compression module designed to operate on ASCII data whenever there is a probability of occurrence for each character in the character set.

The SO character is used to shift out of the current ASCII character set, resulting in the ability of the user to redefine each character in the character set. Similarly, the SI character is used to shift back into the defined ASCII character set.

By using the SO and SI characters in ASCII one obtains a set of either 128 or 256 new characters, depending upon whether one is using a system

Table 2.3 Run-length encoding efficiency based upon original data string of 1000 characters

Number of repeated character occurrences	Average repeated character length	Compression ratio
10	4	1.010
10	5	1.020
10	6	1.031
10	7	1.042
10	8	1.053
10	9	1.064
10	10	1.075
20	4	1.020
20	5	1.042
20	6	1.064
20	7	1.087
20	8	1.111
20	9	1.136
20	10	1.163
30	4	1.031
30	5	1.064
30	6	1.099
30	7	1.136
30	8	1.176
30	9	1.220
30	10	1.266
40	4	1.042
40	5	1.087
40	6	1.136
40	7	1.190
40	8	1.250
40	9	1.316
40	10	1.384
50	4	1.053
50	5	1.111
50	6	1.176
50	7	1.250
50	8	1.333
50	9	1.429
50	10	1.538

that uses the 7-bit or an extended 8-bit ASCII code. This new character set can then be used to represent compression indicating characters.

Figure 2.11 illustrates the utilization of ASCII SO and SI characters to obtain a new character set where the ASCII value 082 (conventional ASCII R) is used to denote run-length encoding. In this example, a string of six Xs was assumed to be followed by a string of seven Ys. Since the ASCII value

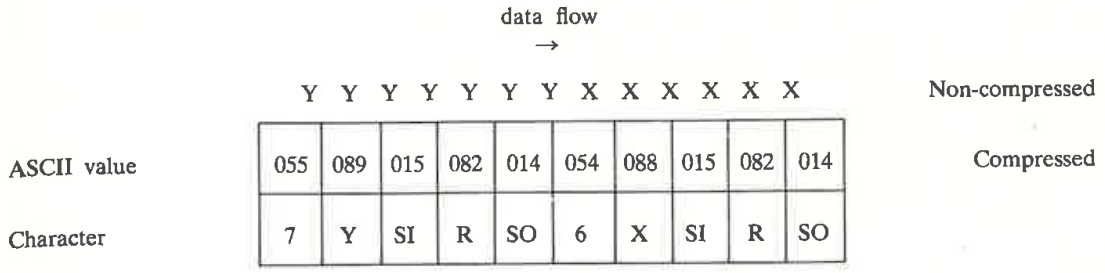


Figure 2.11 Using SO and SI characters. Using SO and SI provides a new set of characters that can be used to indicate different compression techniques, in this example, the ASCII value 082 is used to denote run-length encoding

082 in a newly defined ASCII code will be used to indicate run-length compression, one must first shift out (SO) to the new code, issue the compression indicating character (R) and then shift back into (SI) the normal ASCII code to transmit the character that was compressed (X) and the quantity of X characters compressed (6). Due to the requirement to shift out of the character set to issue the compression indicating character and then to shift back to the normal ASCII character set, two additional characters are required to represent a run-length encoded string. In addition to this technique requiring two extra characters, the use of a shift out code of 14 is used to turn on the double width mode setting of most dot matrix printers while the shift in code of 15 is used to turn on the compressed character mode setting of such printers, making the graphic illustration of this technique tedious at best.

Based upon the preceding information, it was determined that for the examples presented in this book, the use of a single character in the ASCII character set would sufficiently serve as a compression flag in addition to actually saving two characters in representing the compression of data based upon the use of run-length encoding.

Compression program

Figure 2.12 contains the listing of a BASIC program that illustrates the coding required to perform run-length compression.

To facilitate referencing BASIC programs used to illustrate the coding required to compress and decompress data, a simple naming convention has been used throughout this book. Each program filename ends with either the letter C or D, with the former used to denote a program that compresses or encodes data, while the latter references a program that decompresses or decodes data. Thus, the program labeled RUNLENC.BAS in Figure 2.12 illustrates the coding required to compress or encode data using run-length compression. The extension .BAS to the program name indicates that the file is a BASIC language program. Similarly, any data files referenced will have a filename that is a descriptor of the compression technique that will be applied against the file while its extension will be .DAT. Thus, a reference

```
10 REM RUNLENC.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO SUBROUTINES FOR COMPRESSION
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, RUNLEN.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "RUNLENC.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****RUN LENGTH ENCODING SUBROUTINE********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND COMPRESSES OUT REPETITIVE CHARACTERS*
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*********************************************
230 K=1:J=1                          'RESET INDICES
240 FOR I= 1 TO N                    'STEP THRU RECORD
250 A$= MID$(X$,I,1)                 'EXTRACT A CHAR
260 IF A$= MID$(X$,I+1,1) THEN 330 'SAME AS NEXT?
270 IF K>3 THEN 360                  'COMPRESS
280 IF K=3 THEN 420                  'DON'T COMPRESS
290 O$(J)=A$                         'STUFF IN OUTPUT BUFFER
300 J=J+1                            'BUMP BUFFER INDEX
310 NEXT I                           'GO BACK FOR MORE
320 RETURN                           'END OF STRING
330 B$=A$                            'SAVE REPEATED CHAR
340 K=K+1                            'BUMP COUNT
350 GOTO 310                         'KEEP LOOKING
355 '*********************************************************
360 'INSERT COMPRESSION NOTATION IN OUTPUT BUFFER
365 '*********************************************************
370 O$(J)=CHR$(125)                 'SET FLAG FOR RUN-LENGTH
380 O$(J+1)=B$                       'INSERT REPEATED CHAR
390 O$(J+2)=CHR$(K)                  'INSERT COUNT
400 J=J+3:K=1                        'RESET INDEX
410 GOTO 310
420 O$(J)=B$                         'STUFF 1ST REPEAT CHAR
430 O$(J+1)=B$                       'STUFF 2ND REPEAT CHAR
440 J=J+2:K=1                        'RESET INDEX
450 GOTO 310
```

Figure 2.12 RUNLENC.BAS program listing

```
900 '*****TALLY THE COMPRESSION COUNT & WRITE BUFFER******
910 '* DISPLAY BEFORE & AFTER RESULTS OF COMPRESSION      *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD   *
930 '*************************************************
931 N1=N1+N                          'TALLY INPUT CHAR COUNT
932 T=N-J+1                          'NET DIFFERENCE IN BUFFERS
936 T1=T1+T                          'SAVE COUNT FOR SUMMARY
940 FOR I= 1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE COMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "RUNLENC.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER COMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS ELIMINATED FROM ":
9999 PRINT N1;"OR ";INT((T1/N1)*100);"%":CLOSE:END
```

Figure 2.12 (continued)

to the file RUNLEN.DAT references a data file that will be compressed by a run-length compression program.

In the RUNLENC.BAS program, the ASCII value of 125 (right brace) was used as the compression indicating character, which was then followed by the ASCII character being compressed and its repetitious count in decimal notation. Thus, any string in excess of three repeating characters would be subject to compression.

Several statements in the program listing contained in Figure 2.12 warrant discussion for those readers unfamiliar with the IBM PC BASICA version of the BASIC programming language. The LINE INPUT statement in line 130 results in an entire line from a sequential file being read and assigned to the string variable X$. In line 140, the length of the string that represents one line in the data file is determined. The length of the string is then used in the FOR-NEXT loop bounded by lines 240 through 310 to process the string for repeating characters. The MID$ functions in lines 250 and 260 extract the Ith and I+1 characters from the string and compare these characters to one another. When they are equal, the repeated character is saved (line 330) and the count of repeating characters is incremented (line 340). When the repeating string of characters is broken, line 260 is FALSE and a comparison of the repeating count occurs (lines 270 and 280). When the count exceeds three (line 270) data is compressed by the coding contained

in line 360 through 400. If the count equals three there is no advantage to be gained from run-length compression and the routine bounded by lines 420 through 450 simply adds the input characters to the output buffer. When the Ith and I+1 characters are not equal, the Ith character in the input buffer is simply placed in the output buffer (line 290). Lines 900 through 9999 are not actually part of the run-length encoding process and are only included to facilitate file operations and comparison of the input and output buffers to obtain a measurement of the efficiency of this technique when applied to a data file containing a variety of repeating data strings.

Figure 2.13 illustrates a sample execution of the RUNLENC.BAS program using an ASCII file named RUNLENC.DAT as input to the program. Note that RUNLENC.BAS was purposely written to first list the contents of the file prior to its compression which is illustrated in lines 1 to 8 at the top of Figure 2.13. Next, the program lists the file after its contents were compressed based upon the application of run-length encoding to the data contained in the file.

It should be noted that string decimal values ranging below ASCII 32 were purposely omitted from inclusion in the test file since they would cause unwanted carriage returns, line feeds and other non-printable characters to be displayed, which would make an illustration of this compression technique difficult to comprehend. They would, however, be quite appropriate in normal string compression and decompression applications.

```
ENTER ASCII FILENAME. EG, RUNLEN.DAT
? RUNLEN.DAT
PATIENCE - INPUT PROCESSING
FILE RUNLEN.DAT BEFORE COMPRESSION:
1 BEGIN**************************************
2 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
3 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
4 PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
5 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
6 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
7 TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
8 **************************************END
FILE RUNLEN.DAT AFTER COMPRESSION:
1 BEGIN}*#
2 }R!
3 }E"
4 }P#
5 }E$
6 }A%
7 }T&
8 }*%END
 261  TOTAL CHARACTERS ELIMINATED FROM  309 OR  84 %
Ok
```

Figure 2.13 Sample execution of RUNLENC.BAS

Modifications to consider

The ASCII 125 character was used as a compression indicating character due to its representation as a right brace on most printers. Normally, if one's source data does not include characters beyond ASCII 127, then a character in the extended ASCII character set, such as ASCII 129 or another beyond ASCII 127, should be used to represent the occurrence of run-length encoding. For the preceding example, ASCII 129 was purposely excluded because its display on a monitor as the character ii will be printed on some printers as the £ (pound) character, while other printers simply ignore characters beyond ASCII 127. To correctly print characters beyond ASCII 127 using an IBM PC requires one to have a printer capable of printing the extended ASCII character set. In addition, a special disk operating system (DOS) program called GRAFTABL which is available under DOS 3.0 and higher versions of the operating system must be loaded into the computer prior to printing data. Due to this, the ASCII 125 character was used for illustrative purposes as the compression indicating character.

If a character beyond ASCII 127 is used to indicate the occurrence of compression and that character naturally occurs in one's data a false indication of compression will result. To prevent a receiving device from misinterpreting the character as an indication that run-length compression occurred, the program can be modified to send two such characters whenever a compression indicating character occurs naturally in a data stream. Then, at the receiving device the decompression program would first examine each character for the occurrence of a compression indicating character, however, when encountered it would not immediately signify run-length encoding had occurred. The program would then examine the next character to ascertain if that character is also a compression indicating character. If it is, this would serve as an indicator that one compression indicating character occurred naturally in the data, resulting in the removal of the second compression indicating character by the receiver.

Decompression

In Figure 2.14, the reader will find the program listing of RUNLEND.BAS, which is the program developed to decompress data previously compressed by the RUNLENC.BAS program. To as great an extent as possible, program variables and coding modules have been kept the same between compression and decompression programs presented in this book to facilitate their utilization and explanation.

Similar to the previously examined compression program, this program processes data on a line by line basis. The LINE INPUT statement in line 130 reads a line of data from the file used for input. Next, in line 140 the length of the line is determined.

```
10 REM RUNLEND.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO DECOMPRESSION SUBROUTINE  *
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, RUNLENC.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "RUNLEND.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****RUN LENGTH DECODING SUBROUTINE********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND DECOMPRESSES RUN-ENCODED CHARACTERS *
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*********************************************
230 K=1:J=1                          'RESET  INDICES
240 FOR I= 1 TO N                    'STEP THRU RECORD
250 A$= MID$(X$,I,1)                 'EXTRACT A CHAR
260 IF A$= CHR$(125) THEN 360        'COMPRESSION FLAG?
290 O$(J)=A$                         'STUFF IN OUTPUT BUFFER
300 J=J+1                            'BUMP BUFFER INDEX
310 NEXT I                           'GO BACK FOR MORE
320 RETURN                           'END OF STRING
355 '*****************************************************
360 'DECODE COMPRESSION NOTATION TO OUTPUT BUFFER
365 '*****************************************************
370 K$= MID$(X$,I+2,1)               'GET REPEAT COUNT
380 A$= MID$(X$,I+1,1)               'GET REPEAT CHAR
390 K= ASC(K$)                       'SET UP INDEX
400 FOR L= J TO J+K                  'SET OUTPUT LOOP
410 O$(L)= A$                        'STUFF REPEAT CHAR
420 NEXT L                           'KEEP GOING
430 J= L                             'BUMP OUTPUT INDEX
440 I= I+3                           'BUMP INPUT INDEX
450 GOTO 250                         'DONE
```

Figure 2.14 RUNLEND.BAS program listing

The subroutine bounded by lines 180 and 320 is then invoked. In this subroutine the string representing one line from the input file is examined on a character by character basis, using the MID$ function in line 250 to extract one character at a time from the string. In line 260, each extracted character is compared to the character value of 125 which is the right brace character to determine if a compression indicating character occurred. If so, a branch to line 360 occurs where the repeated count and the repeated

```
900 '*****TALLY THE DECOMPRESSION COUNT & WRITE BUFFER****
910 '* DISPLAY BEFORE & AFTER RESULTS OF DECOMPRESSION    *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD   *
930 '*************************************************************
931 N1=N1+N                                 'TALLY INPUT CHAR COUNT
932 T=N-J+1                                 'NET DIFFERENCE IN BUFFERS
936 T1=T1-T                                 'SAVE COUNT FOR SUMMARY
940 FOR I= 1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE DECOMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "BYTED.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER DECOMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS INSERTED"
9999 CLOSE:END
```

Figure 2.14 (continued)

character are extracted from the string in lines 370 and 380. Next, an index is obtained based upon the numerical value of K$, using the ASC function in line 390. This is followed by the FOR-NEXT loop bounded by lines 400 to 420, which place the repeated character in the output buffer the required number of times to match the count character. Then the J and I indexes are increased and the program branches back to line 250.

If a compression indicating character did not occur in the data, line 290 is executed. This line causes the character extracted from the string to be placed directly into the output buffer. Next, the J index is incremented by 1 in line 300 and the boundary of the original FOR-NEXT loop checks to determine if the end of the loop was reached in line 310.

The statements from line 900 to the end of the program were included to tally the decompression count and display the before and after results of the program. Thus, this part of the program was included for illustrative purposes only.

Figure 2.15 contains a sample execution of the RUNLEND.BAS program. The reader will note that the data file RUNLENC.DAT was used as input to the program. This data file was created by the execution of the RUNLENC.BAS program and the top eight numbered lines in Figure 2.15 correspond to the lower eight numbered lines in Figure 2.13. Since the decompression program returns the compressed data to its original format,

```
ENTER ASCII FILENAME. EG, RUNLENC.DAT
? RUNLENC.DAT
PATIENCE - INPUT PROCESSING
FILE RUNLENC.DAT BEFORE DECOMPRESSION:
1 BEGIN}*#
2 }R!
3 }E"
4 }P#
5 }E$
6 }A%
7 }T&
8 }*%END
FILE RUNLENC.DAT AFTER DECOMPRESSION:
1 BEGIN****************************************
2 RRRRRRRRRRRRRRRRRRRRRRRRRRRRRR
3 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
4 PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
5 EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
6 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
7 TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
8 ****************************************END
 276  TOTAL CHARACTERS INSERTED
Ok
```

Figure 2.15 Sample execution of RUNLEND.BAS program

the eight numbered lines at the bottom of Figure 2.15 are exactly the same as the eight numbered lines at the top of Figure 2.13.

2.4 HALF-BYTE PACKING

This data-compression technique can be viewed as a derivative of the bit mapping process. It can be successfully used under several data structure conditions; however, unlike the bit mapping technique, it will never result in a compression ratio of less than unity.

As originally developed, half-byte packing takes advantage of the structure of certain characters in a character set. This technique is effective when a portion of the bit pattern used to represent those characters becomes repetitive. As an example of this type of situation, consider the EBCDIC character set where the first four bit positions used to represent numerics are all set to binary ones as illustrated in Table 2.4.

If a non-compressed data string contains eight level EBCDIC coded characters, then run-length encoding does not permit compression of a sequence of digits that does not repeat by character. Since the first four bits, however, do repeat, compression can be accomplished if one can pack two

Table 2.4 EBCDIC numeric representation. When an 8-bit byte is used to contain numeric values coded in the EBCDIC character set, the first 4 bit positions are always set to all 1s

Bit structure	Numeric character
1111 0000	0
1111 0001	1
1111 0010	2
1111 0011	3
1111 0100	4
1111 0101	5
1111 0110	6
1111 0111	7
1111 1000	8
1111 1001	9

numerics into one character. In a similar way to run-length encoding, a special character is required to indicate that half-byte packing has occurred. Again, like run-length encoding, this character should be selected from one of the unassigned characters in the character set.

When data characters do not have a repetitive bit structure, half-byte packing can still be successfully employed under certain predefined conditions. One example would be to predefine the occurrence of the dollar sign, all 10 numerics, the comma, asterisk and decimal point characters in succession as suitable for compression by half-byte packing. In Table 2.5, the bit structure of ASCII data characters commonly used for financial representations is listed. If the occurrence of a string consisting of any numeric digit as well as a comma, decimal point, dollar sign and asterisk is predefined as suitable for half-byte packing, then the occurrence of such strings as '$123,456.78','123,456' or '$****123,456.78' can be compressed.

Encoding format and technique efficiency

To compress data into half bytes, several encoding formats can be considered. Each format provides a certain level of efficiency based upon the sequence of characters encountered in the original data string. One typical format is illustrated in Figure 2.16. Using this format, up to 15 sequential numeric or predefined data characters in a string occurring sequentially can be compressed. The limit of 15 characters results from the use of a 4-bit, half-byte counter to denote the number of characters compressed. If, instead of a half-byte counter, a full byte is used to indicate the half-byte packing count, up to 2^8 (or 255) numerics can be packed or 256 if the counter starts at zero to indicate 1 packed character. Since an extra half byte is required to increase the counter capacity, only when the average number of characters

Table 2.5 ASCII financial character representation. In this data representation, the parity bit was ignored. If a parity bit exists, it can be stripped along with the first three bits shown prior to the packing of the last four bits into half bytes

Bit structure	Character
011 0000	0
011 0001	1
011 0010	2
011 0011	3
011 0100	4
011 0101	5
011 0110	6
011 0111	7
011 1000	8
011 1001	9
010 0100	$
010 1100	,
010 1110	.
010 1010	*

in sequence is expected to exceed 15 should the full-byte counter be employed. Alternatively, one can use both a half-byte and a full-byte compression format and switch between the two depending upon the number of characters susceptible to half-byte packing that are encountered.

To examine the efficiency of half-byte packing, let us first explore the binary pattern of a sample data stream and the resulting compressed data stream. In Figure 2.17, the numeric sequence in the top part of the illustration consists of seven 8-bit characters or 56 bits. Through the use of the half-byte packing technique employing a half-byte (4-bit) counter, the resultant number of bits in the compressed data string is reduced to 40. In this example, the original data stream has been reduced by 28 per cent ((56 − 40)/56) for 7 sequential numerics. It should be noted that 40 bits would also be required to represent 6 sequentially encountered characters susceptible to half-byte packing if transmission is on a character by character basis. Thus, any even number of sequentially encountered characters suitable for packing with a half-byte counter requires the transmission of 4 additional null bits when data is transferred on a character-by-character basis.

In Table 2.6, the original numeric data stream and its compressed format are compared when a 4-bit counter is used. Here, the number of continuous numerics was varied from 1 to 15. Since the number of bits in the original

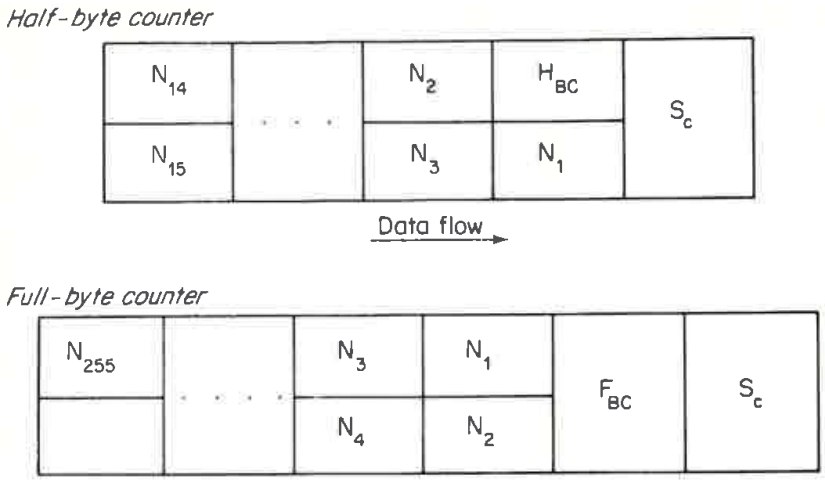


Figure 2.16 Half byte encoding format

S = Special character indicating half-byte encoding.
H_{BC} = Half-byte counter. Four bits are used to denote the number of numerics that have been packed. Number ≦ 15.
F_BC = Full-byte counter. Number ≦ 255.
N_1 to N_{255} = Up to 255 numerics packed 2 per 8-bit character

data stream is less than or equal to the number of bits in the compressed data stream, until the number of continuous numerics exceeds 4, half-byte packing should not occur until 5 or more sequential numerics or predefined characters are encountered in a data stream.

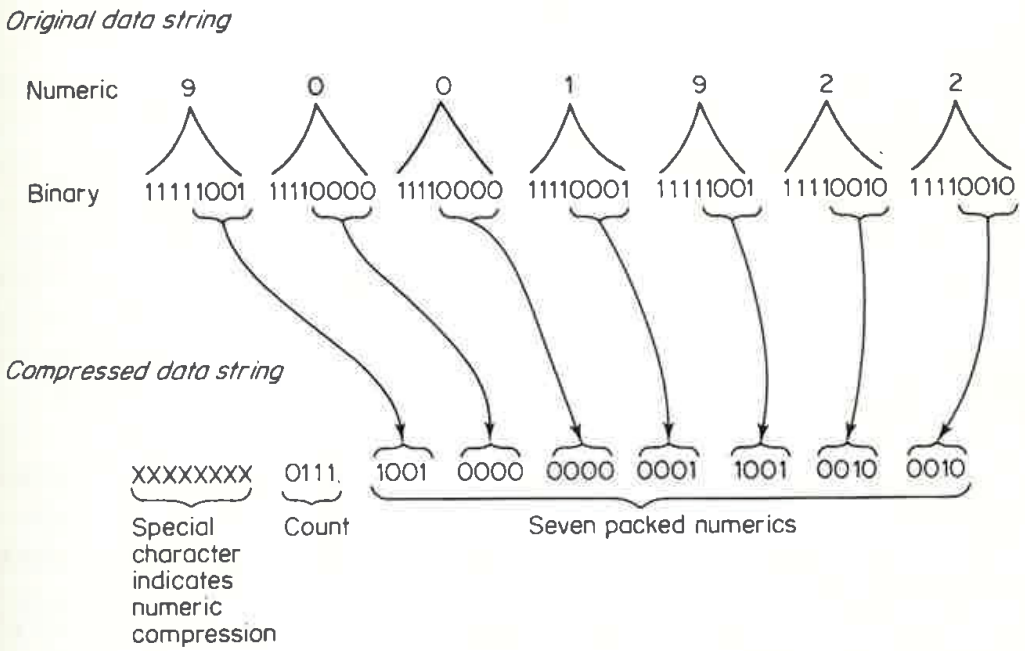


Figure 2.17 Half-byte encoding example. For 8-level character transmission, a multiple of 8 bits of compressed data is transferred. Thus, a half-byte counter with an even number of packed characters will require 4 trailing null bits

Table 2.6 Half-byte compression efficiency using a four-bit counter

Number of sequential compressible characters	Non-compressed bits	Compressed bits	Bit reduction per cent
1	8	16	N/A
2	16	24	N/A
3	24	24	N/A
4	32	32	0.00
5	40	32	20.00
6	48	40	16.66
7	56	40	28.00
8	64	48	25.00
9	72	48	33.33
10	80	56	30.00
11	88	56	36.36
12	96	64	33.33
33	104	64	38.46
14	112	72	35.71
15	120	72	40.00

The preceding can be represented mathematically as follows. For a sequence of S compressible characters, $S \geqslant 4$, the number of bits in the uncompressed string is $8S$. The number of bits in the compressed string is

$$12+4*\left[\frac{S}{2}\right]$$

giving a compression ratio of

$$\left(\frac{1}{8S}*\left\{12+4*\left[\frac{S}{2}\right]\right\}\right)^{-1}.$$

Encoding process

A half-byte packing procedure for compressing numeric characters is illustrated in flow-chart format in Figure 2.18. After the numeric character counter is initialized to zero (1), a character is obtained from the original data string. If the character is numeric (3), the counter is incremented by one (4) and the next character in the original data string is examined (2). If the character comparison (3) shows that the character is not numeric, the counter is compared with four (5). If the counter is less than or equal to four, as previously discussed there is nothing to be gained by compression

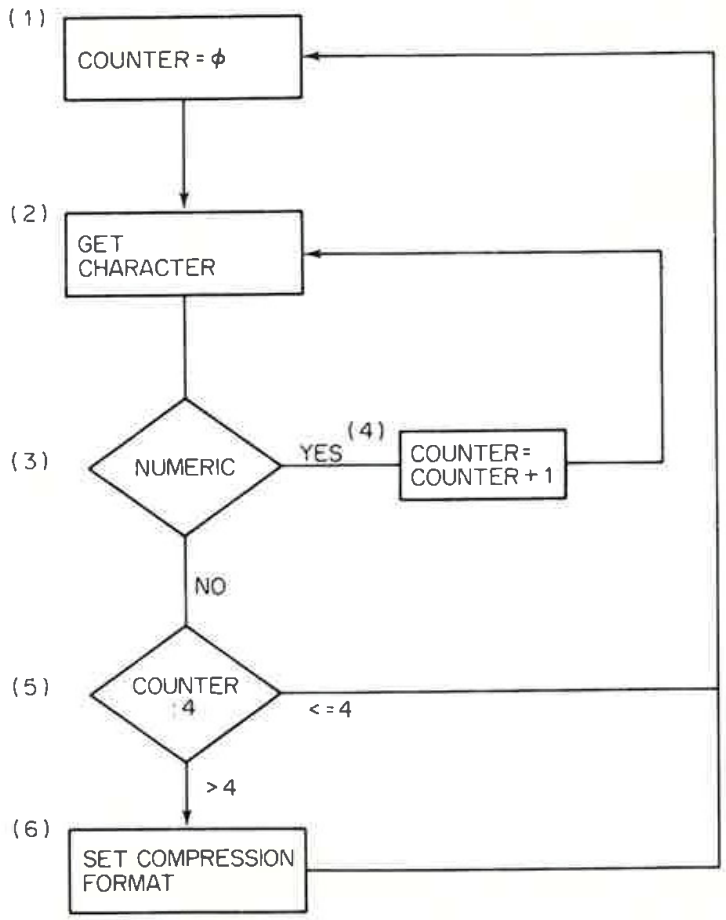


Figure 2.18 Half-byte packing process for numerics

and the counter is reinitialized to zero (1). If the counter is greater than four (5), this means that our string of sequential numerics has ended with a sufficient number of such characters that half-byte compression is effective. At this point in time, we can set the compression format (6). Although the counter in Figure 2.18 does not have a limit, if a half-byte counter is employed, the maximum number of characters that can be packed is 15. Thus, another counter comparison would be required between symbols (5) and (6).

If we desire to compress sequentially encountered strings of predefined characters to include the dollar sign, comma, period, etc., we would test for those characters in place of testing for numerics.

Buffer considerations

When a full character or multiple characters are used as a counter, buffer memory limitations must be considered in determining the maximum number of sequential characters that can be compressed, 2 to a byte. In Figure 2.19, half-byte packing buffer considerations are illustrated. As the original data

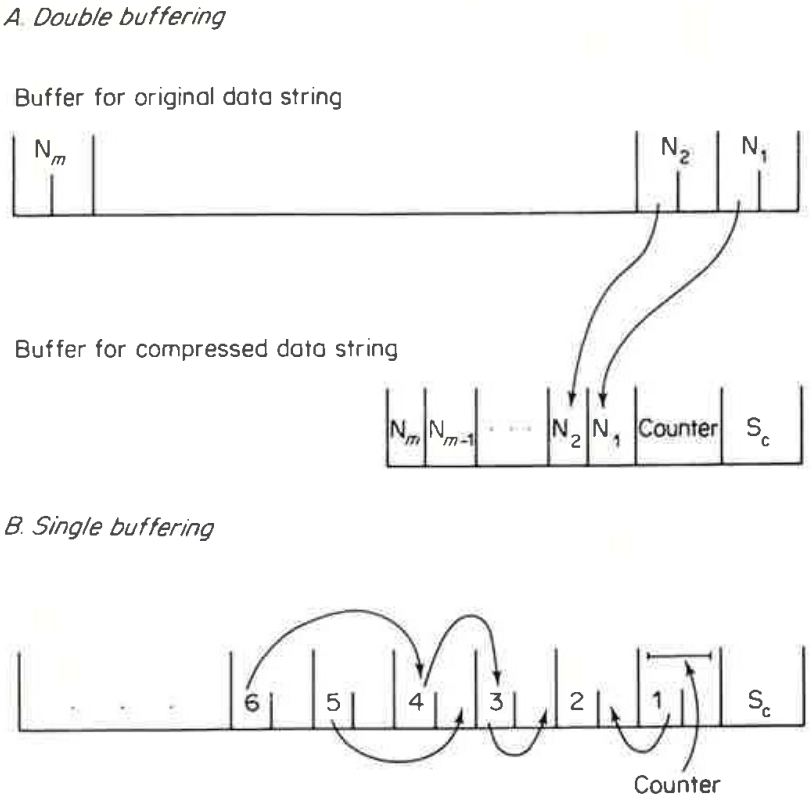


Figure 2.19 Half-byte encoding buffer considerations. Although single buffering requires additional processing, it eliminates the necessity of maintaining a separate buffer for compressed data. S_c = Special character indicating half-byte encoding

stream is examined, sequential characters suitable for packing 2 per byte are placed into a buffer as illustrated in the top portion of that figure. When the counter exceeds 4 and the next character it not suitable for packing, the data in the first buffer can be operated upon. One half of each character is then transferred to its proper location in the compressed data string buffer as illustrated in the lower portion of Figure 2.19. Since the special character used to indicate half-byte compression and a count character can be preplaced in the contiguous compressed data string buffer, this technique of double buffering is suitable if one wishes to employ a direct memory access (DMA) feature of the computer or microprocessor used for compression. Through the use of the DMA, data transfers can be effected independently of program control and data blocks are transferable on a word basis (bit parallel) to and from portions of main memory and peripheral devices. Thus, once the buffer in the lower portion of Figure 2.19 is completed, it can be set up for transmission through the use of a DMA transfer while the computer clears the original data string buffer and continues processing the incoming data stream. For an example of buffer size, consider the use of an 8-bit counter. In this situation, the buffer for the original data stream would have to be set up to hold up to 256 characters while the buffer for the compressed data stream would have to hold up to 130 characters, 256 compressed characters

packed 2 per byte, a character count and the special character used to indicate half-byte packing.

Although double buffering is illustrated in the top part of Figure 2.19 for half-byte packing, single buffering can also be used. This is shown in the lower part of that illustration. In this situation, sequential characters suitable for packing are first placed into a buffer and once a non-compressible character is encountered in the original data stream and the counter exceeds 4, the data elements in the buffer are manipulated as shown. In contrast to double buffering, this technique requires much more processing; however, it eliminates the necessity of having a separate buffer for compressed data. To determine total buffer requirements, the interrelationship of all data buffers must be examined as illustrated in Figure 2.20. In this example, the data to be operated upon is first read into a data-stream buffer where several different types of processing may be performed, depending upon the processing power and memory area availability of the computer being utilized. This data-stream buffer can be as small as 1 character or as large as a data block used for transmission. The buffer can be examined for compressible characters in several ways. First, a search can be made for any character suitable for half-byte packing; if none are encountered, the data-stream buffer can be directly transferred to the output data-stream buffer. Another method is to examine the data-stream buffer character by character. Non-compressible characters can then be sent to the output data-stream buffer while compressible characters are transferred to the original data buffer. If less than 5 compressible characters are in the original data buffer

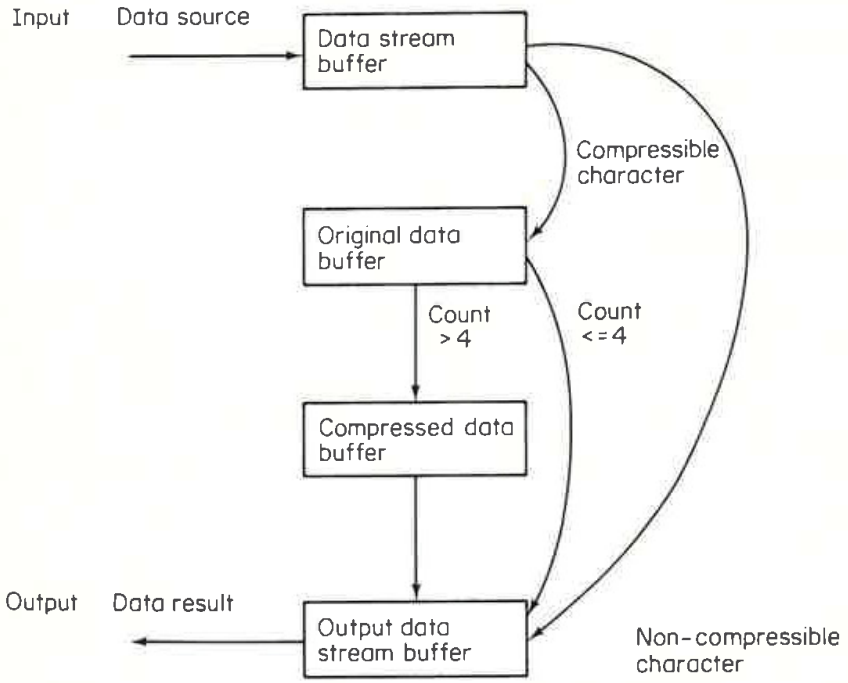


Figure 2.20 Data buffer relationships. To determine total buffer requirements, the interrelationship of all data buffers must be examined

when a non-compressible character is encountered in the data-stream buffer, the contents of the original data buffer are transferred to the output data-stream buffer. If there are 5 or more characters in the original data buffer when a non-compressible character is encountered in the data-stream buffer, the compression operation causes the contents of the original data buffer to be transferred in compressed format to the compressed data buffer. Finally, the contents of the compressed data buffer are transferred to an appropriate location in the output data-stream buffer.

Decoding

Decoding data compressed asccording to the half-byte packing technique is a relatively simple procedure. The decoding routine searches for the special character that is used to indicate that half-byte packing has occurred. Once that character is encountered, the next character or the following half byte will contain the count of the number of packed characters that follows. The special compression indicator character itself can be used to inform the decoding software whether a full- or a half-byte counter is employed. Through the use of the buffering techniques previously discussed, the packed characters can be unpacked and the original data stream reconstructed.

Encoding application

Since strings of non-repeating numerics are not compressible by run-length encoding, the use of half-byte packing can be very advantageous when data files contain many numerical sequences. If predefined characters to include the dollar sign, comma, decimal point and asterisk are added to the numerics, half-byte packing becomes a very appropriate technique for compressing financial data.

Programming examples

Two different examples of half-byte encoding of data will be presented in this section. The first set of programming examples utilizes only the digits 0 to 9 for the encoding of data, following the classical approach of half-byte packing of numeric data. The second set of programming examples extends the number of characters that can be packed two per byte by including such characters as the comma, decimal point, asterisk and dollar sign as previously discussed in this section.

Encoding

The BASIC program BYTEC.BAS is listed in Figure 2.21. This program contains the coding required to perform simple half-byte encoding of strings containing 5 or more digits in sequence. The ASCII 126 character was

```
10 REM BYTEC.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '***********MAIN ROUTINE************************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO SUBROUTINES FOR COMPRESSION
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, BYTE.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "BYTEC.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****HALF-BYTE ENCODING SUBROUTINE*********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND ENCODES NUMERIC STRINGS OF DATA INTO*
210 '* HALF-BYTE OR 4 BIT REPRESENTATION USING *
215 '* DOUBLE BUFFERING WITH O$ AS OUTPUT BUFF.*
220 '*********************************************
230 K=1:J=1                          'RESET INDICES
240 FOR I=1 TO N STEP 2              'STEP THRU RECORD
250 IF (MID$(X$,I,1)<"0") OR (MID$(X$,I,1)>"9") THEN 290
260 IF (MID$(X$,I+1,1)<"0") OR (MID$(X$,I+1,1)>"9") THEN 290
270 K=K+2                            'BOTH NUMERIC-BUMP COUNT
280 NEXT I                           'GO BACK FOR MORE
285 RETURN                           'END OF STRING
290 IF K > 4 THEN GOSUB 350          'ENOUGH TO ENCODE
300 IF K > 1 THEN GOSUB 440          'DON'T ENCODE
310 O$(J) = MID$(X$,I,1)             'OUTPUT 1ST CHAR.
320 O$(J+1) = MID$(X$,I+1,1)         'OUTPUT 2ND CHAR.
330 J=J+2:K=1                        'BUMP OUTPUT-RESET COUNT
340 GOTO 280                         'AND GO FOR MORE
345 '***** SUBROUTINE TO PERFORM HALF-BYTE ENCODING *****
350 O$(J)=CHR$(126)                  'FLAG FOR HALF-BYTE ENCODE
360 O$(J+1)=CHR$(K-1)                'INSERT LENGTH OF STRING
370 J=J+2                            'BUMP OUTPUT INDEX
380 FOR L=I-K+1  TO K STEP 2         'ENCODE 2 BYTES INTO 1
390 X= VAL(MID$(X$,L+1,1)):Y=VAL(MID$(X$,L,1))
400 O$(J)=CHR$(X+(Y*10))             'STUFF BYTE IN OUTPUT
410 J=J+1                            'BUMP OUTPUT INDEX
420 NEXT L                           'GO BACK FOR MORE
430 K=1:RETURN                       'RESET COUNT AND RETURN
435 '***** SUBROUTINE FOR STRING NOT WORTH ENCODING *****
440 FOR L=I-K+1 TO K                 'PICKUP SHORT STRING
450 O$(J)=MID$(X$,L,1)               'STUFF IN OUTPUT BUFFER
460 J=J+1                            'BUMP OUTPUT INDEX
470 NEXT L                           'GO BACK FOR MORE
480 K=1:RETURN                       'RESET COUNT AND RETURN
```

Figure 2.21 BYTEC.BAS program listing

```
900 '*****TALLY THE COMPRESSION COUNT & WRITE BUFFER******
910 '* DISPLAY BEFORE & AFTER RESULTS OF COMPRESSION      *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD   *
930 '*************************************************************
931 N1=N1+N                          'TALLY INPUT CHAR COUNT
932 T=N-J+1                          'NET DIFFERENCE IN BUFFERS
936 T1=T1+T                          'SAVE COUNT FOR SUMMARY
940 FOR I=1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
1000 PRINT
1020 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE COMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "BYTEC.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER COMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS ELIMINATED FROM ";
9999 PRINT N1;"OR ";INT((T1/N1)*100);"%":CLOSE:END
```

Figure 2.21 (continued)

used in this programming example to indicate the occurrence of half-byte encoding.

Referencing the listing contained in Figure 2.21, the array 0$ is the output buffer into which each line input from an ASCII file is placed after it is first analysed and compressed according to the half-byte encoding scheme, if so compressible. Each line from the file is read in line 130 and its length determined in line 140. Next, a branch to the subroutine starting at line 180 occurs. This subroutine steps through the record obtained from the file in increments of 2 character positions in line 240. The record is examined in increments of 2 character positions since the statements in lines 250 and 260 compare character I and character I+1 to the range between and including the digits 0 and 9. If either the Ith or Ith+1 character is in that range a branch to line 290 occurs.

To extend half-byte encoding to the characters $, . and * one could include them in the comparisons occurring in lines 250 and 260. This would be both tedious and slow, due to the time required to execute a group of MID functions joined together by many OR operators. A more elegant and speedier solution could be obtained by the creation of a one-dimensional array containing the characters to be encoded by half-byte compression. As an example, the following BASIC statements would initialize the array

HBYTE, so each of its 14 elements would contain one of the characters that would be suitable for half-byte compression.

```
DIM HBYTE (14)
FOR I=I TO 14
READ HBYTE (I)
NEXT I
DATA "$","*",".",",","0","1","2","3",
DATA "4","5","6","7","8","9"
```

An interesting and practical assignment for the reader prior to examining the second version of this program presented in this section would be the modification of the half-byte encoding subroutine to include the compression of strings containing the characters $, . and * as well as the 10 numerics.

Returning to the listing illustrated in Figure 2.21, if the Ith or Ith+1 character is not a digit the counter is incremented by two in line 270 and the subroutine continues processing the line of input obtained from the file.

When the Ith or Ith + 1 character in the string is a numeric, a branch to line 290 in the program will occur. At this location, a comparison occurs to determine if there are enough numeric characters in sequence to encode. When K is greater than four a branch to line 350 occurs. At this program location, the subroutine actually performs the half-byte encoding of the data. In line 350 the ASCII character represented by the value 126 is placed into the Jth element of the array 0$. This character is used as the compression indicating character and will be displayed as a tilde (~). In line 360, the length of the string is placed into the next element of the 0$ array and the output index is then incremented by 2 in line 370. Lines 380 to 420 perform the actual encoding of two bytes of non-compressed data into their half-byte representation and join two half bytes into a single byte.

Prior to examining the technique employed in line 400, let us first examine a conventional method to pack two numeric bytes into one byte in BASIC. In line 390, the VAL function is used to obtain the numeric part of the L and L+1 characters contained in the X$ string. Thus, X represents one numeric character while Y represents the second numeric character. Suppose X was 6 and Y was 9. Their byte composition would appear as follows:

0000 0110	X = 6
0000 1001	Y = 9

Packing two numeric into one byte can be accomplished by multiplying one character by 16 to shift it four bit positions to the left and either add it or AND it with the second character. Assuming Y is multiplied by 16, 9 × 16 is 144 and its bit composition becomes:

1001 0000	Y = 144

Then, adding X and Y results in a value of 150, whose byte composition is:

1001 0110 X + Y packed = 150

A second method to accomplish the stuffing of the two numerics into one byte was used in line 400 of the program listing contained in Figure 2.21. In this method, the numeric value of Y was first multiplied by 10 and then added to the numeric value of X. Then, the character representing the numeric value of the addition of X to Y multiplied by 10 is placed into the 0$ array as a single byte. Returning to the previous example where X was 6 and Y was 9, multiplying 9 by 10 and adding 6 results in the packing of the character that has an ASCII code of 96 into the appropriate element in the 0$ array. Thus, if this half-byte encoding routine encounters the numerical sequence of 6 followed by a 9 and there is a sufficient run of numerics to pack those two characters together they would be displayed as an apostrophe ('), since that character is represented by an ASCII 96. In this technique the ASCII codes from 00 to 99 can be employed to directly represent the 100 possible combinations of two digits.

To determine the original data one can divide the received ASCII code by 10 to obtain one numeric and use the remainder of the division process for the second numeric. Unfortunately, this technique is not applicable if the additional characters previously discussed are included in the string of characters defined as susceptible to half-byte encoding.

Again returning to the program listing contained in Figure 2.21, note that whenever the count of characters suitable for half-byte encoding is less than 5 or a non-numeric character is encountered a branch to the subroutine located at line 440 occurs. This subroutine simply takes the character from its appropriate position in the X$ string and places it in its appropriate position in the output buffer.

The last subroutine in this program was included to print a comparison of each line read from the file used for input and the half-byte encoded version of the line. In addition, the subroutine creates a file containing compressed data that will be used as an input file to test the decompression routine that will be discussed next. Starting at line 900, this subroutine also counts the characters' input and output and computes and prints the percentage of characters eliminated as a result of half-byte encoding.

Figure 2.22 illustrates the execution of the BYTEC.BAS half-byte encoding program, showing the original lines of data contained in the input file followed by its resulting compressed data. The reader should note that for clarity of illustration the input data was structured to insure that certain numeric pairs of characters were excluded. This was done to eliminate, as an example, two encoded half-bytes representing an ASCII 31 character or below, since such characters are non-printable and would not be appropriate for illustrative purposes.

```
ENTER ASCII FILENAME. EG, BYTE.DAT
? BYTE.DAT
PATIENCE - INPUT PROCESSING
FILE BYTE.DAT BEFORE COMPRESSION:
1 '+43456789876543334567898765433345678987@
2 '-98357257894538629657398577526457497356872@
3 '$434445464748495051525354555657585960l@
FILE BYTE.DAT AFTER COMPRESSION:
1 '+~&+-CYWA+!-CYWA+!-CY6@
2 '-~(b#H9Y-&)'9'UM4@911182@
3 '$~$+,-./0123456789:;1@
 54  TOTAL CHARACTERS ELIMINATED FROM  132 OR  40 %
Ok
```

Figure 2.22 Sample execution of BYTEC.BAS program

Decompression

The program BYTED.BAS listed in Figure 2.23 was written to decode or decompress data previously compressed by the BYTEC.BAS program.

Since the BYTEC.BAS program used the ASCII 126 character as a half-byte compression indicator, the BYTED.BAS program was written to search for the occurrence of this character. After a line of data is obtained from a file in line 130 of the program, the length of the line is determined in line 140. Then the subroutine at line 180 is invoked to scan the line character by character, looking for the occurrence of an ASCII 126. The FOR-NEXT loop bounded by lines 240 through 320 accomplishes this, extracting a character from the string through the use of the MID$ function in line 250 and then comparing the extracted character to ASCII 126 in line 260.

If the extracted character does not equal ASCII 126, the character is simply placed into the output buffer in line 290, the index is incremented by 1 in line 300 and the processing of the data in the loop continues. If the character is equal to ASCII 126, a branch to line 360 occurs and the decoding of the compressed data commences. First the repeat count which is the next character in the string is obtained in line 370. This character is then converted into a numeric value in line 390 since it will control the loop index for decompressing the following characters in the string that were previously encoded two per byte. This decoding is controlled by the FOR-NEXT loop bounded by lines 400 through 460. First the numeric value of the byte following the repeat count is obtained the first time line 410 is executed. In line 420, the value obtained in the preceding line is multipled by .1, which, in effect, functions as a right shift. By taking the integer of the multiplication of the byte's numeric value by .1 we obtain a numeric between 0 and 9. This numeric represents the value of Y when X and Y were previously encoded in the BYTEC.BAS program by multiplying Y by 10 and adding the value of X to the result. Since we are working with characters based upon their ASCII values, 48 is added to the value of Y in line 430 to obtain the

```
10 REM BYTED.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO DECOMPRESSION SUBROUTINE  *
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, BYTEC.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "BYTED.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****HALF BYTE  DECODING SUBROUTINE********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND DECOMPRESSES BYTE-ENCODED CHARACTERS*
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*********************************************
230 K=1:J=1                          'RESET INDICES
240 FOR I= 1 TO N                    'STEP THRU RECORD
250 A$= MID$(X$,I,1)                 'EXTRACT A CHAR
260 IF A$= CHR$(126) THEN 360        'COMPRESSION FLAG?
290 O$(J)=A$                         'STUFF IN OUTPUT BUFFER
300 J=J+1                            'BUMP BUFFER INDEX
310 NEXT I                           'GO BACK FOR MORE
320 RETURN                           'END OF STRING
355 '*********************************************************
360 'DECODE COMPRESSION NOTATION TO OUTPUT BUFFER
365 '*********************************************************
370 K$= MID$(X$,I+1,1)               'GET REPEAT COUNT
380 M= I+2                           'SETUP INPUT INDEX
390 K= ASC(K$)                       'SET UP LOOP INDEX
400 FOR L= J TO J+K-1 STEP 2         'SET OUTPUT LOOP
410 X= ASC(MID$(X$,M,1))             'GET ONE BYTE
420 Y= INT(X* .1)                    'SHIFT RIGHT
430 O$(L)= CHR$(Y+48)                'DECODE TENS POS
440 Z= INT(X-(Y* 10))                'SUBTRACT TENS POS
450 O$(L+1)= CHR$(Z+48)              'DECODE UNITS POS
455 M= M+1                           'BUMP INPUT INDEX
460 NEXT L                           'KEEP GOING
470 J= L+1                           'RESET OUTPUT INDEX
480 I= M                             'RESET INPUT INDEX
490 GOTO 250                         'DONE
```

Figure 2.23 BYTED.BAS program listing

appropriate ASCII value of the digit. This value is then an ASCII character between 0 and 9 that represents the 10s position of the previously encoded data. In line 440, the value of Y multiplied by 10 is subtracted from the

```
ENTER ASCII FILENAME. EG, BYTEC.DAT
? BYTEC.DAT
PATIENCE - INPUT PROCESSING
FILE BYTEC.DAT BEFORE DECOMPRESSION:
1 '+^&+-CYWA+!-CYWA+!-CY6@
2 '-^(b#H9Y-&>'9'UM4@91I82@
3 '$^$+,-./0123456789:;1@
FILE BYTEC.DAT AFTER DECOMPRESSION:
1 '+43456789876543334567898765433345678954@
2 '-98357257894538629657398577526457497356504@
3 '$434445464748495051525354555657585949 5@
 54  TOTAL CHARACTERS INSERTED
Ok
```

Figure 2.24 Sample execution of BYTED.BAS program

value of X to obtain the numeric value representing the unit's position in the packed data. Similar to line 430, line 450 adds 48 to the value of Z to obtain the appropriate ASCII character code that represents the decoded digit.

Figure 2.24 illustrates the execution of the BYTED.BAS program, using the file BYTEC.DAT as input to the program. Since the half-byte compression program, BYTEC.BAS, previously created this file it should be of no surprise that lines 1 to 3 at the top of Figure 2.24 are equal to lines 1 through 3 of Figure 2.22, while lines 1 to 3 at the bottom of Figure 2.24 are equal to lines 1 to 3 at the top of Figure 2.22.

Extended half-byte encoding

A second example of half-byte encoding results from the inclusion of additional characters beyond the 10 numerics into half bytes when such characters occur sequentially. In Figure 2.25, the reader will find the program listing of the PACKC.BAS program that was developed to compress a string containing numerics and/or the dollar sign ($), comma (,), decimal point (.) and asterisk (*).

Compression program

Similar to the previously described BYTEC.BAS program, a line of input is obtained from a file in line 130, the length of the line is determined in line 140 and a branch to the half-byte encoding subroutine occurs in line 150 of the program.

The subroutine bounded by lines 180 and 550 processes the line of input and encodes sequences of numerics and the special characters previously mentioned into half bytes. The FOR-NEXT loop bounded by lines 240 and 280 searches through the character positions in the string X$ that represents a line of input data. In line 242, the C(I) array flag is reset while lines 243

```
10 REM PACKC.BAS PROGRAM
20 DIM O$(132),C(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO SUBROUTINES FOR COMPRESSION
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, PACK.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "PACKC.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****HALF-BYTE ENCODING SUBROUTINE*********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND ENCODES  MIXED  STRINGS OF DATA INTO*
210 '* HALF-BYTE OR 4 BIT REPRESENTATION USING *
215 '* DOUBLE BUFFERING WITH O$ AS OUTPUT BUFF.*
220 '*********************************************
230 K=1:J=1                              'RESET INDICES
240 FOR I=1 TO N STEP 2                  'STEP THRU RECORD
242 C(I)=0:C(I+1)=0                      'RESET ENCODE FLAGS
243 A$= MID$(X$,I,1)                     'GET 1ST BYTE
244 B$= MID$(X$,I+1,1)                   'GET 2ND BYTE
246 IF A$= "$" THEN C(I)= 1              'SET 1ST ENCODE FLAG
248 IF A$= "," THEN C(I)= 2
250 IF A$= "." THEN C(I)= 3
252 IF A$= "*" THEN C(I)= 4
254 IF A$< "0" OR A$> "9" THEN 258  'SKIP OTHERS
256 C(I)= 5
258 IF B$= "$" THEN C(I+1)= 1    'SET 2ND ENCODE FLAG
260 IF B$= "," THEN C(I+1)= 2
262 IF B$= "." THEN C(I+1)= 3
263 IF B$= "*" THEN C(I+1)= 4
264 IF B$< "0" OR B$> "9" THEN 268  'SKIP OTHERS
266 C(I+1)= 5
268 IF C(I)= 0 OR C(I+1)= 0 THEN 290  'NOT CANDIDATE
270 K=K+2                                'BOTH NUMERIC-BUMP COUNT
280 NEXT I                               'GO BACK FOR MORE
285 RETURN                               'END OF STRING
290 IF K > 4 THEN GOSUB 350              'ENOUGH TO ENCODE
300 IF K > 1 THEN GOSUB 500              'DON'T ENCODE
310 O$(J) = MID$(X$,I,1)                 'OUTPUT 1ST CHAR.
320 O$(J+1) = MID$(X$,I+1,1)             'OUTPUT 2ND CHAR.
330 J=J+2:K=1                            'BUMP OUTPUT-RESET COUNT
340 GOTO 280                             'AND GO FOR MORE
350 O$(J)=CHR$(129)                      'FLAG FOR BYTE PACKING
352 MASK1= &HF0                          '11110000
354 MASK2= &HF                           '00001111
360 O$(J+1)=CHR$(K-1)                    'INSERT LENGTH OF STRING
```

Figure 2.25 PACKC.BAS program listing

```
370 J=J+2                              'BUMP OUTPUT INDEX
371 FOR L=I-K+1 TO K STEP 2            'SETUP ENCODE LOOP
372 ON C(L) GOTO 376,378,380,382,384 'USE FLAG TO ENCODE
376 X=&HA0:GOTO 388                    '10100000
378 X=&HB0:GOTO 388                    '10110000
380 X=&HC0:GOTO 388                    '11000000
382 X=&HD0:GOTO 388                    '11010000
384 X=VAL(MID$(X$,L,1))                'GET NUM VALUE OF BYTE 1
386 X=X*16                             'SHIFT 4 BITS LEFT
388 X=X AND MASK1                      'MASK LOWER HALF-BYTE
390 ON C(L+1) GOTO 394,396,398,400,410 'USE ENCODE FLAG
394 Y=&HA:GOTO 420                     '00001010
396 Y=&HB:GOTO 420                     '00001011
398 Y=&HC:GOTO 420                     '00001100
400 Y=&HD:GOTO 420                     '00001101
410 Y=VAL(MID$(X$,L+1,1))              'GET NUM VALUE OF BYTE 2
420 Y=Y AND MASK2                      'MASK UPPER HALF-BYTE
440 Z= X OR Y                          'OR THE TWO TOGETHER
450 O$(J)= CHR$(Z)                     'OUTPUT BYTE TO BUFFER
460 J=J+1                              'BUMP OUTPUT INDEX
470 NEXT L                             'GO BACK FOR MORE
480 K=1:RETURN                         'RESET COUNT AND RETURN
500 '***** SUBROUTINE FOR STRING NOT WORTH ENCODING *****
510 FOR L=1-K+1 TO K                   'PICKUP SHORT STRING
520 O$(J)=MID$(X$,L,1)                 'STUFF IN OUTPUT BUFFER
530 J=J+1                              'BUMP OUTPUT INDEX
540 NEXT L                             'GO BACK FOR MORE
550 K=1:RETURN                         'RESET COUNT AND RETURN
900 '*****TALLY THE COMPRESSION COUNT & WRITE BUFFER******
910 '* DISPLAY BEFORE & AFTER RESULTS OF COMPRESSION     *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD  *
930 '*************************************************************
931 N1=N1+N                            'TALLY INPUT CHAR COUNT
932 T=N-J+1                            'NET DIFFERENCE IN BUFFERS
936 T1=T1+T                            'SAVE COUNT FOR SUMMARY
940 FOR I=1 TO J-1                     'OUTPUT FILE LOOP
950 PRINT #3, O$(I);                   'BUFFER CHAR STRING
960 NEXT I
965 PRINT #3, ""                       'NOW WRITE TO FILE
970 RETURN                             'DONE
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE COMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN GOTO 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$: OPEN "PACKC.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER COMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS ELIMINATED FROM ";
9999 PRINT N1;"OR ";INT((T1/N1)*100);"%":CLOSE:END
```

Figure 2.25 (continued)

and 244 extract two bytes from the string. The C(I) array flag is then set to a value between 1 and 4 if the first byte of the string (A$) is one of the special characters. If A$ is a digit between 0 and 9 the C(I) array flag is then set to 5 in line 256. Otherwise, the C(I) array flag remains set to zero and a branch to line 258 occurs where the second byte represented by B$ is processed. Next, lines 258 to 266 process the second byte, assigning the C(I+1) flag a value between 1 and 5 depending upon whether one of four special characters or a numeric is encountered. If either C(I) or C(I+1) equal zero and four or more bytes containing numerics or special characters have been encountered in sequence there is enough to encode and a branch to the subroutine starting at line 350 occurs. If either C(I) or C(I+1) equals zero and between one and three bytes were encountered a branch to the subroutine beginning at line 500 occurs. This subroutine simply takes the encountered characters from the input string and places them into their appropriate positions in the output buffer.

When two bytes are extracted from the input string and no previous bytes were numeric or special characters C(I) and C(I+1) are zero and line 268 causes a branch to line 290 to occur. Since K is zero, lines 310 to 330 are then executed, resulting in the two bytes just extracted from the input string being placed into their appropriate position in the output buffer.

Lines 350 to 480 contain the coding for generating the compression indicating character which is ASCII 129 and then packing the characters eligible for half-byte compression into half bytes. Lines 352 and 354 enable two mask flags that will enable upper or lower half-bytes to be generated by ANDing the numerical value of a byte by the mask flag. Line 360 inserts the length of the string into the output buffer while line 372 examines the C flag and encodes the byte (lines 376 to 382) based upon the type of special character in the byte. If the byte is numeric, line 384 is executed. Here, the numeric value of the byte is extracted. In line 386, it is multiplied by 16 which is equivalent to a shift 4-bit positions to the left while line 388 ANDs the value of the newly formed character flag or shifted byte by the first mask. Similarly, lines 390 to 420 perform the same operation on the second byte by first examining the second C flag. Finally, line 440 adds the two half bytes into one byte by the use of the OR operator and the newly formed character that now represents two characters is placed into the output buffer. Like the other programs previously discussed, lines 900 to 9999 keep track of the compression count and generate a file named PACKC.DAT which represents the compressed data contained in the file PACK.DAT. Later the extended half-byte decompression program called PACKD.BAS will use the PACKC.DAT file as input to perform extended half-byte decompression.

Figure 2.26 illustrates the execution of the PACKC.BAS program using a three line data file whose contents are listed at the top of the figure. Since the packing of some half bytes resulted in the generation of a full byte whose ASCII code was below 31 and therefore unprintable, the first two lines of compressed data may appear odd due to the effect these characters have on the printer used by the author.

```
ENTER ASCII FILENAME. EG, PACK.DAT
? PACK.DAT
PATIENCE - INPUT PROCESSING
FILE PACK.DAT BEFORE COMPRESSION:
1 '+$43,456,789.87**654333456789876543334567898700
2 '-$9835$72.57$89.45$386,296,573.8577526457497356780O
3 '$434445464748495051525354555657585987OO
FILE PACK.DAT AFTER COMPRESSION:
1 '+,ñ;Ekx eC3Eg          eC3Eg    OO

2 '-O-Zr┼z        -Z8k)kW<wRdWIsVOO

3 '$$CDEFGHIPQRSTUVWXYOO
 61  TOTAL CHARACTERS ELIMINATED FROM  146 OR  41 %
Ok
```

Figure 2.26 Sample execution of the PACKC.BAS program

Decompression program

Figure 2.27 contains the program listing of the PACKD.BAS program that was developed to decompress data compressed by the PACKC.BAS program.

Similar in construction to the PACKC.BAS program, PACKD.BAS obtains a line of data from a file in line 130, determines the length of the line in line 140 and then branches to the subroutine starting at line 180 to perform the required decoding. The FOR-NEXT loop bounded by lines 240 and 320 extracts one character at a time from the input string, searching for ASCII 129 which is the compression indicating character used to denote the occurrence of extended half-byte compression.

When the compression flag is encountered in line 260, a branch to line 330 occurs which is the beginning of the routine that decompresses the compressed data. After initializing the masks in lines 330 and 335 the length of the string is obtained in line 340 while the FOR-NEXT loop bounded by lines 350 and 498 break up each byte into the original two characters that were previously compressed. First line 370 takes a byte and ANDs it with the first mask and divides by 16 which is equivalent to a right shift of 4 bit positions. In line 375, the character is tested to determine if it's numeric. If so, a branch to line 430 occurs where 48 is added to the character to obtain its appropriate ASCII value. If the character is not numeric, lines 380 to 410 test to determine what special character the character represents by examining its code value and then based upon its code value the character is reset to its original value. Next, lines 440 to 490 perform the same operation on the second half byte in the received character.

Program execution

Figure 2.28 illustrates the execution of the PACKD.BAS program using PACKC.DAT as the input data file to decompress. The reader will note

```
10 REM PACKD.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '***********MAIN ROUTINE************************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO DECOMPRESSION SUBROUTINE  *
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, PACKC.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "PACKD.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****HALF BYTE  DECODING SUBROUTINE********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND DECOMPRESSES BYTE-ENCODED CHARACTERS*
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*********************************************
230 J=1                              'RESET INDEX
240 FOR I= 1 TO N                    'STEP THRU RECORD
250 A$= MID$(X$,I,1)                 'EXTRACT A CHAR
260 IF A$= CHR$(129) THEN 330        'COMPRESSION FLAG?
290 O$(J)=A$                         'STUFF IN OUTPUT BUFFER
300 J=J+1                            'BUMP BUFFER INDEX
310 NEXT I                           'GO BACK FOR MORE
320 RETURN                           'END OF STRING
322 '*********************************************************
324 'DECODE COMPRESSION NOTATION TO OUTPUT BUFFER        *
326 '*********************************************************
330 MASK1= &HF0                      '11110000
335 MASK2= &HF                       '00001111
340 K= ASC(MID$(X$,I+1,1))           'GET STRING LENGTH
345 M= I+(K/2)                       'SET END OF STRING
350 FOR L=I+2 TO M                   'SETUP LOOP TO DECODE
362 Z= ASC(MID$(X$,L,1))             'GET BYTE
370 X= (Z AND MASK1)/16              'MASK LOWER HALF-BYTE
375 IF X< 10 THEN 430                'ITS NUMERIC
380 IF X= 10 THEN O$(J)= "$"         'SPECIAL
390 IF X= 11 THEN O$(J)= ","         'SPECIAL
400 IF X= 12 THEN O$(J)= "."         'SPECIAL
410 IF X= 13 THEN O$(J)= "*"         'SPECIAL
415 GOTO 440                         'SKIP IF SPECIAL
430 O$(J)= CHR$(X+48)                'OUTPUT 1ST NUMERIC
440 Y= Z AND MASK2                   'MASK UPPER HALF-BYTE
445 IF Y< 10 THEN 490                'ITS NUMERIC
450 IF Y= 10 THEN O$(J+1)= "$"       'SPECIAL
460 IF Y= 11 THEN O$(J+1)= ","       'SPECIAL
470 IF Y= 12 THEN O$(J+1)= "."       'SPECIAL
480 IF Y= 13 THEN O$(J+1)= "*"       'SPECIAL
485 GOTO 495                         'SKIP IF SPECIAL
490 O$(J+1)= CHR$(Y+48)              'OUTPUT 2ND NUMERIC
495 J= J+2                           'BUMP OUTPUT BY TWO
498 NEXT L:I= M                      'CONTINUE, BUMP INPUT INDEX
499 GOTO 310                         'GO BACK FOR MORE
```

Figure 2.27 PACKD.BAS program listing

```
900 '*****TALLY THE DECOMPRESSION COUNT & WRITE BUFFER****
910 '* DISPLAY BEFORE & AFTER RESULTS OF DECOMPRESSION    *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD   *
930 '*************************************************************
931 N1=N1+N                          'TALLY INPUT CHAR COUNT
932 T=N-J+1                          'NET DIFFERENCE IN BUFFERS
936 T1=T1-T                          'SAVE COUNT FOR SUMMARY
940 FOR I= 1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE DECOMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "PACKD.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER DECOMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS INSERTED"
9999 CLOSE:END
```

Figure 2.27 (continued)

```
ENTER ASCII FILENAME. EG, PACKC.DAT
? PACKC.DAT
PATIENCE - INPUT PROCESSING
FILE PACKC.DAT BEFORE DECOMPRESSION:
1 '+,ñ;Ekx▌eC3Eg          eC3Eg    ∅∅

2 '-O⌐Zr┼z        —Z8k)kW<wRdWIsV∅∅

3 '$$CDEFGHIPQRSTUVWXY∅∅
FILE PACKC.DAT AFTER DECOMPRESSION:
1 '+$43,456,789.87**6543334567898765433345678∅∅
2 '-$9835$72.57$89.45$386,296,573.85775264574973560∅
3 '$434445464748495051525354555657585∅∅
 55  TOTAL CHARACTERS INSERTED
Ok
```

Figure 2.28 Sample execution of the PACKD.BAS program

that the first three lines in Figure 2.27 are identical to the last three lines of Figure 2.26 while the last three lines of Figure 2.28 that represents the decompressed data are identical to the top three lines of Figure 2.26. Again, this is no surprise since the decompression program simply reconstructs the compressed data into its original form. The reader should also note that the 61 characters denoted as eliminated by half-byte compression in Figure 2.26

do not take into account the additional compression characters required to indicate each occurrence of half-byte encoding. If this was done, then a total of 55 characters would have been eliminated which matches the 55 character insertion count in Figure 2.28.

2.5 DIATOMIC ENCODING

As the name implies, diatomic encoding is a data-compression process whereby a pair of characters is replaced by a special character. The bit structure of the special character represents the encoded pair of characters and, thus, permits a 50 per cent data reduction or a 2:1 compression ratio.

Since the number of special characters that can be employed to represent different types of compression is limited, the theoretical potential of obtaining 50 per cent data reduction by substituting 1 character for every pair of characters cannot be obtained. To maximize one's potential compression requires a prior understanding of one's data composition. Once one knows the expected frequency of occurrence of pairs of characters, then the most commonly encountered pairs can be selected as candidates for diatomic encoding. The actual number of pairs selected will depend upon the number of special characters available to represent those pairs of frequently occurring characters.

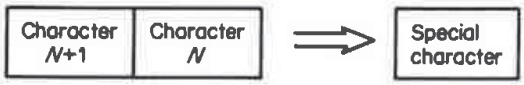
Operation

A block diagram representation of the diatomic encoding process will be found in the top portion of Figure 2.29. In the lower portion of that illustration is a flowchart denoting the major processes required to encode data diatomically. Note that the flowchart assumes that a continuous input data stream occurs. In actuality, the input and output buffers would be of finite length. Since the output buffer will always be less than or equal to the character size of the input buffer, one may be able to assign a pointer which will be incremented through the input buffer. Upon reaching the end of that buffer, the contents of the output buffer will be transmitted while the input buffer will be refilled with additional non-compressed data.

Pair frequency of occurrence

The major problem in the implementation of diatomic encoding is in determining what pairs should be represented by special characters. To perform diatomic encoding and obtain a meaningful compression ratio requires the the assignment of special characters to represent the most frequently occurring pairs of characters one will encounter in the original data steam. This means one must have some prior knowledge concerning the type of data to be operated upon so that one can base the assignment of special characters in a meaningful manner (Snyderman and Hunt, 1970).

A. Diatomic encoding process



B. Diatomic encoding flow chart

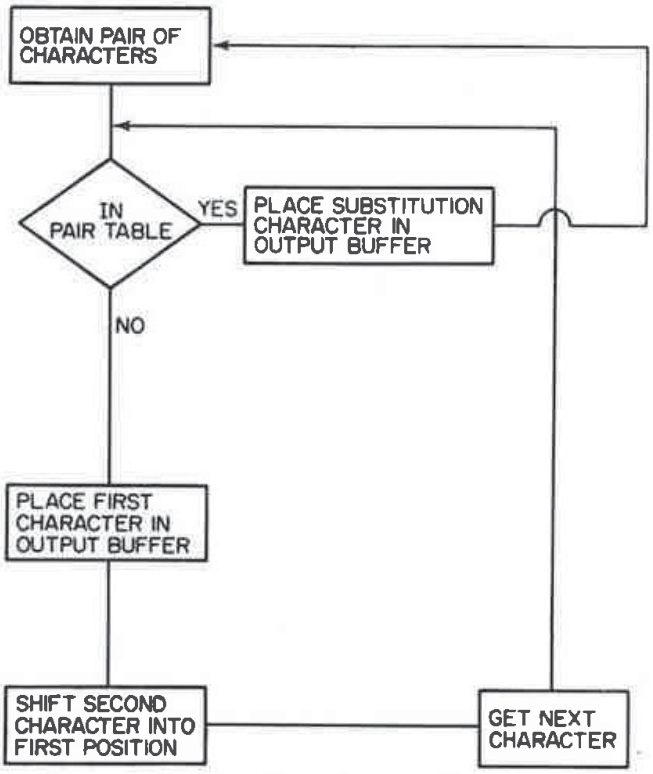


Figure 2.29 The diatomic encoding process

To assist readers in selecting the appropriate character pairs to replace with special characters, several tables of pair combinations are presented in this section. In Table 2.7, the reader will find a table containing the first 25 most frequently encountered pairs of characters in a 12 198 character English language text (Aronson, 1977). This table, prepared by Jewell, denotes the rank, pair combination, number of occurrences of the pair and the occurrences per thousand data characters (Jewell, 1976).

Since many users of data transmission will transfer program files in addition to textual data, an analysis of the paired character composition of BASIC, COBOL and FORTRAN programs is presented. The analysis of these programs was obtained by the execution of the DATANALYSIS program written by 4-Degree Consulting located in Macon, Georgia. This program performs a compression susceptibility analysis upon data files and the paired character analysis listed in Tables 2.8 to 2.11 is but one of several compression

Table 2.7 Jewell character combination pairing

Rank	Combination	Occurrences	Occurrences per thousand
1	E__	328	26.89
2	__T	292	23.94
3	TH	249	20.41
4	__A	244	20.00
5	S__	217	17.79
6	RE	200	16.40
7	IN	197	16.15
8	HE	183	15.00
9	ER	171	14.02
10	__I	156	12.79
11	__O	153	12.54
12	N__	152	12.46
13	ES	148	12.13
14	__B	141	11.56
15	ON	140	11.48
16	T__	137	11.23
17	TI	137	11.23
18	AN	133	10.90
19	D__	133	10.90
20	AT	119	9.76
21	TE	114	9.35
22	__C	113	9.26
23	__S	113	9.26
24	OR	112	9.18
25	R__	109	8.94

Note:—represents a space character

algorithms analysed by that software package. The listing of the software statements in the DATANALYSIS program will be found in Appendix B (p. 162).Its use will facilitate the selection of one or more compression algorithms based upon an analysis of the susceptibility of one's anticipated or actual data traffic to several compression algorithms.

Table 2.8 shows the paired character compression analysis results based upon an examination of a 9322 character BASIC program. In general, most BASIC language programs contain a high proportion of input messages and prompts as well as output headings. This structure makes the paired character consistency form a modified English text paired character consistency. Normally, the degree of deviation from normal English textual data pairs results from the ratio of computation statements to input/output statements in the program. In Table 2.8, note that '__P' 'NT' and 'RI' are the most commonly encountered pairs. All three pairs come from PRINT statements in the program with the pair '___P' resulting from a programmer using a space to precede each PRINT statement. Similarly, the BASIC language statement

Table 2.8 Paired character compression analysis, basic data file of 9322 characters

Pair/count		Pair/count		Pair/count		Pair/count		Pair/count		Pair/count	
__P	13	NT	13	RI	13	__T	12	__	11	__I	8
O__	7	HE	7	__B	6	__S	6	T__	5	N__	5
__A	5	F__	5	__E	4	AB	4	BU	4	EX	4
__L	4	IN	4	IO	4	NI	4	__N	4	__U	4
TO	4	TS	4	__F	3	R__	3	S__	3	ND	3
E__	3	__R	3	OF	3	UR	3	OU	3	__C	3
SE	3	ET	3	__O	3	__D	2	NP	2	NS	2
EL	2	AC	2	ON	2	IR	2	OT	2	IT	2
LI	2	RO	2	LL	2	AT	2	NG	2	UT	2
IM	1	AL	1	AN	1	__K	1	BO	1	IU	1
LA	1	LD	1	BR	1	M__	1	LO	1	LU	1
MB	1	CK	1	NE	1	CO	1	CT	1	NO	1
DO	1	ED	1	__X	1	NU	1	EN	1	OL	1
EQ	1	ER	1	OS	1	ES	1	Y__	1	PP	1
PS	1	PT	1	__Y	1	GH	1	__G	1	SO	1
SS	1	ST	1	TA	1	TE	1	TH	1	TI	1
HI	1	HT	1	TU	1	UG	1	UI	1	UL	1
UN	1	IA	1	VA	1	XX	1	YE	1	YI	1
ZE	1	**	0	**	0	**	0	**	0	**	0

Total combinations found: 288

Note:—represents a space character

of the form 'IF X:Y THEN' can be denoted by the frequently encountered pairs '____I', 'F____' and 'HE'.

In Table 2.9, the results of a similar analysis of a 20 465 character FORTRAN program is presented while Table 2.10 denotes the pairs encountered when a 54417 character COBOL program was analysed. In the FORTRAN program analysis, common pairs result from such frequently used statements as 'FORMAT', 'WRITE' and 'READ'. Similarly, common pairs encountered in the COBOL program are normally a result of the 'PICTURE IS' statement. Finally, Table 2.11 shows the results of an analysis of the merger of the individual BASIC, FORTRAN and COBOL programs into one entity. Here, the 230 paired characters represent 16 266 total combinations. Since the file contained a total of 84 204 data characters, diatomic compression of the 230 most frequently encountered pairs would result in a 19.3 per cent (16 266/84 204) data reduction. Note that the 12 most frequently encountered pairs represent a potential data reduction of 4388 characters or approximately 25 per cent of the theoretical reduction obtainable by diatomically encoding the 230 most frequently encountered pairs. From this, it is apparent that diatomic encoding can be effectively used in conjunction with other compression techniques by selecting only a portion of the most frequently expected pairs of characters for representation by special compression indicator characters.

Table 2.9 Paired character compression analysis, FORTRAN data file of 20 465 characters

Pair/count		Pair/count		Pair/count		Pair/count		Pair/count		Pair/count	
__I	167	__F	116	TE	106	UT	105	OR	99	OU	99
RI	96	__W	87	MA	86	__C	86	IN	86	TP	81
IR	69	HA	66	__S	61	O__	60	ER	60	C__	57
IT	51	HE	48	EN	46	RA	44	__D	42	E__	42
AL	40	RE	39	SI	38	IX	38	ON	37	__T	36
HS	35	HD	34	TO	34	SU	33	__R	32	T__	31
IM	30	__B	30	TA	30	HB	30	HF	30	HN	30
HC	30	HO	29	HR	29	HG	29	HU	29	HV	29
TI	29	HH	29	HL	29	AR	29	HJ	28	HT	28
__N	28	HM	28	HW	28	HX	28	HY	28	HZ	28
IA	28	CT	28	HI	28	IP	28	HP	28	HQ	28
HJ	28	L__	27	IO	26	__G	26	EQ	25	NY	25
__O	25	UB	25	IY	25	LE	24	__E	24	__P	23
AN	23	__	23	N__	23	ND	22	CO	22	SE	22
__A	22	Y__	21	AT	21	R__	20	S__	20	IS	20
RO	20	IC	20	NC	20	PR	20	ED	19	TR	18
G__	18	UR	18	ES	18	OT	17	ET	16	NG	16
NA	16	LY	16	TH	15	AC	15	PE	14	D__	14
PU	14	UM	14	NU	14	CH	14	BL	13	IV	13
LI	13	__J	13	FI	12	PF	12	GO	12	__K	12
OW	12	ST	12	__M	11	IL	11	SS	11	LA	11
AI	11	EP	11	NS	11	DA	11	EA	10	EC	10
TS	10	TY	10	FO	10	UE	10	F__	10	UN	10

Total combinations found: 4370

Note: __represents a space character

Communications hardware implementation

The use of a diatomic data-compression technique was implemented by Infotron Systems in combination with several other compression algorithms on their TL780 statistical multiplexer.

In a conventional time division multiplexer, data from each input channel is assigned to a slot on the high-speed multiplexed output line, regardless of whether or not the bandwidth is used. Since each input line is assigned a corresponding time slot, implementing compression on the high speed link will not increase any individual line efficiency. If compression is implemented on the low-speed line side, normally referenced as the channel side or level, the efficiency of only each low-speed compressed link will be increased, since each link is reserved a fixed slot on the high-speed side. This is illustrated in the upper portion of Figure 2.30.

In a statistical multiplexer, the bandwidth for a particular channel on the high-speed link is used only when the channel is transmitting data or control signals. Therefore, compression of one or more low-speed links permits the

Table 2.10 Paired character compression analysis, COBOL program containing 54 417 characters

Pair/count		Pair/count		Pair/count		Pair/count		Pair/count		Pair/count	
__P	542	__F	391	IC	342	AL	316	IN	309	RE	297
E__	286	__V	251	__X	243	__W	239	LE	235	R__	235
__T	231	IL	229	UE	229	TE	221	PG	212	__O	211
NT	204	M__	190	AR	186	RO	186	O__	182	UT	178
CN	164	__C	164	CT	156	LN	153	RI	152	__M	151
CH	149	__L	148	OV	144	CI	141	FO	139	TY	137
__B	136	OR	129	__S	129	__I	122	__A	118	TO	110
ER	109	G__	108	NG	106	RM	102	EF	101	CE	97
AD	91	__	91	VA	90	PA	88	ES	87	AC	84
NC	84	T__	82	F__	79	AN	79	TA	67	DV	66
FI	63	S__	61	WR	60	TI	59	ST	57	__E	55
Y__	54	PU	52	__R	50	BU	49	QU	48	OM	46
ON	44	OT	43	AT	43	OF	42	CO	40	RK	40
LS	36	CD	35	D__	35	NE	35	AS	34	OU	33
__Z	33	__G	33	ME	32	IV	30	RP	30	EN	30
ND	29	__U	28	BE	27	OP	27	L__	25	H__	24
EL	24	TP	23	SE	23	CA	22	__H	22	__D	22
TH	22	DD	21	TR	20	ET	20	VE	20	VI	20
EC	20	YI	20	__N	19	GS	19	IT	19	C__	19
GE	18	WA	18	UA	18	SH	18	__K	17	IM	17
RA	17	RS	17	MO	16	DE	16	GI	15	EX	15
ED	15	MP	15	CC	15	WO	15	EQ	15	UN	15
LI	14	IO	14	__Q	14	UR	14	DI	13	FL	13

Total combinations found: 12 509

Note: __represents a space character

statistical multiplexer to utilize less of the bandwidth of the high-speed line for the low-speed link being compressed. The compression of the low-speed link at the channel side will then result in a lower, high-speed line rate or permit more low-speed channels to be added since compression reduces the total number of data characters transmitted over the high-speed line. Conversely, if compression is performed at the high-speed line level, the number of characters transmitted on that link will be reduced. This will permit a lower composite high-speed operating data rate or permit additional low-speed channels to be added. While some vendors have elected to compress the high-speed link, Infotron uses a diatomic encoding process combined with additional data-compression techniques on their low-speed channel adapters to perform compression at the channel level. This technique permits the user to select which channels, if any, should be compressed.

In the Infotron technique, statistical multiplexer compression occurs through the use of multiple-space codes, repeated character codes, common character pair codes (diatomic encoding) and packed decimal codes (half-

Table 2.11 Paired character compression analysis, combined 84 204 character file

Pair/count		Pair/count		Pair/count		Pair/count		Pair/count		Pair/count	
__P	578	__F	510	IN	399	IC	362	AL	357	RE	336
E__	331	TE	328	__W	326	__I	297	UT	285	__T	279
RI	261	LE	259	R__	258	__V	255	__C	253	__X	252
O__	249	NY	242	IL	240	UE	239	__O	239	OR	231
AR	215	PG	212	RO	208	__S	196	M__	194	CT	185
__B	172	ER	170	CN	164	CH	163	__M	162	__L	159
LN	153	FO	149	TO	148	TY	147	__A	145	OV	144
CI	141	OU	135	G__	126	___	125	NG	124	T__	118
CE	107	ES	106	RM	105	TP	104	NC	104	AN	103
EF	102	AC	101	PA	98	TA	98	MA	94	F__	94
AD	91	VA	91	TI	89	__R	85	S__	84	__E	83
ON	83	EN	77	HA	77	C__	76	Y__	76	FI	75
IT	72	IR	71	ST	70	DV	66	PU	66	AT	66
__D	66	CO	63	OT	62	HE	62	RA	61	__G	60
WR	60	QU	56	ND	54	OF	54	BU	54	OM	53
L__	52	__N	51	D__	49	SE	48	IM	48	IO	44
IV	43	SI	41	NE	41	EQ	41	RK	40	N__	40
ET	39	IX	38	TH	38	TR	38	ME	38	HD	37
AS	37	LS	36	HR	36	UB	35	HS	35	CD	35
__U	35	ED	35	__Z	34	SU	33	HO	33	HP	32
HY	32	UR	32	IA	31	IS	31	HI	31	OP	31
HF	30	HB	30	HN	30	PR	30	HC	30	RP	30
__K	30	EC	30	LI	29	HV	29	HH	29	BE	29

Total combinations found: 16 266

Note: ___ represents a space character

byte encoding). In addition, since data must be queued at the channel level to compress it, it becomes necessary to transmit control signals through the data path of the high-speed link to preserve the time relationship between data and control signals. The addition of these signals reduces the overall compression efficiency. Since each channel adapter on the multiplexer requires a buffer area and a microprocessor to effect compression, compression of a large number of low-speed channels becomes more expensive from a hardware standpoint than compressing data at the high-speed line level where only one buffer area and a single microprocessor are required.

The Infotron channel adapter that performs compression only operates on asynchronous ASCII coded data. To obtain a sufficient number of special compression indication codes, the parity bit in the normal 8-bit ASCII code is stripped for transmission. This results in 128 character codes that can be used to represent and indicate compressed information. The stripping of parity by the microprocessor within the multiplexer has no effect on errors since the multiplexer employs an HDLC-like frame transmission on the high-speed link level to include generating a cyclic redundancy check of transmitted frames.

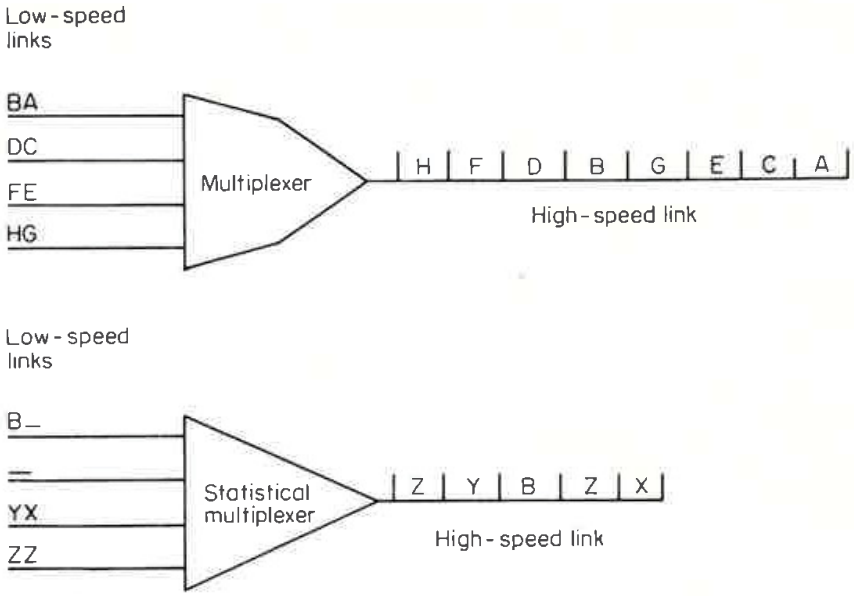


Figure 2.30 Multiplexing and compression. If compression occurs on one or more low-speed links, the effective information transfer ratio of those individual links will increase when a conventional TDM is employed. When statistical multiplexers are employed, data may be compressed at the individual channel level or overall at the high-speed line level

In the Infotron system, codes are assigned to represent groups of 2 to 7 consecutive spaces for various multiple-space compression code schemes. These codes are most effective when transmitted data has been formatted in columns separated by groups of spaces or for textural information that contains paragraph indentations and margin justification through the use of spaces.

To represent repeated characters, 16 codes were assigned to represent groups of 3 to 18 consecutive identical characters. This code is followed by the character to be repeated, in a similar way to run-length encoding, and results in a 2-byte code. To represent common character pairs, 48 codes have been assigned. The characters pairs used by Infotron are listed in Table 2.12. With the exception of the decimal point space and carriage return line

Table 2.12 Common character pair codes compressed by Infotron: both upper and lower case

S__	__T	IN	TE	AN
T__	__A	ED	ER	TI
E__	__N	AT	RE	ON
R__	__O	ES	TH	CRLF
D__	__I	SE	HE	
—				

Note: __ represents a space character. CRLF denotes carriage return followed by line feed

feed pairs, all other pairs include both upper and lower case characters.

Lastly, 16 codes are assigned to specify when 4 to 19 characters are in packed decimal (half-byte) format. Here, characters are represented by 4-bit codes packed 2 per 8-bit byte. In addition to numerics, the dollar sign, period, comma, per cent and diagonal sign and space are stripped of the leading 4 bits if they occur in the string and are included in the packed format.

Although the effectiveness of the compression technique employed obviously depends upon the data to which the technique is applied, using multiple techniques increases the possibility of being able to use one technique effectively upon a portion of the data stream. During channel adapter compression tests, a compression ratio of up to 1.8 was noted by Infotron, indicating that only 55 per cent of the input data stream was actually transmitted.

Programming examples

The BASIC program PAIRC.BAS listed in Figure 2.31 was developed to perform diatomic compression based upon the Jewell character combination

```
10 REM PAIRC.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO SUBROUTINES FOR COMPRESSION
80 '*******************************************
90 PRINT "ENTER ASCII FILENAME. EG, PAIR.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "PAIRC.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
115 GOSUB 400                    'PAUSE TO SET UP TABLE
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****DIATOMIC COMPRESSION SUBROUTINE*******
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND COMPRESSES OUT COMMON PAIRS         *
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*******************************************
230 I=1                          'RESET INDICES
240 FOR J= 1 TO N-1              'STEP THRU RECORD
250 A$= MID$(X$,J,2)             'EXTRACT A PAIR
```

Figure 2.31 PAIRC.BAS program listing

```
260 FOR K = 1 TO 25                 'SETUP PAIR TABLE LOOP
270 IF A$=P$(K) THEN GOSUB 350      'IS INPUT PAIR IN TABLE?
280 NEXT K                          'NO - TRY NEXT
290 IF M = 1 THEN 310               'IF MATCH FLAG SET?
300 O$(I) = MID$(A$,1,1)            'NO-STUFF 1ST CHAR IN BUFFER
310 I=I+1                           'BUMP INPUT STRING INDEX
320 M=0                             'RESET MATCH FLAG
330 NEXT J                          'GO BACK FOR MORE
340 RETURN                          'DONE
350 M=1                             'SET PAIR MATCH FLAG
355 '*************************************************************
360 'INSERT COMPRESSION NOTATION IN OUTPUT BUFFER
365 V = K + 224                     'INDEX OUT TO SUBSTITUTE CHAR
370 O$(I)=CHR$(V)                   'INSERT PAIR SUBSTITUTION
380 J=J+1                           'FORCE INPUT SHIFT 2 OVER PAIR
390 K = 25                          'FORCE END OF PAIR SEARCH
395 RETURN                          'GO BACK FOR MORE
400 DIM P$(25)                      'JEWELL CHAR. COMBINATION PAIRS
410 DATA "E "," T",TH," A","S ",RE,IN,HE,ER," I"," O","N ",ES,
420 DATA " B",ON,"T ",TI,AN,"D ",AT,TE," C"," S",OR,"R "
425 FOR I = 1 TO 25                 'SETUP PAIR TABLE
430 READ Z$                         'GET COMMON PAIR
440 P$(I) = Z$: NEXT I              'AND STUFF INTO PAIR TABLE
450 RETURN                          'DONE - TABLE COMPLETE
900 '*****TALLY THE COMPRESSION COUNT & WRITE BUFFER******
910 '* DISPLAY BEFORE & AFTER RESULTS OF COMPRESSION     *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD  *
930 '*************************************************************
931 N1=N1+N                         'TALLY INPUT CHAR COUNT
932 T=N-I+1                         'NET DIFFERENCE IN BUFFERS
936 T1=T1+T                         'SAVE COUNT FOR SUMMARY
940 FOR I=1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
1000 PRINT
1010 PRINT "**RUN-LENGTH ENCODING SAVED ";T;" CHARACTERS"
1020 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE COMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "PAIRC.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER COMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT T1;" TOTAL CHARACTERS ELIMINATED FROM ";
9999 PRINT N1;"OR ";INT((T1/N1)*100);"%":CLOSE:END
```

Figure 2.31 (continued)

pairing previously listed in Table 2.7. Although this example of diatomic compression was programmed to use the Jewell character combination pairing, it is easily modified to compress data based upon the use of other character pairs that may more appropriately reflect the reader's data.

Similar to other compression routines previously presented in this chapter, the diatomic compression program was developed using subroutines linked together to provide distinct code modules that can be easily analysed by the reader. After the data file is opened in line 100, the subroutine commencing at line 400 is invoked. This subroutine initializes the P$ array elements to the Jewell character combination pairing, resulting in 25 character pairs assigned to the array P$. The reader can change the data pairs contained in lines 410 and 420 of the subroutine, however, if the number of data pairs is changed from 25, the appropriate indices in the program must be changed to reflect the actual number of pairs. In addition, the dimension size of the P$ array must be changed to reflect the new number of pairs to be used in the diatomic compression routine. Thus, lines 400 and 425 would require modification in the subroutine previously discussed when a new set of character pairs are entered in lines 410 and 420 whose sum differs from 25.

After a line of data is read in line 130, its length is determined in line 140. The subroutine invoked in line 150 processes the line of data read from the file commencing in line 230. After the indices are reset in line 230, the FOR-NEXT loop bounded by lines 240 to 330 steps through the record, extracting pairs of data in line 250. The inner FOR-NEXT loop bounded by lines 260 and 280 compares the pair extracted from the record in line 250 to the pairs contained in the pair table previously set up by the subroutine in line 400. The reader should note that the outer limit of 25 in line 260 should also be changed if the number of pairs used in the program changes from that value.

If a pair of characters extracted from the record matches a pair in the pair table, the subroutine in line 350 is invoked. Line 350 uses the variable M to denote that a match occurred. In line 365, the variable *V* is set to the sum of the variable *K* plus 224. Here the value of *K* is the position in the pair table where the pair extracted from the record matched a predetermined pair. The reason 224 was added to this value was for clarity of display of the results of this compression routine. That is, italics are printed from ASCII 225 upward on many printers including one printer used by the author. Thus, the pair 'E space' is represented by an italic 'a' when printed, and so on.

The pair substitution character is inserted into the appropriate element of the 0$ array as indicated in line 370. Note that J is incremented by 1 in line 380 to force a shift over the current position in the input record. Next, line 390 sets *K* to 25 to terminate the pair comparison in the FOR-NEXT loop bounded by lines 260 and 280, from which the compression routine was called and to which it returns upon execution of line 395.

Since the variable *M* was set to 1 to indicate a pair match occurred, the termination of the FOR K loop causes the execution of line 290 to result in

```
ENTER ASCII FILENAME. EG, PAIR.DAT
? PAIR.DAT
PATIENCE - INPUT PROCESSING
FILE PAIR.DAT BEFORE COMPRESSION:
1 TO BE OR NOT TO BE THAT IS THE QUESTION
2 THE RAIN IN SPAIN FALLS MAINLY IN THE PLAIN
FILE PAIR.DAT AFTER COMPRESSION:
1Γ∩ß· NO±T∩ßπJΩσπßQUØ≥≡
2Γ§ RAτΩ∞SPAτ FALLσMAτLYΩ∞πßPLAτ
 30  TOTAL CHARACTERS ELIMINATED FROM  91 OR  32 %
Ok


ENTER ASCII FILENAME. EG, PAIR.DAT
? PAIR.DAT
PATIENCE - INPUT PROCESSING
FILE PAIR.DAT BEFORE COMPRESSION:
1 TO BE OR NOT TO BE THAT IS THE QUESTION
2 THE RAIN IN SPAIN FALLS MAINLY IN THE PLAIN
FILE PAIR.DAT AFTER COMPRESSION:
1Γ∩ß· NO±T∩ßπJΩσπßQUØ≥≡
2Γ§ RAτΩ∞SPAτ FALLσMAτLYΩ∞πßPLAτ
 29  TOTAL CHARACTERS ELIMINATED FROM  86 OR  33 %
Ok
```

Figure 2.32 Sample execution of PAIRC.BAS program as displayed on a monitor

a branch to line 310. Here the index used for the 0$ array is increased by one and the match flag is reset to zero prior to the loop terminating.

Figure 2.32 illustrates how the execution of the diatomic compression routine will appear on one's monitor while Figure 2.33 illustrates the screen image after it has been 'dumped' to a printer that outputs ASCII values from 225 upward as italics. Thus, some readers may prefer to use the execution illustrated in Figure 2.33 to compare the compression characters in italics with respect to the original data and the Jewell character combination pairs used in the program. Since an italic lower case 'a' represents the first combination pair while an italic 'b' represents the second pair and

```
ENTER ASCII FILENAME. EG, PAIR.DAT
? PAIR.DAT
PATIENCE - INPUT PROCESSING
FILE PAIR.DAT BEFORE COMPRESSION:
1 TO BE OR NOT TO BE THAT IS THE QUESTION
2 THE RAIN IN SPAIN FALLS MAINLY IN THE PLAIN
FILE PAIR.DAT AFTER COMPRESSION:
1bOoay NOqTOo∞cujec∞QUmrp
2bh RAgjlSPAg FALLeMAgLYjlc∞PLAg
 30  TOTAL CHARACTERS ELIMINATED FROM  91 OR  32 %
```

Figure 2.33 Sample execution of PAIRC.BAS program when printed using a printer that displays characters greater than ASCII 224 as italics

so on, it should be easier to use the second example of the PAIRC.BAS program execution for readers who wish to follow the logical flow of the program in detail.

Decompression

The program listing of PAIRD.BAS is listed in Figure 2.34. As indicated by the naming conventions used in this book, this program performs decompression upon previously compressed pairs of characters.

From an examination of the program coding listed in Figure 2.34, the reader will note that the construction of the code modules for decompression closely resemble the previously examined compression program. Although our programming goal was to do this to facilitate a comparison between programs, due to the relationship between compression and decompression such modular coding relationships will normally be the rule and not the exception.

After opening files for input and output, the subroutine beginning at line 500 is invoked by line 115 of the program. This subroutine simply builds the P$ table that will contain the Jewell character combination pairs that the program will search for. In line 130, the familiar LINE INPUT statement is used to obtain a record from the input file. Next, line 140 is employed to determine the length of the record while line 150 invokes the subroutine beginning at line 180 which performs the actual decompression of data.

The FOR-NEXT loop bounded by lines 240 and 310 searches through the record previously extracted from the input file on a character by character

```
10 REM PAIRD.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO DECOMPRESSION SUBROUTINE  *
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, PAIRC.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "PAIRD.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
115 GOSUB 500
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
```

Figure 2.34 PAIRD.BAS program listing

```
180 '*****DIATOMIC   DECODING SUBROUTINE********
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$   *
200 '* AND DECOMPRESSES PAIR-ENCODED CHARACTERS*
210 '* USING O$ AS THE OUTPUT BUFFER.           *
220 '*********************************************
230 K=1:J=1:V=0                     'RESET INDICES
240 FOR I= 1 TO N                   'STEP THRU RECORD
250 A$= MID$(X$,I,1)                'EXTRACT A CHAR
260 IF A$> CHR$(224) THEN 360       'COMPRESSED PAIR?
290 O$(J)=A$                        'STUFF IN OUTPUT BUFFER
300 J=J+1                           'BUMP BUFFER INDEX
310 NEXT I                          'GO BACK FOR MORE
320 RETURN                          'END OF STRING
355 '*******************************************************
360 'DECODE COMPRESSION NOTATION TO OUTPUT BUFFER
365 '*******************************************************
370 K= ASC(A$)                      'GET ORDINAL EQUIV.
380 K= K-224                        'SUBTRACT FOR INDEX
390 O$(J)= P$(K)                    'STUFF PAIR IN BUFFER
400 J= J+1                          'BUMP OUTPUT INDEX
405 V= V+1                          'SUM VARIABLE COUNT
410 GOTO 310                        'DONE
500 DIM P$(25)                      'JEWELL CHAR. COMBINATION PAIRS
510 DATA "E "," T",TH," A","S ",RE,IN,HE,ER," I"," O","N ",ES,
520 DATA " B",ON,"T ",TI,AN,"D ",AT,TE," C"," S",OR,"R "
530 FOR I = 1 TO 25                 'SET UP PAIR TABLE
540 READ Z$                         'GET COMMON PAIR
550 P$(I) = Z$: NEXT I              'AND STUFF INTO PAIR TABLE
560 RETURN                          'DONE - TABLE COMPLETE
900 '*****TALLY THE DECOMPRESSION COUNT & WRITE BUFFER****
910 '* DISPLAY BEFORE & AFTER RESULTS OF DECOMPRESSION   *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD  *
930 '*******************************************************
931 N1=N1+N                         'TALLY INPUT CHAR COUNT
932 T=N-J+1+V                       'NET DIFFERENCE IN BUFFERS
936 T1=T1-T                         'SAVE COUNT FOR SUMMARY
940 FOR I= 1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
970 RETURN
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE DECOMPRESSION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "PAIRD.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER DECOMPRESSION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$:PRINT ABS(T1);" TOTAL CHARACTERS INSERTED"
9999 CLOSE:END
```

Figure 2.34 (continued)

```
ENTER ASCII FILENAME. EG, PAIRC.DAT
? PAIRC.DAT
PATIENCE - INPUT PROCESSING
FILE PAIRC.DAT BEFORE DECOMPRESSION:
1ΓΟΠß· NO±TOΠßπJΩσπßQUØ≥≡
2Γ§ RAτΩ∞SPAτ FALLσMAτLYΩ∞πßPLAτ
FILE PAIRC.DAT AFTER DECOMPRESSION:
1 TO BE OR NOT TO BE THAT IS THE QUESTION
2 THE RAIN IN SPAIN FALLS MAINLY IN THE PLAIN
 29  TOTAL CHARACTERS INSERTED
Ok
```

Figure 2.35 Sample execution of PAIRD.BAS program

basis. This is accomplished by the use of the MID$ function in line 250. If the character extracted from the record exceeds a value of 224, it is assumed that diatomic or paired compression has occurred. This assumption is based upon the selection of each character beyond ASCII 224 to represent a pair of characters in this coding example. If the character extracted from the record equals or is less than ASCII 224, that character does not represent a previously compressed pair of characters. Thus, line 290 simply places the extracted character into its appropriate position in the output buffer.

When an ASCII character greater than 224 is encountered, the branch to line 360 in the program results in the actual decompression of a previously compressed pair of characters. In line 370, the numerical value of the character that actually represents a pair of characters is obtained. Next, line 380 subtracts 224 from the numerical value of the character to obtain the appropriate index in the paired table (P$(25)). Line 390 places the pair of characters that was previously represented by one character into the output buffer while lines 400 and 405 increment the index position in the output buffer and the variable *V* which is only employed to compute the difference in size between the input and output buffers and is not required for decompression.

Figure 2.35 illustrates the execution of the PAIRD.BAS program as it would appear on our monitor using the data file PAIRC.DAT as input. PAIRC.DAT was created by the PAIRC.BAS program. Thus, it is of no surprise that the two compressed lines of data at the top of Figure 2.35 match lines 1 and 2 in the lower part of Figure 2.32, while lines 1 and 2 at the bottom of Figure 2.35 match those lines at the top of Figure 2.32.

2.6 PATTERN SUBSTITUTION

This compression technique is basically a sophisticated form of diatomic encoding. Here, a special character code is substituted for a predefined character pattern. The employment of the pattern substitution compression technique can be highly advantageous when one is transmitting program listings and other types of data files containing known repeating patterns.

The advantage offered by pattern substitution is best understood by examining a higher-level language such as FORTRAN. In any FORTRAN program, a very high probability exists that one or more types of statements will be encountered containing common key words such as 'READ', 'WRITE' and 'FORMAT', among others. Instead of transmitting the characters of these key words on a character by character basis each time they appear, one of the unassigned characters from the employed character set can be substituted. When pattern substitution is applied to language text, common key words or phrases can similarly be replaced. For English text transmission, such commonly encountered words as 'and', 'the', 'that' and 'this' would be among the first candidates for substitution.

The pattern table

To employ pattern substitution, a pattern table is required. This table contains a set of list arguments and a set of function values. Each function value is a special compression indicator character which represents the compressed value of a particular argument (Aronson, 1977). Figure 2.36 shows an example of the use of a pattern table. Although each list argument was of similar length, this table can be expanded to include many additional entries of various character length. Strings of 4, 5, 6 and more blanks, for example, could be assigned values represented by different special characters as well as patterns of alphanumeric data.

Encoding process

To obtain the compressed data stream, the source data must be broken down into distinct search arguments, initially equal to the smallest sized argument in the pattern table. The search argument is matched with those list argu-

NOW IS THE TIME FOR ALL GOOD MEN

Pattern table

List arguments	Function values
THE	S_{c_1}
FOR	S_{c_2}
ALL	S_{c_3}

Compressed data streem

NOW IS S_{c_1} TIME S_{c_2} S_{c_3} GOOD MEN

Figure 2.36 Pattern table utilization. Upon a portion of the original data stream matching the list argument, the appropriate function value is substituted. In the above example, special compression indicator characters S_{c1} to S_{c3} are substituted for the words 'the', 'for' and 'all' as they are encountered

ments of equal character width. If a match is obtained, the function value associated with the list argument then replaces that portion of the original data stream and results in data compression. If no match is obtained, the width of the search argument is increased to the width of the next larger list argument or series of list arguments and the process is repeated. If after increasing the width of the search argument to the largest width of the list argument no match results, the first character of the original data string is passed to the compressed data string and the process is repeated, starting with the second character from the original data stream.

A second method of performing pattern substitution results from the use of blanks as delimiters. The binary or octal value of the characters between blanks can be generated and compared with the binary or octal values in the list argument portion of the pattern table. This process simplifies the searching of a long argument list and minimizes the processing time required to encode patterns.

Patterns in programming languages

Due to the utilization of keywords or reserved words in most programming languages, pattern substitution is often a very effective compression technique for storing or transmitting program files. Since the number of keywords or reserved words in a programming language can be as high as several hundred, a 2-byte sequence can be employed to represent each keyword pattern substitution. Here, the first byte or character would be used to indicate pattern substitution has occurred, while the following character would denote the actual pattern that was substituted for the keyword or reserved word. To illustrate this concept in additional detail, let us assume that the version of BASIC we are working with is limited to eight keywords. Table 2.13 lists these keywords and their equivalent function values contained in the pattern table that could be constructed.

For clarity of explanation the dollar sign ($) was employed as the compression indicating character in Table 2.13, although obviously any character

Table 2.13 BASIC language pattern table

Keywords	Function values
END	$1
GOTO	$2
IF	$3
INPUT	$4
LET	$5
PRINT	$6
REM	$7
THEN	$8

BASIC program	*Compressed program*
100 REM COMMISSION CALCULATION	100$7COMMISSION CALCULATION
110 PRINT "ENTER SALE PRICE"	110$6"ENTER SALE PRICE"
120 INPUT W	120$4W
130 PRINT "ENTER NUMBER SOLD"	130$6"ENTER NUMBER SOLD"
140 INPUT N	140$4N
150 LET C=W*N*.0875	150$5C=W*N*.0875
160 PRINT "COMMISSION=";C	160$6"COMMISSION=";C
170 PRINT "ANOTHER CALCULATION-Y/N"	170$6"ANOTHER CALCULATION-Y/N"
180 INPUT A$	180$4A$$
190 IF A$ <> "Y" THEN 210	190$3A$$<>"Y"$8210
200 GOTO 110	200$2110
210 END	210$1

Figure 2.37 Compressing a BASIC program

in the character set could be used. Preferably, one should select a character which is seldom or, better yet, never used. Since there is always the possibility that the character could occur in a BASIC program, one can replace each occurrence of the pattern compression indicating character by duplicating that character when it is encountered. Then, the decompression routine would disgard the second occurrence of a pattern compression indicating character followed by itself. The compression of a short BASIC program is illustrated in Figure 2.37 based upon the employment of pattern substitution, which in actuality is the replacement of BASIC keywords. Note that the pattern table contained in Table 2.13 was used for the compression process. Since most BASIC languages require keywords to be delimited by spaces, we have assumed that the keywords entered in Table 2.13 contained leading and trailing blanks, enabling the functional value substituted for the keyword to be a more effective substitution. Using this method of substitution, 25 spaces as well as 26 other characters are eliminated from the program while 2 characters are added. The additional characters are due to the replacement of the natural occurrence of the $ character in the program by the special sequence $$ in lines 180 and 190.

Although the overall data reduction, which in this example was approximately 20 per cent, may not appear significant, it should be noted that the actual effort involved to compress data using pattern substitution may not be significantly demanding. To increase the data reduction resulting from compression usually requires the application of several compression techniques to one's data. In this particular example, one might first preprocess programming files through the utilization of pattern substitution compression. Then one could statistically encode the resulting compressed data. Since the statistical encoding process results in the replacement of frequently occurring characters by short bit sequences, statistically encoding data where keywords

were previously replaced by short patterns is more effective than the statistical encoding of the original data. As an example, a 5-bit sequence might be required to represent the keyword PRINT, however, a short bit sequence would be required to represent the character sequence $6 that was substituted for the keyword. The reader is referred to Section 2.9 for additional information concerning statistical encoding.

2.7 RELATIVE ENCODING

Relative encoding is a compression technique that is not normally applicable to the transmission of conventional data files. This type of compression is effectively employed when there are sequences of runs in the original data stream that vary only slightly from each other or the run sequences can be broken into patterns relative to each other. An example of the former is telemetry data while the bit patterns of digital facsimile machines represent a version of the latter.

Telemetry compression

In telemetry data generation, a sensing device is used to record measurements at predefined intervals. These measurements are then transmitted to a central location for additional processing. One example of telemetry signals is the numerous space probes which transmit temperature readings, colour spectrum analysis and other data, either upon command from earth stations or at predefined time intervals. Normally, telemetry signals contain a sequence of numeric fields consisting of subsequences or runs of numerics that vary only slightly from each other as illustrated in the top portion of Figure 2.38.

Prior to actual data transmission, compression occurs to reduce the total amount of data necessary to represent the recorded measurements. Each measurement other than the first is coded with the relative difference between it and the preceding measurement, as long as the absolute value of the increment is less than some predetermined value. This is shown in the lower portion of Figure 2.38. If the increment should exceed this value, a special character is inserted to denote that the particular value at that location is not available or the special character could be followed by the measurement that is out of the boundary range for the relative encoding process. This

Original telemetry measurements

46 46 46.1 46.1 46.1 46 46 46 46.1 46.1 46.1 46.2

Relative encoding

46 0 .1 0 0 —.1 0 0 .1 0 0 .1

Figure 2.38 Relative encoding process. Telemetry signals often consist of a sequence of numerics that vary only slightly from each other during a certain time interval

limits wide fluctuations and is one disadvantage associated with the utilization of this technique. Another disadvantage is that if data values consistently vary both within and outside the relative encoding boundary range and a combination of a special character and actual value is transmitted, this will cause an expansion instead of a compression of the data stream.

Additional techniques may be employed to obtain a higher degree of compression depending upon the original telemetry measurements and the resultant data due to the relative encoding process. In the top portion of Figure 2.38, the original telemetry measurements illustrated consist of 38 characters to include numerics and decimal points. As a result of the relative encoding process, the number of numerics and decimal point characters has been reduced to 18. By the incorporation of a second compression technique, the number of characters used to represent the relative encoding process may be further reduced. One method that could be used is the half-byte packing process where each numeric digit is stripped of its first 4 bits and packed 2 per character. If we use a 4-bit representation for the decimal point and minus sign, half-byte packing will result in the transmission of nine 8-bit bytes of data. Thus, while the relative encoding process resulted in a 2.24 (38/17) compression ratio, recompressing the relative encoding results employing the half-byte packing technique approximately doubles the compression ratio to 4.223 (38/9).

While the half-byte packing process was illustrated as the combining or second compression technique, other techniques may be employed with results dependent upon the variability of the original telemetry measurements. If the original telemetry measurements indicated a stable 46 for the time interval sampled, the relative encoding process would result in a long string of zeros after the value indicator of 46. For this situation, run-length encoding would be more effective as the second compression technique.

Digital facsimile

Several relative encoding techniques can be employed to compress digital facsimile data. Prior to discussing these techniques, a review of the elements of facsimile technology is warranted.

Facsimile systems use the basic concept of scanning—normally on a line-by-line basis—to create a stream of information concerning the lightness or darkness of the small area being scanned at any given point in time. The resulting stream of information is then transmitted and used to drive an image-reproducing device at a facsimile receiver where the original information is reproduced. In general, the operation of a facsimile device is quite similar to the technology employed in television, where 525 lines on the US domestic television system are used to reproduce images. For facsimile systems, the clarity depends upon the fineness of the scan. Normally, approximately 100 scan lines per inch are required to successfully reproduce a page of typewritten material. Thus, a normal $8\frac{1}{2} \times 11$ sheet of paper, scanned

longitudinally, would require approximately 850 scan lines. Each scan line in turn consists of approximately 1730 picture elements (pixels, or pels), resulting in approximately 1 million bits for an $8\frac{1}{2} \times 11$ sheet of paper. To transmit this data at 4800 bps without compression, 209 s or approximately 3.5 min are required.

For facsimile systems, the degree of compression theoretically obtainable is normally very large for the typical facsimile message. As an example, consider a typewritten memorandum containing 500 characters. In conventional data transmission, each character can be represented and transmitted by 8 bits. Thus, the entire message could be transmitted, ignoring control characters, by 500×8 or 4000 bits of information. If transmitted at 4800 bps, the total transmission time would be less than 1 s. In comparison, the same message sent by conventional facsimile requires the transmission of almost 1 million bits and takes about $3\frac{1}{2}$ min without data compression, a difference of approximately 270 to 1 between conventional facsimile code and character transmission.

Facsimile techniques

One of the earliest facsimile compression techniques was run-length encoding. Here, the transmission of the digital line scan is replaced by the transmission of a quantity count of each of the successive runs of black or white scanned pels.

Since the vast majority of documents to be scanned contains a much higher quantity of white pels than black ones, transmitting the difference between scans may significantly reduce the quantity of data to be transmitted. In this method of compression, one complete scan is held in a memory area of the device and compared with the subsequent scan. Transmitting only changes relative to the preceding scan results in the relative compression process. Once the differences between the first and second scans are transmitted, the first scan is removed from memory and replaced by the second scan. Next, a third scan is compared with the second scan now located in memory. A flow chart showing the required steps for this type of relative encoding process is illustrated in Figure 2.39.

In Figure 2.40, a portion of the relative changes resulting from the comparison of two scan lines is shown. Several methods can be used to denote the relative changes between the Nth and $(N + 1)$th scan lines. One method is to denote the position of the change by what is normally called a positional identification. Here, the position of each relative change is denoted with respect to the first pel of the line. If there are many consecutive changes, the transmission of each individual position could require more data bits than the transmission of the original line prior to comparison with the preceding line. To take advantage of successive relative changes between scanned lines, the position indicator can be followed by a quantity count which contains the number of successive relative changes. This is illustrated

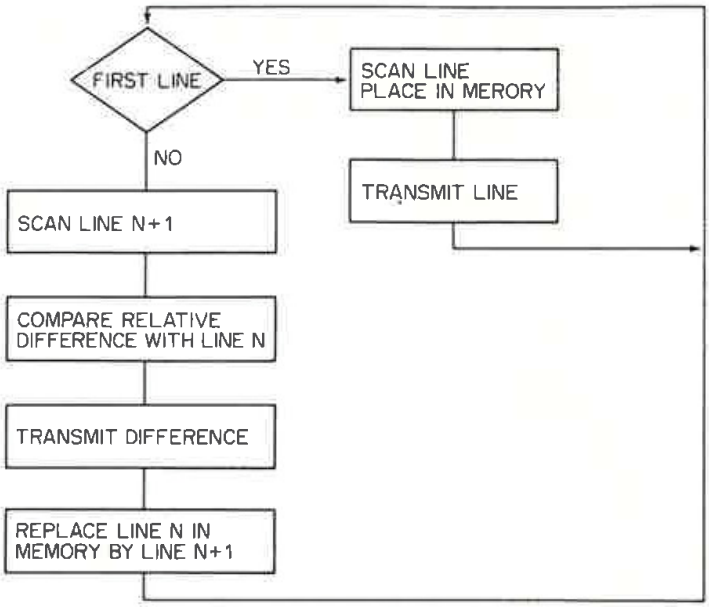


Figure 2.39 Facsimile relative encoding process

in Figure 2.41 where the table at the top of the figure tabulates the initial position of the relative change between line scans and the number of succeeding relative changes, while the transmission sequence is indicated at the lower portion of that figure. Under the Consultative Committee for International Telephone and Telegraph (CCITT) digital facsimile standards, there are 1728 picture elements or points to be read by the scanner along the width of a document 215 mm wide. Due to the large number of positions, the transmission of positional information can rapidly increase in duration, especially when a number of relative changes occur at the far end of the scan line. One method used to alleviate this 'end of the line' increase is by the use of displacement notation. As with positional notation, the relative changes between scan lines are first computed. Then, instead of transmitting all of the initial positions of the relative changes and the number of successive

N-th scan line

. . . 0 0 0 0 0 0 1 1 1 0 0 0 1 1 0 . .

(N + 1) th scan line

. . . 0 0 0 0 0 1 1 1 1 0 0 1 1 0 0 . .

Relative change

. . . – – – – – – X – – – – – X – X – – – – . . .

Figure 2.40 Relative change. To denote the relative changes between scan lines several methods can be employed to include identification by position and displacement

Initial position of relative change	Number of successive relative changes
40	6
80	20
175	4
350	31
480	8
930	14
1250	16
1310	5
1340	4

Transmitted data

40	6	80	20	175	4	350	31	480	8	930	14	1250	16	1310	5	1340

Figure 2.41 Transmitting positional information. Using the positional relative process, the initial position of each relative change is followed by the number of successive relative changes

changes as illustrated in Figure 2.41, only the first initial position is transmitted. Thereafter, the displacement between relative changes is transmitted. This displacement method can include the transmission of successive relative change information and is illustrated in Figure 2.42. This figure is based upon the data provided in the tabular portion of Figure 2.41. In comparing the illustrated positional and displacement methods, the positional method requires 41 numeric characters while the displacement method can be accomplished by the use of 35 such characters. If numerics are packed 2 per byte, then the displacement technique will result in 140 bits being required to represent the 1728 points in the example while the positional method would require 164 bits.

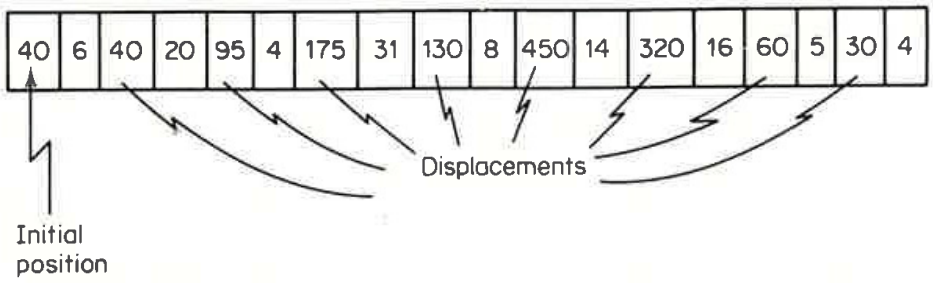


Figure 2.42 Transmitting displacement information. Another relative encoding technique results in the transmission of displacement information

2.8 FORMS MODE OPERATION

Forms mode operation is a method of compression that can be employed when data is to be communicated to and from a CRT display in a predefined series of formats. When operated in the forms mode, the display can be used for a fill in the blank type of operation. In this mode of operation, two basic types of data are displayed—protected information and variable information. Fixed or protected information corresponds to the preprinted information of a data field in a standard printed form such as name, address, social security number and similar types of information. Such information when the operation is in the forms mode is not cleared when the screen is erased, is not transmitted to the central processor and is not alterable by accidental keyboard entries. Each fixed field is one-half of a field pair, the other half being the corresponding variable field. Thus, in the forms mode the fixed field can be viewed as the question while the information entered into the corresponding variable field can be considered as the answer.

An example of forms mode data entry is illustrated in Figure 2.43. Here, the blank spaces indicate the additional positions available for data entry into the variable fields.

In using the forms mode of operation, the operator denotes the form he or she wishes to complete and that form is transmitted from the computer to the terminal display or is locally generated from terminal memory or from

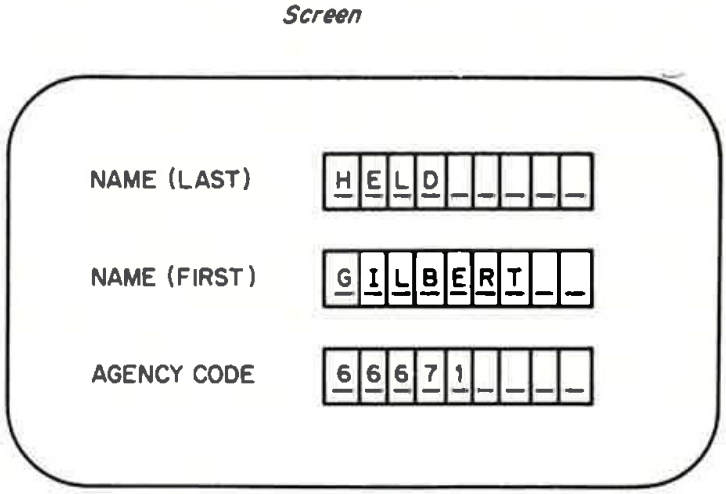


Figure 2.43 Forms mode data entry

a peripheral device attached to the terminal. Fixed fields are preceded by an 'FS' (start fixed field) character while variable fields are preceded by a 'GS' (begin variable field) character and a parameter character. The parameter character is used to define the allowable operations within the variable field such as numeric only, alphabetic only, alphanumeric, inhibit transmission and so on. The exact sequence of the GS and FS characters as well as the bit configuration of the parameter character to define allowable operations depends upon the terminals program. When in forms mode, certain keyboard operations are usually changed from those of the normal mode of operation. As an example, the TAB key on most displays permits the operator to move the cursor (positioning data entry marker) to the first character position of the next sequential variable field, permitting rapid skipping-over of variable fields for which no data is to be entered (Peterson, Bitner and Howard, 1978).

Transmission

The transmission of data in the forms mode is normally performed on a screen basis. When the operator depresses the TRANSMIT key, only the data previously entered into the variable fields is transmitted, with all trailing blanks eliminated.

Here, transmission can occur online to the computer or it can be to one of the peripheral units of the terminal. In the case of the latter, a large number of terminal screens may be batched onto a peripheral device such as a cassette or floppy disc for transmission to the computer at one time. Using this combination of forms mode encoding and off-line storage for transmission by batching screens of information, computer system resources in the form of computer ports and line requirements can be reduced or used more effectively. By reducing the amount of transmission time required to send batched screen information, a reduction in the number of computer ports required to support remote terminals may be possible. Concerning more effective line utilization, consider the situation where 10 terminals operate in a poll and select environment connected via a common modem sharing unit and modem to a central computer as illustrated in Figure 2.44. In the configuration illustrated, all terminals except the terminal transmitting or receiving data are locked out for the duration of the transmission. Normally, blocks of data up to the screen size, 1920 (80 × 24) characters or less, are transmitted. At 4800 bps, the transmission of a 1920 8-bit character block to completely fill a screen would require 3.2 s. If 10 terminals were connected to the modem sharing unit with a round robin polling sequence and each operator transmitted or received a full screen of data, it would take 32 seconds, ignoring transmission overhead, until the first terminal operator could again transmit or receive information. Thus, reducing the number of characters transmitted and received through the employment of forms mode encoding can be used to decrease the response time of existing

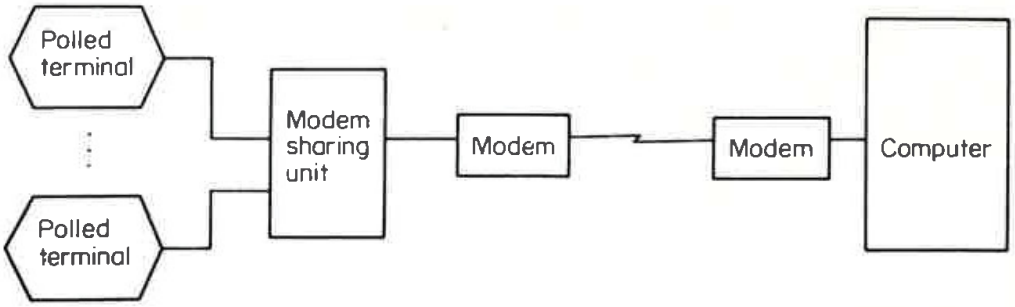


Figure 2.44 Forms mode encoding increases line service. Forms mode encoding reduces the poll and select time required per terminal, permitting more terminals to be connected on a shared line or an increase in throughput to existing terminals sharing the line

terminals or to permit additional terminals to be clustered without increasing overall response times.

Returning to the example in Figure 2.43, the 'HT' (horizontal tab) character is normally used as a variable field separator, resulting in the transmitted message indicated in the lower portion of that illustration. If a maximum of 8 characters can be entered into each of the 3 variable fields shown in Figure 2.43, a maximum of 26 characters (24 data and 2 horizontal tab characters) will be transmitted to the comptuer for each form completed. This method of forms mode data entry should be contrasted with conventional time-sharing as shown in Figure 2.45. Here, the message 'ENTER NAME (LAST), NAME (FIRST), AGENCY CODE' serves as a variable field indicator, denoting to the terminal operator the data to be entered. The carriage return (C/R) character acts as a line termination character; however, if data was entered incorrectly, such as alphabetic characters in an all-numeric field, data must first be sent to the computer for processing to determine that such an error has occurred. In such cases, the computer would transmit an error message to the terminal operator who would then hopefully retype the entire line correctly and retransmit the data. In contrast, using an intelligent terminal and forms mode operation the data entry operation can be preprocessed and such errors corrected prior to transmission.

In comparison with the operator depressing the transmit key on the display and having the forms mode method of operation transmit and clear the variable fields so new data can be entered, conventional time-sharing requires the program to prompt the operator to determine if more data is to be entered. The 'MORE?' and 'YES' (C/R)' sequence in Figure 2.45 add additional characters beyond the repeated message used as a variable field indicator. In comparing the sample forms mode data entry with the conventional time-sharing data entry example, 18 characters are required for the former while 65 characters, excluding line feed and carriage return characters, are required for the latter.

ENTER NAME (LAST), NAME (FIRST), AGENCY CODE

HELD, GILBERT, 6671 (C/R)

MORE?
YES (C/R)

ENTER NAME (LAST), NAME (FIRST), AGENCY CODE

Figure 2.45 Conventional time-sharing data entry. In conventional timesharing, the prompt messages requesting data as well as the user responses are transmitted

2.9 STATISTICAL ENCODING

One common element of the eight previously discussed data compression techniques is that they all operate upon characters codes of a fixed bit size. In comparison with those compression methods statistical encoding takes advantage of the probabilities of occurrence of single characters and groups of characters, so that short codes can be used to represent frequently occurring characters or groups of characters while longer codes are used to represent less frequently encountered characters and groups of characters. The statistical encoding process 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 representations 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 of dots and dashes were used to represent characters that appear less frequently. Included in the class of statistical compression methods is the Huffman coding technique. Prior to discussing statistical encoding techniques in detail, a review of some basic information theory concepts is warranted. These concepts will provide an understanding of how redundancy can be statistically reduced.

Information theory

For a system capable of transmitting at n discrete levels at λ second intervals, the number of different signal combinations in T seconds is $n^{T/\lambda}$. Since information is proportional to the length of time of transmission, we can take the logarithm of $n^{T/\lambda}$, to obtain the information transmitted in T seconds being proportional to $(T/\lambda) \log n$.

The proportionality factor will depend upon the base of the logarithm used, the most common choice being the base 2. This results in the information unit H becoming

$$H = \frac{T}{\lambda} \log_2 n.$$

The unit of information defined in the preceding manner is known as the bit or binary digit. For the transmission of data over a 20 second period using 2 discrete levels (0 and 1) at 1 second intervals, the information content becomes:

$$H = \frac{20}{1} \log_2 2 = 20 \text{ bits.}$$

The capacity of a given system is defined as the maximum amount of information per second that a system can transmit and can be expressed in bits per second. Thus, the capacity of the preceding example becomes:

$$C = \frac{H}{T} = \frac{1}{\lambda} \log_2 n = \frac{1}{1} \log_2 2 = 1 \text{ bit/s.}$$

The relative frequency of occurrence of any one combination or event is defined as the probability of occurrence, denoted symbolically as P, where

$$P = \frac{\text{number of times an event occurs}}{\text{total number of possibilities}}.$$

If n possible events are specified to be the n possible signal levels, then $P = 1/n$ for events that are equally likely to occur. The information contained by the appearance of any one event in one time interval (H_1) becomes:

$$H_1 = \log_2 n = -\log_2 P \text{ bits/interval}$$

where P represents $1/n$. During t periods of time, consisting of periods λ s long, we should have t times as much information, or

$$H = tH_1 = -t \log_2 P \text{ bits in } t \text{ periods.}$$

Since the number of periods, t, equals the total time, T, divided by the number of intervals, λ, the information available in T seconds becomes:

$$H = -\frac{T}{\lambda} \log_2 P = \frac{T}{\lambda} \log_2 n \text{ bits in } T \text{ s.}$$

With the preceding serving as a foundation, we can consider the case where different events or signal levels do not have equal probabilities of occurrence. Let us assume just two levels are to be transmitted, 0 or 1, the

first with probability P and the second with probability Q, where $P + Q = 1$. Then:

$$P = \frac{\text{number of times 0 occurs}}{\text{total number of possibilities}}$$

$$Q = \frac{\text{number of times 1 occurs}}{\text{total number of possibilities}}$$

The information content of a long message consisting of many 0s and 1s is thus dependent upon $P^*\log_2 P + Q^*\log_2 Q$ which is the information in bits per occurrence of a 0 or 1 times the relative frequency of occurrence of the bit value. We can let the frequency of occurrence of each possible signal level or signal be denoted by P_i, where $P_1 + P_2 + \ldots + P_n = 1$. Then each interval carries $-\log_2 P_i$ bits of information. In t periods of time, i will appear on the average t^*P_i times. By summing the information in bits contributed on the average by each symbol appearing t^*P_i times over the t intervals, we obtain:

$$H = -t^* \sum_{i=1}^{n} P_i \log_2 P_i \text{ bits in } t \text{ periods.}$$

For the interval T, we then obtain:

$$H = -\frac{T}{\lambda}^* \sum_{i=1}^{n} P \log_2 P_i \text{ bits in } T\text{s.}$$

For a message with n possible symbols or levels with probability of occurrence P_i to P_n, the average information per single symbol interval of λ is:

$$H_{\text{avg}} = -\sum_{i=1}^{n} P_i \log_2 P_i \text{ bits/symbol interval.}$$

The above equation represents the mathematical definition of entropy, a term used in information theory to denote the average number of bits required to represent each symbol of a source alphabet.

Based upon the preceding, it becomes possible to compute the redundancy contained in information. Since the unit of information is $\log_2 n$ for a system capable of transmitting at n discrete levels, its redundancy becomes

$$R = \log_2 — H_{\text{avg}}$$

Then, when there is zero redundancy:

$$H_{\text{avg}} = \text{iog}_2 n$$

Table 2.14 Coin toss representing four-symbol alphabet

Coin toss outcome	Alphabet symbol	Outcome probability	Representative code
TT	X_1	0.25	00
TH	X_2	0.25	01
HT	X_3	0.25	10
HH	X_4	0.25	11

Entropy examples

We can experiment with the well-known coin tossing model in order to expand upon the concept of entropy. The two sides of a coin, heads (H) and tails (T), correspond to members X_1 and X_2 from an alphabet X containing two symbols. If we toss two coins and encode the results so that T = 0 and H = 1, the coin toss result probabilities correspond to a four-symbol alphabet as tabulated in Table 2.14. The entropy or average number of bits required to represent each possible outcome or symbol from our four-symbol alphabet becomes:

$$H_{\text{avg}} = -\sum_{i=1}^{4} P_i \log_2 P_i = -4 \times 0.25 \log_2 0.25 = 2.$$

For the coin toss experiment results listed in Table 2.14, two binary symbols were required to encode each alphabetic symbol. If for some reason the coin toss was fixed such that only tails (T) occurs, the only symbol required in our alphabet would be X_1. Under this condition, we would never have to do any coin tossing to determine the outcome since the result is always known in advance. The entropy of this one-symbol alphabet can be computed as follows:

$$H_{\text{avg}} = -\sum_{i=1}^{4} P_i \log_2 P_i = -\sum_{i=1}^{1} \log_2 1 = 0.$$

In this case, since the outcome is known in advance the symbol provides no information; hence, its entropy is zero.

We can again fix the coin toss experiment; however, this time we will fix it so the probability of tails (T) occurring is increased to 0.75, leaving a 0.25 probability of heads occurring. Under these circumstances, the tabular results of the coin toss outcomes representing a four-symbol alphabet would be as

Table 2.15 Fixed coin-toss representing four-symbol alphabet. Probability of head = 0.25; probability of tail = 0.75

Coin toss outcome	Alphabet symbol	Outcome probability	Representative code
TT	X_1	0.5626	00
TH	X_2	0.1875	01
HT	X_3	0.1875	10
HH	X_4	0.0625	11

listed in Table 2.15. Although the representative code, number of coin toss outcomes and alphabet symbols have remained the same, the outcome probabilities have changed. Thus, the probability of two tails is now 0.75 times 0.75 or 0.5625 and so on. The entropy of this four-symbol alphabet is now:

$$H_{\text{avg}} = -\sum_{i=1}^{4} P_i \log_2 P_i = 0.5625 \log_2 0.5625 + 0.1875 \log_2 0.1875 + 0.1875 \log_2 0.1875 + 0.0625 \log_2 0.0625 = 1.62 \text{ bits per symbol.}$$

Based upon the preceding, let us compute the redundancy in the fixed coin-toss experiment. Since a two-symbol event results in four discrete levels

$$R = \log_2 n = H_{\text{avg}} = \log_2 4 - 1.68 = 2 - 1.68 = 0.38$$

In comparison with the first coin-toss experiment, the average number of bits required to represent a symbol from the four-symbol alphabet has been reduced by 0.38. This indicates that using another type of coding scheme to represent the four-symbol alphabet could result in an approximate 20 per cent reduction from the two bits per symbol previously used to represent the four-symbol alphabet. To obtain this reduction, we must assign short codes to the most frequently occurring symbols of the alphabet and longer codes to the less frequently encountered symbols. This method will result in a long string of data symbols having, on the average, fewer bits per symbol and is the foundation for what is known as Huffman coding. (Dishon, 1977; Moilanen, 1978).

Huffman coding

Huffman coding is a statistical data-compression technique whose employment will reduce the average code length used to represent the symbols of an alphabet. The alphabet can be the English language alphabet or a type of data-coded alphabet such as the ASCII or EBCDIC character sets.

The Huffman code is an optimum code since it results in the shortest average code length of all statistical encoding techniques. In addition, Huffman codes have a prefix property which means that no short code group is duplicated as the beginning of a longer group. This means that if one character is represented by the bit combination 100, then 10001 cannot be the code for another letter since in scanning the bit stream from left to right the decoding algorithm would interpret the 5 bits as the 100 bit configuration character followed by a 01 bit configuration character.

The Huffman code can be developed through the utilization of a tree structure as illustrated in Figure 2.46. Here, the symbols are first listed in descending order of frequency of occurrence. The groups with the smallest frequencies (X_3 and X_4) are combined into a node with a joint probability of occurrence of 0.25. Next, that node is merged with the next lowest probability of occurrence symbol or pair of symbols. In this illustration, the pair X_3X_4 is merged with X_2 to produce a node whose joint probability is 0.4375. Finally, the node representing the probabilities of occurrence of X_2,X_3 and X_4 is merged with X_1, resulting in a node whose probability of occurrence is unity. This master node represeents the probability of occurrence of all four characters in the character set. By assigning binary 0s and 1s to every segment emanating from each node, one can derive the Huffman code for each character. The code is obtained by tracing from the 1.0 probability node to each character symbol, noting the 1s and 0s encountered.

The average number of bits per symbol can be calculated by multiplying the Huffman code lengths by their probability of occurrence. Thus, the code uses:

$$1*0.5625 + 2*0.1875 + 3*0.1875 + 3*0.0625$$

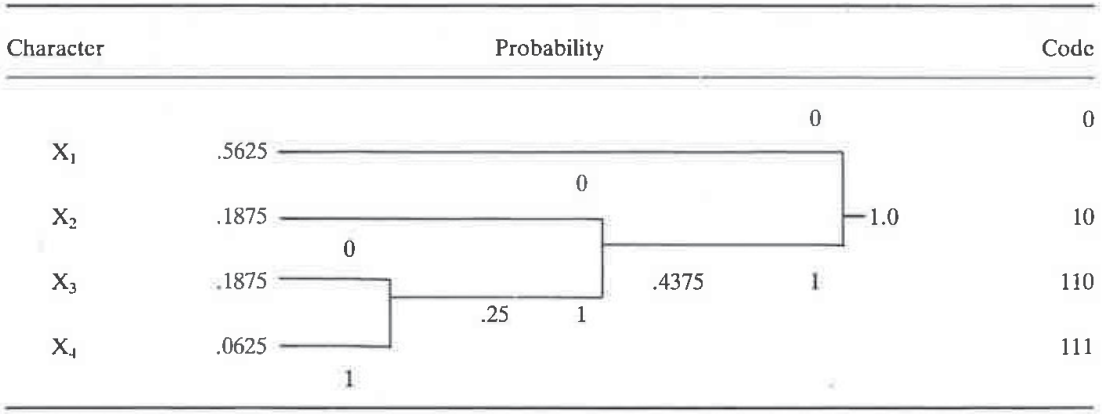


Figure 2.46 Huffman code development employing a tree structure. Huffman codes can be developed by employing a tree structure. The Huffman code resulting from this construction method is derived by tracing from the 1.0 probability node to each source character (symbol), noting 1s and 0s encountered

or 1.63 bits per symbol. Note that the Huffman code result of 1.63 bits per symbol closely approaches the entropy of 1.62 bits per symbol (Dishon, 1977; Moilanen, 1978).

A key property of the Huffman code is that it can be instantaneously decoded as the coded bits in the compressed data stream are encountered. An example of the instantaneous decoding property is illustrated in Figure 2.47. Here, the compressed data stream can be decoded immediately by reading left to right without waiting for the end of the block of data to occur.

The substitution of a number of bits representing a particular data character or group of characters is a fairly simple process when the number of substitutions is limited. As the number of substitutions increases, the complexity of the substitution process increases. In Figures 2.48 and 2.49, the development of a Huffman code for the English alphabet is illustrated. The tree structure used to develop the code shown in figure 2.48 is produced as follows:

A. The character set is arranged in a column on the left in order of decreasing frequency of occurrence with the frequency placed in a column next to each character.
B. Commencing at the bottom of the table, lines are drawn horizontally from each character frequency. The lines with the two lowest frequencies of occurrence are merged and their associated frequencies are added to obtain a composite frequency. This composite frequency is entered on a single new line and reflects the combined frequency of the previously paired characters.
C. The process of combining the two lowest frequency lines into a single line containing combined frequencies is continued until all the lines have been merged.

After the tree has been developed, the Huffman code for each character can be assigned by placing a 0 bit to one side of each nodal point and a 1 bit to

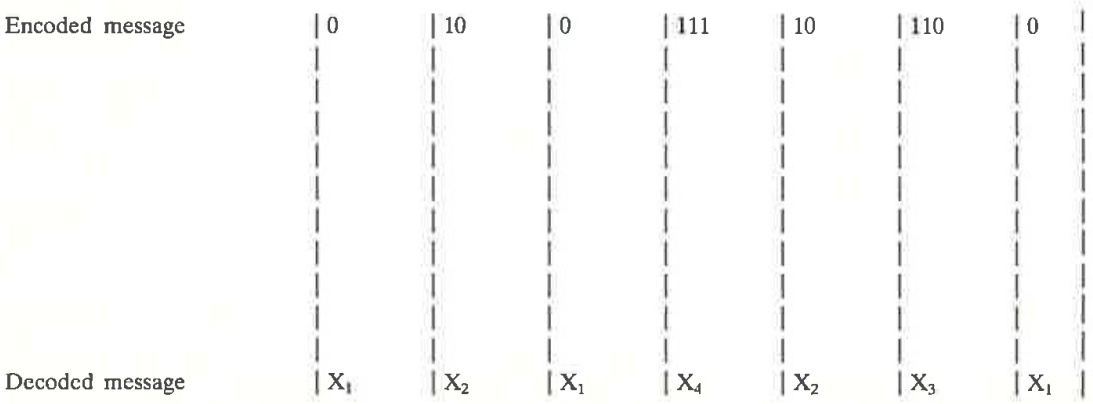


Figure 2.47 Instantaneous decoding property. One of the key properties of the Huffman technique is the fact that encoded data can be instantly decoded

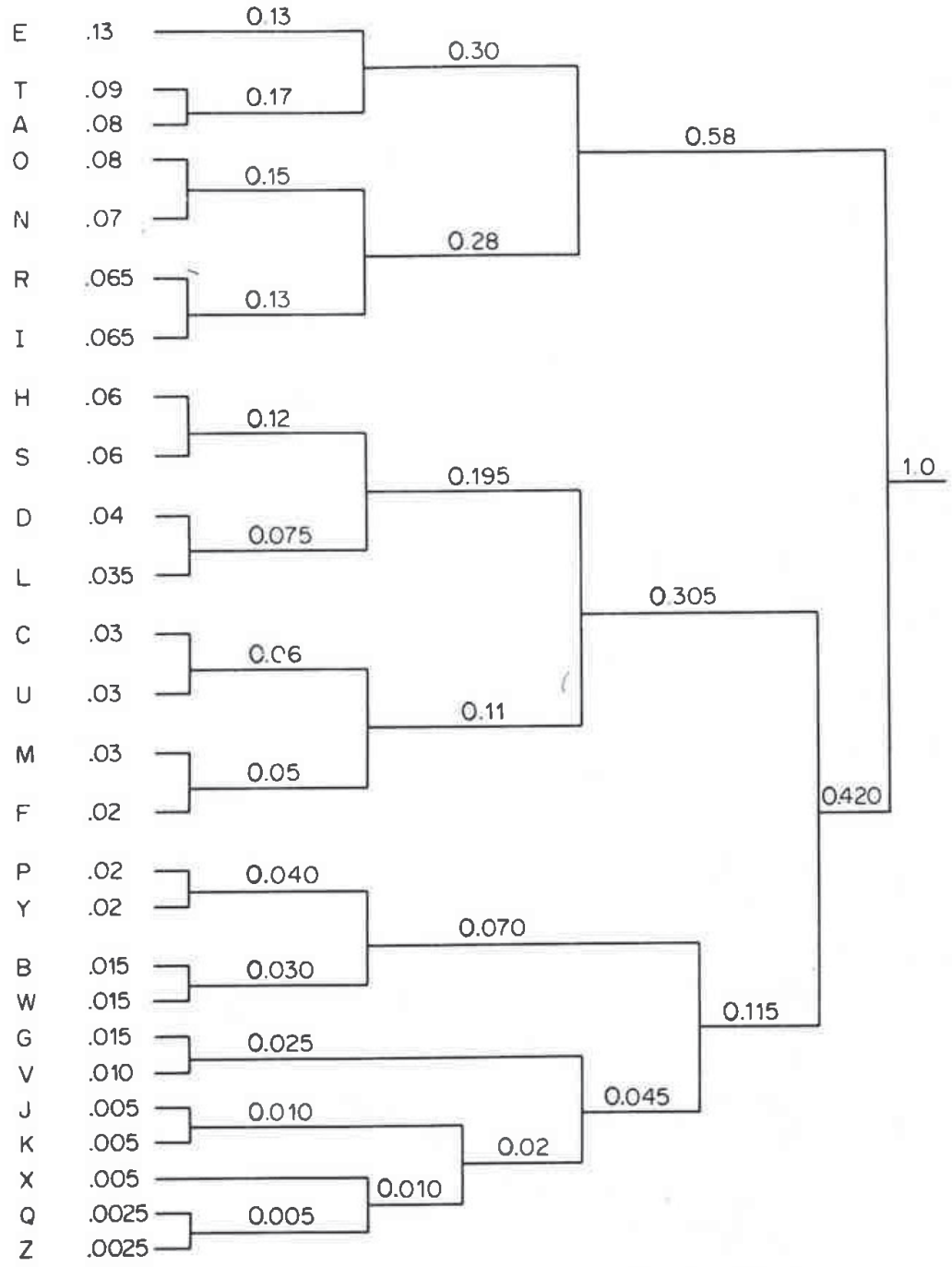


Figure 2.48 Developing a tree structure for the alphabet

the other path emanating from that point towards the left-hand symbol. The assignment of 0 and 1 bits is arbitrary. The appropriate bit sequence assigned to each data character is then determined by tracing the route from the master nodal point where the probability of all character frequencies of occurrence is unity back to the starting node for the appropriate character, noting the bits assigned to the path. The assignment of bits to the paths and

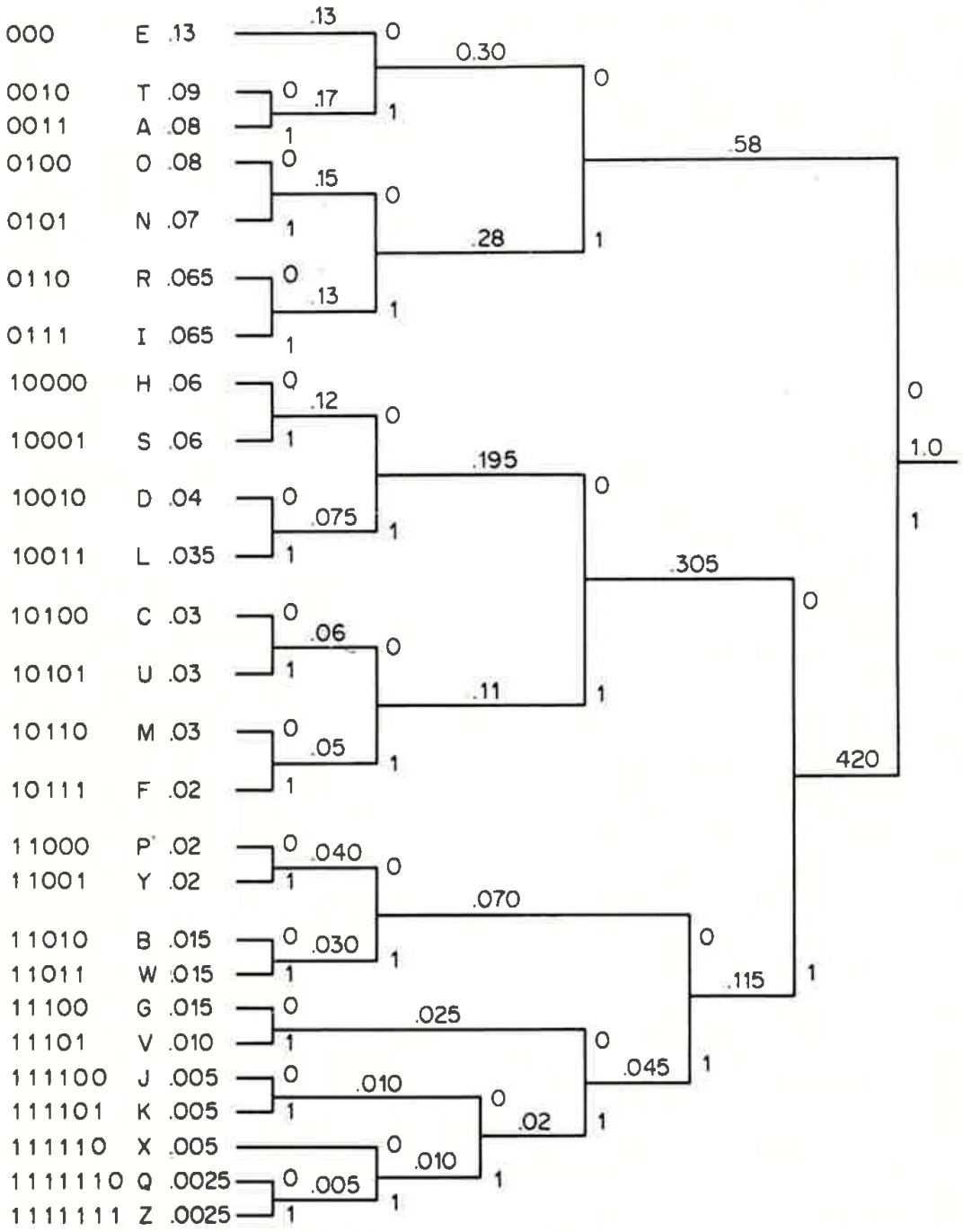


Figure 2.49 Assigning the Huffman code

the resulting Huffman coded values for the English alphabet are illustrated in Figure 2.49.

The number of bits required to encode a letter using the Huffman technique can be determined from the following formula:

$$b = f(-\log_2 P)$$

where:

P= probability of occurrence of the letter
$f(x)$ = the closest integer greater than or equal to x.

Since the probability of E is 0.13 and $-\log_2 0.13$ is 2.94, then the integer greater than or equal to 2.94 is 3. Thus, 3 bits are required to encode the letter E (Peterson, Bitner and Howard, 1978).

Information requirements

To develop a Huffman code whose average code length will approach its entropy requires the frequency distribution of the characters or symbols to be encoded to be known in advance. Since the frequency distribution of a data stream is proportional to the end use of the stream, this factor can result in a preselected frequency distribution used to develop a Huffman code resulting in a code far from optimum during certain data transmission sequences. As an example, the frequency distribution of English text, such as that resulting from a data file used for computerized typesetting, may be quite different from the data file containing the results of a FORTRAN program compilation. In the first instance, the distribution of characters should follow the distribution of normal English, with E the most frequently occurring character while Z is one of the least frequently characters. For the FORTRAN compilation, special characters such as parenthesis, + for addition, − for subtraction, * for multiplication and/for division have a high degree of occurrence not normally encountered in English text.

To compensate for frequency distribution differences, several encoding schemes can be considered. First, the analysis of mixed data files can be conducted employing the computer program listed in Appendix B (p. 00). This will enable one to ascertain the appropriate relationship between the frequency of occurrence of characters of different types of data.

A second method to consider is an adaptive Huffman encoding technique. Such a technique might first require a frequency analysis of a large block of data which would then be encoded based upon that distribution. Prior to the transmission of the encoded data, a table of the symbols and Huffman codes developed for each symbol must be transmitted to enable the encoded data to be successfully decoded. With a little imagination, one can visualize that frequently changing data streams would result in numerous tables as well as encoded data being transmitted. These tables can be considered as overhead, resulting in the compression frequency decreasing as the number of data-stream frequency distributions change per unit time. Another problem encountered with some adaptive Huffman coding techniques is determining the size of the data stream to sample and the sample intervals. The larger the sample, the greater the processing requirement becomes. If the data is to be transmitted, a buffer area is required to place the sample into while

the frequency analysis is conducted. Concerning the sample interval, if three FORTRAN jobs are followed by an English text job, all of equal size, T, sampling at $T, T + 2$ and $T + 4$ would result in the English text job being excluded from the sample. Since a remote batch terminal operator submits jobs and pulls system output, he or she knows ahead of time the type of job that will be transmitted to or received from the computer. For this type of operating situation, predefined frequency distributions can be selected by the operator and conveyed to the opposite end of the transmission link by the transmission of a special code.

To eliminate the previously described problems resulting from the generation of frequency tables, one can construct a truly adaptive or self-adapting Huffman encoding technique. This technique builds frequency tables at both ends of a transmission link as data transmission occurs and adaptively adjusts those tables during transmission. The reader is referred to Section 2.10 which discusses this technique in detail.

A third method of compensating for frequency distribution differences is by the use of a plain text code, which is used to indicate that the character following it should be reproduced exactly as received. This permits characters that rarely occur in the source data to be excluded from the encoding process and results in the development of one type of modified Huffman code. Here, one could group all characters of low frequency of occurrence into one probability of occurrence and assign a Huffman code to represent that summed probability. This would be the plain text code and would indicate that the next 8 bits represent an actual non-encoded data character. Without the use of a plain text code, large strings of, say, 20 or more bits might result in the representation of low frequency of occurrence characters. If the plain text code were 4 bits in length, then a maximum of 12 bits would be required to represent any low frequency of occurrence character. The pre-emption of a 4-bit code to signify that the next 8 bits are the plain text representation of an 8-bit character means that some relatively high frequency of occurrence character which would have had a 4-bit code as its Huffman representation will be represented by some longer code. Thus, although there will be no very long codes present, the mean number of bits per character will increase when a plain text code is employed.

Modified Huffman codes

The representation of characters and symbols by an appropriate Huffman code is excellent in theory if one desires to have the average number of bits per symbol approach entropy. In practice, however, a number of difficulties can arise when Huffman coding is applied to certain applications, most particularly in the area of facsimile transmission.

When applying Huffman coding to facsimile transmission, each facsimile line can be viewed as consisting of a series of black or white 'runs', each run consisting of a series of similar picture elements. If the type of the first

run is known, then the type of all successive runs will be known, as black and white runs must alternate. The probability of occurrence of each run of a given length of pels can be calculated and short code words can be used to represent runs that have a high frequency of occurrence while longer code words can be used to represent runs that have a low probability of occurrence. In a way similar to the changing of data processing jobs, statistics for the run-length probabilities associated with line scans change on a line-to-line and document-to-document basis. Thus, an optimum or near optimum code for a particular line or document may be far from optimum for a different line or document. A second major problem is the fact that the creation of the Huffman code on a real-time basis requires a large degree of processing power, normally in excess of the capabilities of facsimile machines where the cost of the scanner, transmitter/receiver, central logic and power supply results in a machine rental under a few hundred dollars per month to remain competitive. To reduce some real-time processing requirements, a table look-up approach can be employed. Since CCITT standards require 1728 pels per line, the use of a table look-up technique would require storage for 1728 variable length locations for each facsimile machine, each location containing a binary code word corresponding to a particular run length. The implementation problems associated with applying the full Huffman coding technique to facsimile applications resulted in the development of one modified Huffman coding scheme more suitable to the hardware cost constraints of the competitive facsimile marketplace.

In the development of a modified Huffman coding scheme for facsimile applications, a change was made which, while only rarely permitting the average symbol length to approach entropy, does permit significant compression while minimizing hardware and processing requirements. Here, the probability of occurrences of different run lengths of picture elements (pels) was calculated for all lengths of white and black runs based upon statistics obtained from the analysis of a group of 11 documents recommended by the CCITT as being typical. To reduce table look-up storage requirements, the Huffman code set was truncated by the creation of a base 64 representation of each run length and the utilization of two code tables to reduce the overall table size in comparison with the table size that would be required if only one table were used (McCullough, 1977).

Based upon the run-length probabilities of 11 typical documents, code tables were developed for run lengths ranging from 1 to 63 pels. Since the probability of occurrence of white runs differs from the frequency of occurrence of black runs, a table must be developed for both runs. This dual table set is listed in Table 2.16 for run lengths ranging from 0 to 63 pels. The code in this table set represent the least significant digit (LSD) of the code word and are often referred to as the termination code. In order to permit the encoding of runs in excess of 63 pels, a second set of code tables must be employed to handle runs ranging in size from 64 pels to the maximum line scan length of 1728 pels. These codes are listed in Table 2.17. These represent

Table 2.16 Least significant digit codes for the modified Huffman process

White run length	Code word	Base 64 representation	Black run length	Code word
0	00110101	0	0	0000110111
1	000111	1	1	010
2	0111	2	2	11
3	1000	3	3	10
4	1011	4	4	011
5	1100	5	5	0011
6	1110	6	6	0010
7	1111	7	7	00011
8	10011	8	8	000101
9	10100	9	9	000100
10	00111	a	10	0000100
11	01000	b	11	0000101
12	001000	c	12	0000111
13	000011	d	13	00000100
14	110100	e	14	00000111
15	110101	f	15	000011000
16	101010	g	16	0000010111
17	101011	h	17	0000011000
18	0100111	i	18	0000001000
19	0001100	j	19	00001100111
20	0001000	k	20	00001101000
21	0010111	l	21	00001101100
22	0000011	m	22	00000110111
23	0000100	n	23	00000101000
24	0101000	o	24	00000010111
25	0101011	p	25	00000011000
26	0010011	q	26	000011001010
27	0100100	r	27	000011001011
28	0011000	s	28	000011001100
29	00000010	t	29	000011001101
30	00000011	u	30	000001101000
31	00011010	v	31	000001101001
32	00011011	w	32	000001101010
33	00010010	x	33	000001101011
34	00010011	y	34	000011010010
35	00010100	z	35	000011010011
36	00010101	A	36	000011010100
37	00010110	B	37	000011010101
38	00010111	C	38	000011010110
39	00101000	D	39	000011010111
40	00101001	E	40	000001101100
41	00101010	F	41	000001101101
42	00101011	G	42	000011011010
43	00101100	H	43	000011011011
44	00101101	I	44	000001010100
45	00000100	J	45	000001010101
46	00000101	K	46	000001010110

Table 2.16 (continued)

White run length	Code word	Base 64 representation	Black run length	Code word
47	00001010	L	47	000001010111
48	00001011	M	48	000001100100
49	01010010	N	49	000001100101
50	01010011	O	50	000001010010
51	01010100	P	51	000001010011
52	01010101	Q	52	000000100100
53	00100100	R	53	000000110111
54	00100101	S	54	000000111000
55	01011000	T	55	000000100111
56	01011001	U	56	000000101000
57	01011010	V	57	000001011000
58	01011011	W	58	000001011001
59	01001010	X	59	000000101011
60	01001011	Y	60	000000101100
61	00110010	Z	61	000001011010
62	00110011	*	62	000001100110
63	00110100	#	63	000001100111

the most significant digit of the code word and are known as the master code.

When a run of 63 pels or less is encountered, the appropriate type of LSD code set is accessed to obtain a single base 64 code word. To encode a run of 64 pels or more, two base 64 code words must be used. First, the most significant digit code word is obtained from the MSD code table such that N*64, $1 \leq N \leq 27$, does not exceed the run length. Next, the difference between the run length and N*64 is obtained and the least significant digit is accessed from the appropriate LSD code table. Figure 2.50 shows an example of the table look-up operations for a sample sequence of black and white runs of various pel sizes. In the upper portion of this illustration, the relationship between a series of original video data and its representation in the modified Huffman code is tabulated.

To employ the modified Huffman coding scheme successfully, some rules must be developed and followed to alleviate a number of deficiencies inherent from employing a statistical encoding technique. In such techniques, code words do not contain any inherent positional information which is necessary for synchronization. This can be compensated for by making it a rule that the first run of each line must be a white run, even if it results in a run length of zero. Thereafter, runs must alternate between black runs and white

Table 2.17 Most significant digit codes for the modified Huffman process

White run length	Code word	Base 64 representation	Black run length	Code word
64	11011	1	64	0000001111
128	10010	2	128	000011001000
192	010111	3	192	000011001001
256	0110111	4	256	000001011011
320	00110110	5	320	000000110011
384	00110111	6	384	000000110011
448	01100100	7	448	000000110101
512	01100101	8	512	0000001101100
576	01101000	9	576	0000001101101
640	01100111	a	640	0000001001010
704	011001100	b	704	0000001001011
768	011001101	c	768	0000001001100
832	011010010	d	832	0000001001101
836	011010011	e	836	0000001110010
960	011010100	f	960	0000001110011
1024	011010101	g	1024	0000001110100
1088	011010110	h	1088	0000001110101
1152	011010111	i	1152	0000001110110
1216	011011000	j	1216	0000001110111
1280	011011001	k	1280	0000001010010
1344	011011010	l	1344	0000001010011
1408	011011011	m	1408	0000001010100
1472	010011000	n	1472	0000001010101
1536	010011001	o	1536	0000001011010
1600	010011010	p	1600	0000001011011
1664	011000	q	1664	0000001100100
1728	010011011	r	1728	0000001100101
EOL	0000000000		EOL	00000000001

runs. To denote the beginning and end of each scan line, a unique line delineation code, sometimes called an end-of-line code (EOL), can be employed. Once each line is encoded, fill bits of 0s may be employed as pad bits prior to transmitting the EOL for timing purposes. The end result of the incorporation of these rules permits a line format to be defined as shown in Figure 2.51. Through the incorporation of the modified Huffman coding technique, the transmission time of a typical business document has been reduced to under 60 s at a transmission rate of 4800 bps.

The significance of the reduction becomes apparent when one considers that the resolution of 1780 pels per line and 96 horizontal lines per inch results in a total of 1 410 048 pels for an $8\frac{1}{2} \times 11$ document. Without compression, a

Original video data	Modified Huffman code base 64 representation	Modified Huffman code base 2 representation MSD	LSD
5 black pels	5 (black)	NA	0011
17 white pels	h (white)	NA	101011
32 black pels	w (black)	NA	000001101010
32 white pels	w (white)	NA	00011011
728 black pels	bϕ (black)	0000001001011	00000010111
1728 white pels	rϕ (white)	010011011	00110101
64 black pels	1ϕ (black)	0000001111	0000110111
55 white pels	T (white)	NA	01011000
1028 white pels	g2 (white)	011010101	0111

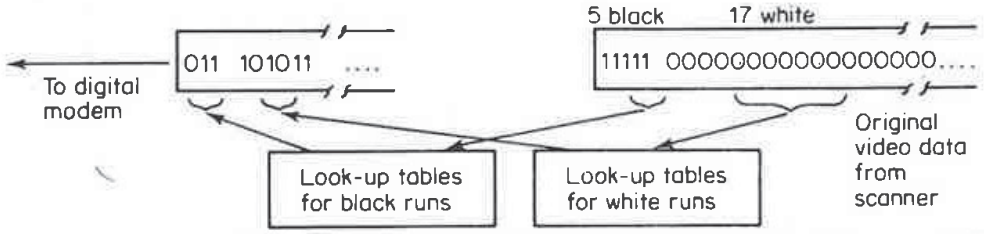


Figure 2.50 Encoding using the modified Huffman code. By a sequence of tabular references for black and white runs the modified Huffman code is constructed

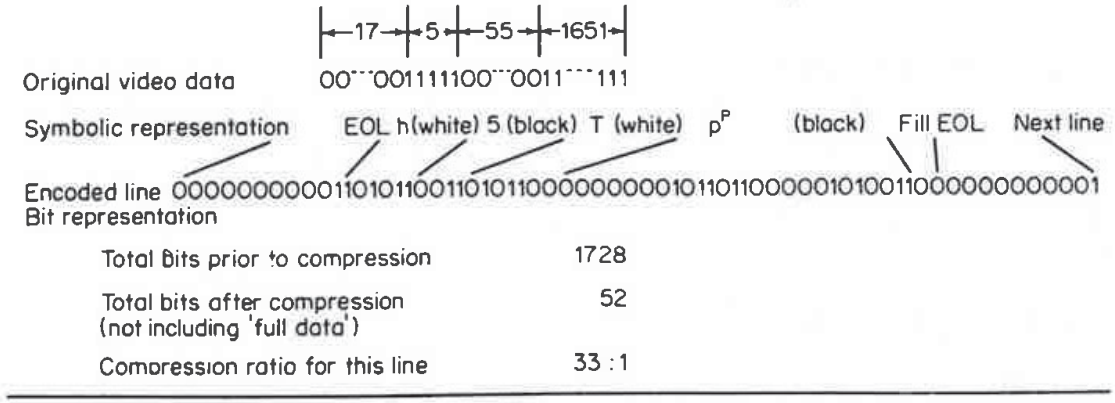


Figure 2.51 Rules define line format. To denote the beginning and end of each scan line an end-of-line code (EOL) is employed.

transmission time of approximately 5 minutes would be required for the data without considering the transmission of the end-of-line codes.

Shannon–Fano coding

Similar to Huffman coding, Shannon–Fano coding results in a variable length code that is instantly decodable. Prior to developing the code for each character in your character set, you must determine the probability of

occurrence of each character. Then, arrange your character set in descending order based upon the probability of occurrence of each character.

Once your character set is arranged in descending order of its probability of occurrence, the set must be divided into two equal or almost equal subsets based upon the probability of occurrence of the characters in each subset. The first digit in one subset is assigned a binary zero value while a binary one is assigned as the first digit in the second subset. This process of forming subsets is repeated until the character set is completely subdivided. Then, a suffix bit is added to each character in a two-character subset as required to distinguish one character's binary composition from the other character in the subset.

To obtain an understanding of the Shannon–Fano coding procedure, let us assume our character set contains seven characters whose probabilities of occurrence are indicated in Table 2.18.

By arranging the characters in the character set in descending order based upon their probability of occurrence, we can begin to form our subsets. In our subset construction process, we will group the characters into each subset so that the probability of occurrence of the characters in each subset is equal or as nearly equal as possible. Then we will assign binary ones to one subset and binary zeros to the other subset and continue to repeat the process until all possible subsets are constructed. Figure 2.52 illustrates this process.

Note that after the initial coding process is completed, the subsets represented by the character pairs X_6,X_3 and X_4,X_5 are not unique. Thus, a binary 1 and 0 must be added to the pairs in each subset. Doing so results in the completion of the variable length coding process in which each character is represented by a unique bit combination that is instantaneously decodable. The completed code for each character in our character set is illustrated in Figure 2.53.

Efficiency comparison

To compare the efficiency of the Shannon–Fano coding process to the previously covered Huffman coding technique, let us develop the Huffman

Table 2.18 Character set probability of occurrence

Character	Probability of occurrence
X_1	0.10
X_2	0.05
X_3	0.20
X_4	0.15
X_5	0.15
X_6	0.25
X_7	0.10

Character	*Probability*	*Code*
X_6	0.25	1
X_3	0.20	1
X_4	0.15	0 1
X_5	0.15	0 1
X_1	0.10	0 0 1
X_7	0.10	0 0 0
X_2	0.05	0 0 0

Figure 2.52 Initial Shannon–Fano coding process

Character	*Probability*	*Code*
X_6	0.25	1 1
X_3	0.20	1 0
X_4	0.15	0 1 1
X_5	0.15	0 1 0
X_1	0.10	0 0 1
X_7	0.10	0 0 0 1
X_2	0.05	0 0 0 0

Figure 2.53 Completed Shannon–Fano coding process

code for the character set whose probability of occurrence was previously listed in Table 2.18. Figure 2.54 (top) illustrates the construction of the Huffman code for the 7-character character set listed in Table 2.18. The lower portion of that illustration shows the assignment of binary 1s and 0s to each path member and the resulting Huffman code for each character when the binary digits in each path are recorded beginning at the unity or apex point in the coding tree.

Table 2.19 compares the codes generated by the Shannon–Fano coding procedure to the Huffman coding procedure for the 7-character character set used for each coding example. The average code length generated by each coding procedure can be computed by using the formula:

$$\lambda = \sum_{i=1}^{7} \lambda_i P_i$$

For the Shannon–Fano code, the average code length is:

$$\lambda = 2 \times 0.25 + 2 \times 0.20 + 3 \times 0.15 + 3 \times 0.15 + 3 \times 0.10 + 4 \times 0.10 + 4 \times 0.05 = 2.7 \text{ bits}$$

For the Huffman code, the average code length is:

$$\lambda = 2 \times 0.25 + 3 \times 0.75 = 2.75 \text{ bits}$$

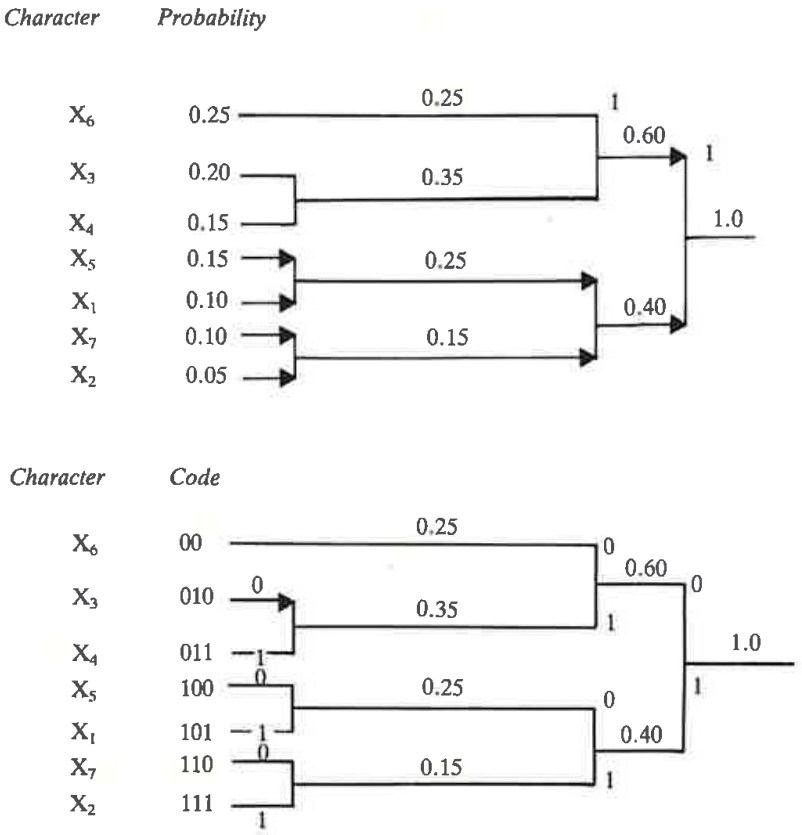


Figure 2.54 Huffman code construction

Table 2.19 Coding comparison

Character	Probability	Shannon–Fano code	Huffman code
X_6	0.25	11	00
X_3	0.20	10	010
X_4	0.15	011	011
X_5	0.15	010	100
X_1	0.10	001	101
X_7	0.10	0001	110
X_2	0.05	0000	111

Although the Shannon–Fano code is more efficient since its average code length is less than that of the Huffman code, the reader should note that it is not necessarily always more efficient. The previous illustrations were based upon one group of assigned probabilities of occurence to a 7-character character set. To illustrate how efficiencies between the two codes can change, let us assume that the probabilities of occurrence of the characters in the character set are now represented by the data listed in Table 2.20.

Table 2.20 Revised character set

Character	Probability of occurrence
X_1	0.0625
X_2	0.0625
X_3	0.1250
X_4	0.1250
X_5	0.0625
X_6	0.5000
X_7	0.0625

The top portion of Figure 2.55 illustrates the Shannon–Fano coding process while the lower portion of that illustration shows the Huffman coding process. Note that based upon the revisions in the probability of occurrence of the characters in the character set, the average code length for each coding technique is the same. That is, the average code length for the Shannon–Fano coding process is:

$$\lambda = 1 \times 0.5 + 3 \times 0.125 + 3 \times 0.125 + 4 \times (4 \times 0.0625) = 2.25 \text{ bits}$$

which is exactly the same code length obtained from the Huffman coding process.

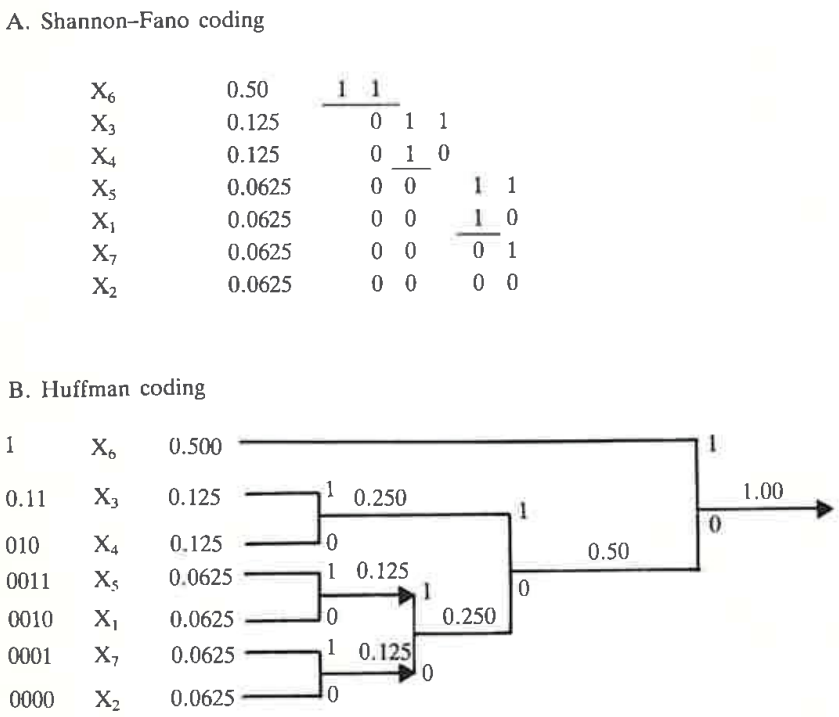


Figure 2.55 Recoding the new character set

Now let us assume that the probability of occurrence of each character in the character set is again altered. Suppose the new probabilities of occurrence are as indicated in Table 2.21.

The top portion of Figure 2.56 illustrates the Shannon–Fano coding process for the revised character set while the lower portin shows the Huffman coding process.

Now let us compute the average code length for each coding process. For the Shannon–Fano code, its average code length is:

$$\lambda = 2 \times 0.4 + 2 \times 0.1 + 3 \times 0.1 + 3 \times 0.1 + 3 \times 0.1 + 4 \times 0.1 + 4 \times 0.1 = 2.7 \text{ bits}$$

Table 2.21 New character set probabilities

Character	Probability of occurrence
X_1	0.10
X_2	0.10
X_3	0.10
X_4	0.10
X_5	0.40
X_6	0.10
X_7	0.10

A. Shannon–Fano coding

X_6	0.40	1	1		
X_1	0.10	1	0		
X_2	0.10	0	1	1	
X_3	0.10	0	1	0	
X_4	0.10	0	0	1	
X_5	0.10	0	0	0	1
X_7	0.10	0	0	0	0

B. Huffman coding

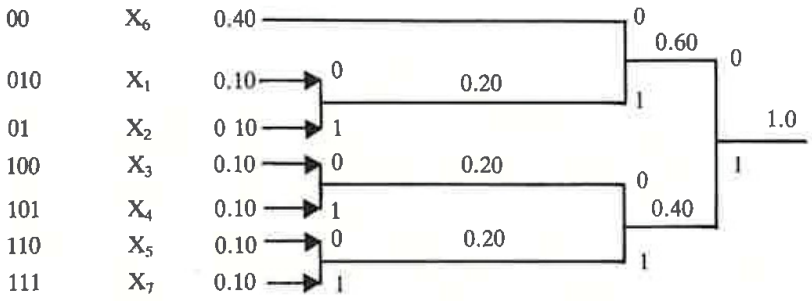


Figure 2.56 Recoding the revised character set

For the Huffman code, its average code length is:

$$\lambda = 2 \times 0.4 + 3 \times 0.6 = 2.6 \text{ bits}$$

Thus, in this instance the Huffman code results in a more efficient bit representation of the character set than the Shannon–Fano coding method.

In general, as the probabilities of each character in the character set approach probabilities that are negative powers of 2 both codes will have their average code length approach entropy. That is, if all the probabilities of the characters in the character set were negative powers of 2 the average code length would equal entropy and the efficiency of each code would be 100 per cent. If the probabilities of occurrence of the elements in a set have a large variance, the Shannon–Fano code will be more efficient while the Huffman code becomes more efficient as the variance in probabilities decreases between elements in the set.

2.10 ADAPTIVE COMPRESSION

The examples of compression techniques previously covered in this chapter were based upon the assumption of prior knowledge of the data to be compressed. Using this prior knowledge permits us to predefine compression indicating characters and the character sequences which can then be substituted for strings of data containing predefined redundancy. In addition, we can construct a fixed compression table that will enable the statistical encoding of data to occur based upon the expected frequency of occurrence of the data. Run length and diatomic encoding are examples of character sequence and character substitution where some prior knowledge or expectation of the composition of the data resulted in the definition of a single character or short sequence of characters to replace longer sequences of characters. Huffman and modified Huffman encoding are examples of data compression techniques that would employ a fixed compression table whose construction is based upon prior knowledge or assumed knowledge of the data.

The fixed compression table

Figure 2.57 illustrates the general format of a fixed compression table. In actuality, this table can be two separate tables, with a relationship established between the elements in each table or the table can consist of paired entries. Each character in the original data stream is compared to the entries in the 'data to compress' part of the compression table. When the character to be encoded matches an entry in the 'data to compress' portion of the table, the code that represents the character is extracted from the compression table. Thus, the process required to replace each character with its statistical code is reduced to a table look-up operation.

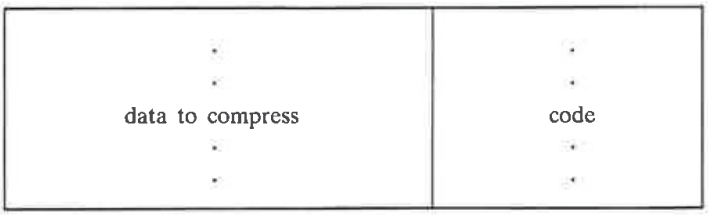


Figure 2.57 Fixed compression table format

Data to compress	Resulting code
X_1	0
X_2	10
X_3	110
X_4	111

Figure 2.58 Resulting fixed compress table

To illustrate the utilization of a fixed compression table, let us assume that as a result of an analysis of a 4-character character set (X_1,X_2,X_3 and X_4) we determined that the probability of occurrence of each character was 0.5625, 0.1875, 0.1875 and 0.0625 respectively. The Huffman code previously developed in Figure 2.46 for this character set results in the assignment of 0, 10, 110 and 111 to characters X_1 to X_4. Thus, based upon prior knowledge of the data we can develop the Huffman code for the character set which then enables us to construct the fixed compression table for this character set. This table is illustrated in Figure 2.58.

The probability of occurrence of the characters in the character set must be determined prior to constructing a fixed compression table.

The use of a fixed compression table requires each character in the original data string to be compared to the 'data to compress' entries in the table. When a match occurs, the coded entry then replaces the character in the original data string. Thus, the sequence of characters

$$X_2X_4X_1X_2X_2$$

would be replaced by the Huffman code for each character, which would result in the binary sequence:

1011101010

Efficiency

What happens to the efficiency of the predefined Huffman code when the probability of occurrence of the characters in the character set differs from

the prior or expected knowledge of their frequency of occurrence? Since short codes are employed to represent frequently occurring characters while longer codes represent characters that occur less frequently, the predefined Huffman codes variance from entropy increases as the data varies from its prior or expected frequency of occurrence. One technique that can be used to maintain the efficiency of the resulting code obtained by compressing data statistically is the use of an adaptive or dynamic compression scheme, which is the main topic of this section.

Adaptive compression

When adaptive compression is performed, the data to be compressed is analysed in order to generate appropriate changes into a variable compression table.

Similar to the use of a fixed compression table, each character in the original data stream is first compared to the entries in the 'data to compress' portion of the compression table. When a match occurs, the corresponding entry in the 'resulting code' portion of the table is extracted and represents the statistically encoded character.

Where adaptive compression differs from fixed compression is in the employment of a count field in the compression table. This field is continuously updated and serves as a mechanism for the resequencing of the entries in the table. The updating of the field occurs after a character in the original data stream is matched with an entry in the 'data to compress' portion of the compression table and the 'resulting code' is extracted from the table. Then, a comparison of the entries in the count field occurs. Based upon the results of the comparison, the character and its count value may be repositioned in the compression table. This technique ensures that whenever the composition of the data changes, the compression table changes in tandem, resulting in a variable compression table that provides the most efficient statistical compression possible. Figure 2.59 illustrates how a variable compression table can be resequenced based upon the composition of the data being transmitted.

Figure 2.59, part A, illustrates the initial composition of the variable compression table. Although this table was initially established based upon the frequency of occurrence of the characters in the character set, since the table is self-adjusting, we do not have to concern ourselves with the size of the sample used to initialize the entries into the table.

In Figure 2.59, part B, we assumed that the character X_2 was encountered. Since the binary code 10 is assigned to X_2 (Figure 2.59, part A), that bit string is transmitted, the count for X_2 is incremented by one and the variable compression table is resequenced. Similarly, at the receiver the bit sequence 10 is received, which is decompressed into the character X_2. The receiver then increments the count for X_2 in its compression table and its table is also resequenced.

A. Initial table

Data transmitted

Data to compress	Count	Resulting Code
X_1	0	0
X_2	0	10
X_3	0	110
X_4	0	111

B. X_2 encountered

10

Data to compress	Count	Resulting code
X_2	1	0
X_1	0	10
X_3	0	110
X_4	0	111

C. X_4 encountered

111

Data to compress	Count	Resulting code
X_2	1	0
X_4	1	10
X_1	0	110
X_3	0	111

D. X_2 encountered

0

Data to compress	Count	Resulting code
X_2	2	0
X_4	1	10
X_1	0	110
X_3	0	111

Figure 2.59 The variable compression table

In Figure 2.59, part C, we have assumed that the character X_4 is the next character encountered in the data to be compressed. Based upon the table then in use (Figure 2.59, part B), this character is encoded as the binary string 111. Next, the count of the frequency of occurrence of X_4 is incremented and the variable compression table is resequenced.

Figure 2.59, part D, assumes that the next character encountered in the original data string is X_2. Since the table illustrated in Figure 2.59, part C,

was then in use, X_2 is encoded as the single bit 0. Then the count for X_2 is incremented by one; however, since X_2 was at the top of the compression table, the table is not resequenced.

As illustrated in Figure 2.59, adaptive compression dynamically changes the order of the entries in the compression table in tandem with the changes in the frequency of occurrence of the characters in the character set. Thus, this method of implementing a statistical compression technique should always be more efficient than the utilization of fixed compression.

Coded example

Figure 2.60 contains the ADAPTC.BAS program listing. This BASIC language program was developed to illustrate many of the programming concepts involved in adaptive compression. For simplicity of illustration only four characters—E, T, I and O—are considered to be in the character set suitable for adaptive compression. All other characters encountered in the data strings the program will operate upon will be passed 'as-is' to the output buffer.

In line 115, the program branches to the subroutine commencing at line 400 which initializes the character table P$(I) to the characters E, T, I and O. Similar to the other coding examples presented in this chapter, line 130 obtains a line of up to 132 characters from a data file while line 140 obtains the length of the line.

The subroutine commencing at line 180 processes the records read from the data file. To illustrate the operation of adaptive compression, when the characters E, T, I and O are encountered they will be replaced by the characters #, $, % and &. For simplicity, the resulting Huffman table will be displayed on a line-by-line basis instead of on an individual character basis while the code changes in the adaptive compression table will similarly occur on a line by line basis.

In line 230, the subroutine commencing at line 2120 is invoked. This subroutine prints the current values of the compression table. Next, lines 240 to 280 examine the extracted record from the data file on a line-by-line basis, comparing each character in the record to any of the characters in our compressible four-character character set (E, T, I, O). If a match occurs, the subroutine commencing at line 350 is invoked. Otherwise, the program simply places the character extracted from the input record into the output buffer.

The subroutine commencing at line 350 sets the character match flag to one and then adds 34 to the value of K in line 365. This action sets the ASCII value of V to either the #, $, % or & character which is used in this example to illustrate the substitution of a Huffman code for an appropriate character in the four-character character set we are using. Next, line 370 inserts the substituted character into the output buffer and the count is then incremented in line 380.

```
10 REM ADAPTC.BAS PROGRAM
20 DIM O$(132)
30 WIDTH 80:CLS
40 '**********MAIN ROUTINE**********************
50 '* THIS ROUTINE READS RECORDS FROM AN ASCII *
60 '* FILE INTO A STRING CALLED X$ WHICH IS    *
70 '* THEN PASSED TO SUBROUTINES FOR COMPRESSION
80 '*********************************************
90 PRINT "ENTER ASCII FILENAME. EG, ADAPT.DAT"
100 INPUT F$: OPEN F$ FOR INPUT AS #2
105 OPEN "ADAPTC.DAT" FOR OUTPUT AS #3
110 PRINT "PATIENCE - INPUT PROCESSING"
112 PRINT "SUBSTITUTION BASED ON ENTRY IN TABLE: 1=# 2=$ 3=% 4=&"
115 GOSUB 400                          'PAUSE TO SET UP TABLE
120 IF EOF(2) THEN GOTO 9000
130 LINE INPUT #2, X$
140 N= LEN(X$)
150 GOSUB 180
160 GOSUB 900
170 GOTO 120
180 '*****ADAPTIVE COMPRESSION SUBROUTINE*******
190 '* THIS ROUTINE PROCESSES RECORDS FROM X$  *
200 '* AND COMPRESSES WITH HUFFMAN CODES       *
210 '* USING O$ AS THE OUTPUT BUFFER.          *
220 '*********************************************
230 GOSUB 2120                         'PRINT HUFFMAN TABLE USED
235 I=1                                'RESET INDICES
240 FOR J= 1 TO N                      'STEP THRU RECORD
250 A$= MID$(X$,J,1)                   'EXTRACT A CHARACTER
260 FOR K = 1 TO 4                     'SETUP HUFFMAN LOOP
270 IF A$=P$(K) THEN GOSUB 350         'IS INPUT CHAR IN TABLE?
280 NEXT K                             'NO - TRY NEXT
290 IF M = 1 THEN 310                  'IS MATCH FLAG SET?
300 O$(I) = MID$(A$,1,1)               'NO-STUFF CHAR IN BUFFER
310 I=I+1                              'BUMP INPUT STRING INDEX
320 M=0                                'RESET MATCH FLAG
330 NEXT J                             'GO BACK FOR MORE
340 RETURN                             'DONE
350 M=1                                'SET CHAR MATCH FLAG
355 '*********************************************************
360 'INSERT COMPRESSION NOTATION IN OUTPUT BUFFER
365 V = K + 34                         'INDEX OUT TO SUBSTITUTE CHAR
370 O$(I)=CHR$(V)                      'INSERT SUBSTITUTION
380 P(K)= P(K) + 1                     'BUMP COUNT OF OCCURANCE
390 K = 4                              'FORCE END OF SEARCH
395 RETURN                             'GO BACK FOR MORE
400 DIM P$(4)                          'COMMON HUFFMAN CANDIDATES
410 DATA E,T,I,O
420 FOR I = 1 TO 4                     'SETUP CHARACTER TABLE
430 READ Z$                            'GET CHARACTER
440 P$(I) = Z$: NEXT I                 'AND STUFF INTO TABLE
450 RETURN                             'DONE - TABLE COMPLETE
```

Figure 2.60 ADAPTC.BAS program listing

```
900 '*****TALLY THE COMPRESSION COUNT & WRITE BUFFER******
910 '* DISPLAY BEFORE & AFTER RESULTS OF COMPRESSION      *
920 '* AND SHOW THE NET RESULTS OBTAINED BY EACH METHOD   *
930 '*************************************************************
931 N1=N1+N                          'TALLY INPUT CHAR COUNT
932 T=N-I+1                          'NET DIFFERENCE IN BUFFERS
936 T1=T1+T                          'SAVE COUNT FOR SUMMARY
940 FOR I=1 TO J-1
950 PRINT #3, O$(I);
960 NEXT I
965 PRINT #3, ""
966 GOSUB 2000                       'RESEQUENCE HUFFMAN TABLE
970 RETURN
2000 '*****RESEQUENCE & PRINT TABLE FOR ADAPTIVE COMPRESSION*****
2010 FOR J=1 TO 3                    'SETUP 1ST LOOP
2020 FOR K=J+1 TO 4                  'SETUP 2ND LOOP
2030 IF P(J) >= P(K) THEN 2100       'IS CURRENT ENTRY GREATER?
2040 TEMP= P(J)                      'NO-SAVE IN TEMP
2050 TEMP$= P$(J)                    'AND SAVE CHAR
2060 P(J)= P(K)                      'PICKUP GREATER COUNT
2070 P$(J)= P$(K)                    'AND ASSOC CHAR
2080 P(K)= TEMP                      'SWAP LESSER COUNT
2090 P$(K)= TEMP$                    'AND ASSOC CHAR
2100 NEXT K                          'FINISH 2ND LOOP
2110 NEXT J                          'FINISH 1ST LOOP
2115 RETURN                          'DONE-TABLE RESEQUENCED
2120 L= L + 1                        'REMEMBER LINE NO.
2130 PRINT "HUFFMAN TABLE USED FOR LINE";L;": ";
2140 FOR I=1 TO 4                    'SETUP PRINT TABLE LOOP
2150 PRINT P$(I);:PRINT P(I);        'PRINT CHAR AND COUNT
2160 NEXT I
2175 PRINT
2180 RETURN                          'DONE-TABLE PRINTED
9000 CLOSE: OPEN F$ FOR INPUT AS #2
9010 PRINT "FILE ";F$;" BEFORE SUBSTITUTION:"
9020 LINE INPUT #2,X$
9030 IF EOF(2) THEN 9060
9040 PRINT X$
9050 GOTO 9020
9060 PRINT X$:OPEN "ADAPTC.DAT" FOR INPUT AS #3
9070 PRINT "FILE ";F$;" AFTER SUBSTITUTION:"
9080 LINE INPUT #3,O$
9090 IF EOF(3) THEN 9998
9100 PRINT O$
9110 GOTO 9080
9998 PRINT O$
9999 CLOSE:END
```

Figure 2.60 (continued)

When the match flag is set, line 290 causes a branch to line 310, where the input string index is incremented by one, after which the match flag is reset to zero in line 320. If the match flag was not set, line 300 simply extracts one character from its appropriate position in the input record and places it into the output buffer.

Each time prior to a line of input being processed in this program, the subroutine call contained in line 230 will be invoked. This subroutine simply prints out the current status of the adaptive 'Huffman' compression table to include the character order and the frequency of occurrence of each character. Although this program was constructed to facilitate the visual observation of the changes in an adaptive compression table on a line-by-line basis, in developing an actual adaptive compression routine the tables would be subject to change on an individual character basis.

The actual resequencing of the adaptive compression table occurs in lines 2000 to 2115 of the program. This subroutine module sorts the characters in the adaptive compression table based upon their frequency of occurrence.

Figure 2.61 illustrates the sample execution of the ADAPTC.BAS program, with the status of the compression table displayed for each line of data in the file to be processed. In addition, the program displays the contents of the file prior to and after the substitution of characters from the previously defined 4-character character set. As an example of the operation of the program note that prior to line 1 being processed all entries in the compression table have a count of zero and the order of the entries is E, T, I and O.

The first line in the data file contains the string BEGIN, followed by many asterisks. Since the characters E and I will be replaced by the 'Huffman' codes # and %, after line 1 is processed the count for E and I should be one, while the adaptive compression table should be resequenced to account for the new frequency of occurrence. Examining the Huffman table used for

```
ENTER ASCII FILENAME. EG, ADAPT.DAT
? ADAPT.DAT
PATIENCE - INPUT PROCESSING
SUBSTITUTION BASED ON ENTRY IN TABLE: 1=# 2=$ 3=% 4=&
HUFFMAN TABLE USED FOR LINE 1 : E 0 T 0 I 0 0 0
HUFFMAN TABLE USED FOR LINE 2 : E 1 I 1 T 0 0 0
HUFFMAN TABLE USED FOR LINE 3 : I 5 0 5 T 3 E 2
HUFFMAN TABLE USED FOR LINE 4 : 0 9 E 7 T 5 I 5
HUFFMAN TABLE USED FOR LINE 5 : 0 19 I 13 T 11 E 11
FILE ADAPT.DAT BEFORE SUBSTITUTION:
1 BEGIN**************************************
2 OVATION OVATION FOR THE MUSICIAN
3 ENCORE ENCORE FOR THE ACTOR
4 0000000000 IIIIIIII TTTTTT EEEE
5 ****************************************END
FILE ADAPT.DAT AFTER SUBSTITUTION:
1 B#G%N**************************************
2 &VA%$&N &VA%$&N F&R %H# MUS$C$AN
3 &NC$R& &NC$R& F$R %H& AC%$R
4 ########## &&&&&&&&& %%%%%% $$$$
5 ****************************************&ND
Ok
```

Figure 2.61 Sample execution of ADAPTC.BAS program

Table 2.22 Adaptive compression table change

Initial table	
Character sequence	E T I O
Code substitution	# $ % &
After line 1 processed	
Character sequence	E I T O
Code substitution	# $ % &

line two in Figure 2.61, the reader will note that the count of E and I are set to 1, while the order of the characters in the table has been rearranged to take into consideration their new frequency of occurrence.

Examining line two in the ADAPT.DAT data file, the reader will note that OVATION contains four characters that can be substituted by the adaptive 'Huffman' code. Since the character O did not change its place in the compression table, the 'Huffman' code of & is substituted for that character. Next, the T in OVATION, which would have initially been replaced by the 'Huffman' code of $, is replaced by the 'Huffman' code of % since the adaptive table entries changed, which caused the 'Huffman' code substitutions to change. Table 2.22 summarizes the changes in the adaptive compression table prior to and after the first line of data in the input file is processed. As an exercise, the reader may wish to follow the code substitutions for the 4-character character set for the remaining lines in the ADAPT.DAT file that are processed by the ADAPTC.BAS program.

DATA COMPRESSION
Techniques and Applications, Hardware and Software Considerations
Second Edition

Gilbert Held
4-Degree Consulting, Macon, Georgia, USA
and
Thomas R. Marshall
(software author)

Are you spending more time and money on data storage and transmission than you need to? About 95 per cent of all data transmission consists of blanks, strings of spaces, numeric and alphabetic repetitions, not only buzzing through the airways but also embedded in a large number of databases. In this book the author shows how to increase the efficiency and cut the cost of data transmission and storage through the application of practical data compression routines.

Written as a no-nonsense, practical guide for implementing data compression, the techniques given in this book will prove invaluable whether your organization is large or small, whether you use a mainframe or microcomputer, and whether you are an end user or an equipment designer.

Also included are IBM PC programs and routines to compress and decompress data and to analyse the susceptibility of data to compression. The programs are now available on disk for those who prefer to save keying time and the introduction of errors. To obtain a copy of the disk please see the order form in the book.

Contents

Chapter One **Rationale and Utilization**
Logical Compression, Physical Compression, Compression Benefits, Terminology, Communications Applications, Data Compression and Information Transfer
Chapter Two **Data-Compression Techniques**
Null Suppression, Bit Mapping, Run Length, Half-Byte Packing, Diatomic Encoding, Pattern Substitution, Relative Encoding, Forms Mode Operation, Statistical Encoding, Adaptive Compression
Chapter Three **System Considerations and Data Analysis**
System Considerations, Data Analysis
Chapter Four **Software-Linkage Considerations**
Compression Routine Placement, Timing Considerations
Chapter Five **Using Compression-Performing Devices**
Asynchronous Data Compressors, Multifunctional Compression Devices
Appendices
Data Codes and Compression-Indicating Characters, DATANALYSIS Program Descriptions and Listings, SHRINK Program Descriptions and Listings
References
Further Reading
Index

ISBN 0-471-91280-8
9 780471 912804

JOHN WILEY & SONS
Chichester · New York · Brisbane · Toronto · Singapore