To the Graduate Council:

I am submitting herewith a thesis written by John Miller Bray entitled "An Analysis and Improvement of the Kermit File Transfer Protocol by Data Reduction." I have examined the final copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Robert W. Heller, Major Professor

We have read this thesis
and recommend its acceptance:

Accepted for the Council:

The Graduate School

AN ANALYSIS AND IMPROVEMENT OF THE KERMIT FILE TRANSFER PROTOCOL BY DATA REDUCTION

A Thesis

Presented for the

Master of Science

Degree

The University of Tennessee, Knoxville

John Miller Bray

August 1984

Copyright © John Miller Bray, 1984

All rights reserved

DEDICATION

This thesis is dedicated to my father who, had he lived, would have would have been so proud when it was finished.

ACKNOWLEDGEMENTS

During the course of this research several persons have been generous with their assistance. I am particularly indebted to my major professor, Professor Robert W. Heller, and to the other members of my committee, Professors David W. Straight and Terry Feagin. I want to thank my friend, Richard O. Whitehouse, who gave his time to proof read this work and--most importantly--who was a friend every time that I needed one. I am indebted to Ann Wayburn for her assistance in collecting the files for analysis. Without her help, the task would have been much harder.

I also want to recognize Dr. Isabelle H. Tipton, late of The University of Tennessee Physics Department. Were it not for her, I would have never arrived at the point to begin these studies.

ABSTRACT

The composition of over 200 files typical of those transferred between computers in a university environment is determined. The files are analyzed on a byte basis and frequencies of repeated characters and of strings of non-graphic characters are determined. Two operators which result in less protocol overhead for strings of non-graphic characters are developed and shown to operate properly. The relative savings in protocol overhead from these mechanisms is determined by file type: source, listing, object, text, and Huffman coded. Savings ranging to almost 50 percent would result from the combination of the non-graphic string operators and a repeat sequence operator. Most commonly, however, savings is in the range of 10 to 15 percent.

TABLE OF CONTENTS

CHAPTER		PAGE
I.	INTRODUCTION	1
II.	PROTOCOLS	6
III.	PROPOSED PROTOCOL CHANGES	19
IV.	BINDING ORDER	25
V.	ANALYSIS	31
VI.	CONCLUSIONS	42
LIST OF REFERENCES		45
APPENDIXES		47
	APPENDIX A	48
	APPENDIX B	51
	APPENDIX C	57
	APPENDIX D	60
VITA		75

LIST OF FIGURES

FIGURE		PAGE
A-1.	Example File Analysis Output	49
B-1.	Source File Data	52
B-2.	Text File Data	53
B-3.	Listing File Data	54
B-4.	Object File Data	55
B-5.	Huffman File Data	56
C-1.	Sample Repeat Processor Output Routine	58
C-2.	Sample Repeat Processor Input Routine	59
D-1.	File Analysis Program	61

CHAPTER I

INTRODUCTION

One of the most important problems facing computer scientists in the 1980s is the need for developing effective, fast, and convenient methods of communication between computers. The use of computing equipment has proliferated throughout society: the number of types of computers has rapidly increased and personal microcomputers have become common. Simultaneously, as the use of personal computers has increased, the need for convenient and accurate mechanisms for transfer of various types of data--including binary data, programs, and text--between computers has grown in a like manner.

In the past computing machinery was extremely expensive. As a consequence of this high cost, computing facilities generally consisted of large mainframe computers capable of concurrently servicing multiple users. These large machines, with their ancillary equipment, were housed in a central site, and connected by various means to multiple terminals which served as user interface devices.

Currently, these terminals are rapidly being replaced by microcomputers as the cost of hardware decreases and the power of microcomputers increases. These microcomputers combine almost all the resources of the mainframe

machines in one location. Often the currently available microcomputers are also used as work stations to prepare data of various types for processing on larger, mainframe machines. It has become essential to have the capability to transfer this data from the smaller machine to the larger machine for storage or computation, then send the results back to the microcomputer.

In the past the major medium for transfer of data between computers was magnetic tape. While magnetic tape has the advantage of holding large amounts of data (often twenty megabytes or more), it is generally a non-interactive medium--the tape must be mounted on the tape drive by an operator, data must be recorded on the tape, an operator must then dismount the tape, it must be returned to the user, and finally, the user must arrange to have the tape sent to the other computer site. While non-interactive, and somewhat clumsy, this remains a good mechanism for transferring very large amounts of data between computers.

Three problems with the tape method of data transfer are apparent. One is that because of the cost few, if any, microcomputers possess any form of 9-track tape drive equipment. Secondly, the tape recording formats used by various manufacturers are often incompatible, even though an ANSI standard for 9-track tapes has existed for some time. Even different machines built by the same

manufacturer often use incompatible tape formats, e.g.: the Digital Equipment Corporation's DECsystem 10 and VAX tape formats. The third problem is that typically a microcomputer user will wish to send to or receive from the mainframe or mini-computer relatively small amounts of data (normally much less than one megabyte--usually 10 to 100 kilobytes). Magnetic tapes are not efficient for this type of data transfer. With the exception of the rare case where very large amounts of data must be transferred, a completely interactive method of data transfer is highly desirable, if only for the speed and the relative ease of such transfers as compared to tape transfers. Often the data is transferred as part of a cycle of sending, computing on the other machine, reviewing the results, and sending the data again. In this situation, only an interactive method of data transfer suffices.

As an alternative to the magnetic tape method of data transfer, data may be transferred between machines by telephone or dedicated lines using modems at each end. In the more common general case considered here, the data is transmitted in an asynchronous bit-serial form and reconstituted into bytes or words at the receiving end of the transfer. This method is commonly available on microcomputers, and readily lends itself to interactive techniques. The main disadvantage to asynchronous bit-serial data transfer is that for very large amounts of data, it

may be much slower than the magnetic tape method. Most modems in general use operate at 1200 bits per second (bps) or less. Dedicated lines which may allow higher data transfer rates are rarely longer than a few thousand meters, and most commonly operate at speeds less than 9600 bits per second. Consequently a user may spend a significant amount of time waiting for a data transfer to complete.

For communications utilizing telephone lines, the costs of data transfer are directly related to the time required for the transfer. Similarly, connect time is often a large proportion of use charges in time-sharing installations. In an asynchronous bit-serial transfer, any reduction in the actual number of bits necessarily transmitted from one machine to the other will reduce the time needed for the transfer, and the connection costs of the transfer. While reduction of the number of bits transferred between machines is important, the integrity of the data must be protected. Errors must be detected and corrected, in order for the data to be reconsitituted correctly at the receiving end of the transfer.

Various techniques have been developed to compress data in such a manner that it can be later restored to the original configuration (Greenlaw, 1981 and DeMaine, 1972). As the Kermit file transfer protocol (Da Cruz, 1983) deals exclusively with byte oriented data, several mechanisms

are developed here which will function within that protocol to compress the number of bits necessary for transfer and later reconstitution.

While there has been interest for some years in data compression, little is known about the actual contents of typical files transferred between computers in a university environment. Most data compression methods, including the ones investigated here, are quite sensitive to the make-up of the data. The contents of a wide variety of actual data files used in a university environment have been investigated. The efficacy of several data compression mechanisms has been investigated for use within the Kermit protocol in a typical university setting. The results of this investigation are used as the basis of recommended changes to the Kermit protocol.

CHAPTER II

PROTOCOLS

Protocols form the basis for data transmission from one computer to another. In its simplest form a protocol is an agreement between the computers involved that data will be transmitted between the machines in a particular manner. Tanenbaum (1981) defines a protocol as "The rules and conventions used in this conversation are collectively known as the . . . protocol . . . ." McNamara (1977) indicates that "A protocol is basically a set of rules for operating a communication system." He further indicates that a protocol should solve problems involved in framing, error control, sequence control, transparency, line control, special cases, and timeout and startup control (McNamara, 1977).

There are three basic types of protocols used for non-local communications systems. They are the character-oriented, byte count-oriented, and bit-oriented protocols. Each of these types has different methods of framing data, and of making the data transparent to the framing. Local area networks (LANs) are used for short range computer communication, usually over distances of one kilometer or less and generally using directly wired connections between the computers involved. As this investigation is concerned with longer distance data

transfers, primarily over telephone lines using data communication equipment, local area networks will not be discussed here.

Character-Oriented Protocols

In a character-oriented protocol, all information is considered to be in fixed size groups of bits, usually of eight bits (one byte). Special characters are used to indicate the framing of a message or packet of data. The ARPANET data link layer protocol is an example of this technique. Framing is done with a three character sequence of Synchronize (SYN), Data Link Escape (DLE), and Start of Text (STX) characters. In order that this framing information be unique to the start of a packet, this particular sequence of characters is not allowed within the data portion of the packet. If a DLE SYN sequence is contained in the data to be transmitted, an extra DLE is inserted into the data stream on transmission, and deleted on reception. This technique is called character stuffing. Consequently the character-oriented protocols are often called character stuffing protocols. The International Business Machines (IBM) binary synchronous communication protocol (BISYNC) (IBM, 1970) is another example of this type of protocol. In the BISYNC protocol, the STX (start of text) character is used to indicate the beginning of a data sequence, and either the ETB (end of

transmission block) character to indicate the termination of the message or the ETX (end of text) character to terminate all messages. Unlike the ARPANET protocol, the header which precedes the message is not defined in the BISYNC protocol; it is defined by the network in which this protocol is to be used.

Byte Count-Oriented Protocols

Byte count-oriented protocols are often more precisely defined than the BISYNC protocol. In byte count protocols each message starts with a header, which at a minimum defines the number of data bytes to follow and which begins with a specific character to indicate the start of the packet. This header is then followed by the defined number of characters. Some type of block check character then follows the data. Byte count protocols normally include message sequencing information in the header, and may also include historical information indicative of the messages received correctly up to that point. These protocols depend almost exclusively on the byte count for framing information. If the byte count is incorrect due to an error condition, these protocols may loose framing synchronization. As a consequence of this, these protocols often define a second block check character specifically for the header, which is primarily used to validate the byte count.

The Digital Data Communication Message Protocol (DDCMP), developed by Digital Equipment Corporation (DEC, DIGITAL), is a example of the byte count-oriented protocols. This protocol uses the SOH (start of heading) character to indicate the beginning of a message. The header includes not only a count of data bytes in the packet, but also sequencing information, acknowledgement of prior packets received correctly, a header block check character, and optionally the address of the intended recipient for use on multi-drop lines (Tanenbaum, 1981, McNamara, 1977).

Bit-Oriented Protocols

The third major type of protocol is the bit-oriented, or bit stuffing, protocol. These protocols are considered bit-oriented because the unit of information is a bit, as opposed to the byte-oriented protocols discussed above. In the bit-oriented protocols the length of the data may be an arbitrary number of bits. Packet headers are usually of a predefined length, as are block check characters and other information following the data portion of the packet. The beginning of a frame of information is indicated by a specific bit pattern similar to the SOH used in the DDCMP byte count protocol. However, in a bit-oriented protocol, this one particular pattern may not occur in the data actually transmitted to the remote machine. Its

occurrence at any point in the data stream received by the remote machine is taken as a start of packet, and the preceding bits as the block check for the prior frame.

When the pattern used as a start of packet marker is part of the data bits to be transmitted to the remote machine, additional bits are inserted into the data stream before transmission in order to make the pattern of transmitted data bits different from the start of packet pattern. The receiving machine then deletes these bits as the data is received in order to restore the data to its original form. This process, known as bit-stuffing, is the mechanism which renders data transparent to the framing of packets in these protocols. There are several bit-oriented protocols. Two of the better known protocols are the IBM Synchronous Data Link Control (SDLC) protocol and the International Standards Organization's (ISO) adaptation of SDLC, High-level Data Link Control (HDLC) protocol.

Problems With Standard Protocols

All of these types of protocols are used on various mainframe and mini-computers. However, there are problems with all of them when used with personal computers. Most of them make assumptions about the host machines on both ends that may be unwarranted when applied to personal computers.

The first problem is the assumption that both machines are capable of continuous full duplex communication. Most current personal computers however, exclusively use polling techniques in communications port driver software, and have little to no interrupt capability included in the hardware configuration. Consequently, the disk controller is treated as another series of ports which must be polled for command signals or data. During disk accesses the personal computer must continuously poll the disk controller ports in order to determine if a character is available from the disk controller, or if the controller is ready for another character. In some cases, the controller may even stop the central processing unit of the computer until it is ready to send or receive another character. During these times, the personal computer is neither able to receive data from the communications port without data loss, nor to transmit data to the communications port. Used with any of the above communication protocols, a personal computer will always loose some portion of any data received by the communications port during its disk accesses.

A second problem with these mainframe oriented protocols is that there is an implicit assumption of transparency, both in any networks used between the machines and in the front end processors of the machine themselves. This is a reasonable assumption when the connection

between the machines is directly hard-wired and is to communication ports designed for such communication. This is generally not the situation for personal computer users. In almost all cases, the personal computer must be connected to the mainframe computer via a path designed for terminal to mainframe communication. In effect, the personal computer must appear as a virtual terminal to the larger machine, and must cope with the assumptions inherent in terminal communications to mainframe computers. Common assumptions about such terminal communications are: very low speed data input; fixed length data only (seven bits is usual); and predefinition of the functions of certain non-graphic control characters.

This path between a personal computer and the remote host may include a connecting network designed for terminal to multiple host communication. Such networks often support interrupt characters to enable a user to disconnect from one remote host and establish a connection with another. A character sent as data during a protocol data transfer, but which was a network disconnect character, would immediately terminate the connection to the remote host, with possible consequent loss of all transferred data. Concomitantly, the accessible input ports to the remote host machine are usually designed for terminal communication. These ports are in some cases designed in a manner that does not allow any control over the

interpretation of certain characters as commands. The IBM 370-series machines are an example of this problem. Certain characters are interpreted by the input/output preprocessor where there is no reasonable method to modify or eliminate this interpretation.

Microcomputer Protocols

There are several protocols which are designed specifically for personal computer use. They are all of the stop and wait type, which in effect are half-duplex protocols. In stop and wait protocols the sending machine must wait for the receiving machine to acknowlege the last packet before it may send the next packet. This required wait condition may be used by the receiving machine to stop any data transmissions during disk access periods, by sending the acknowlege message only after the access has occurred. In this way no data is lost by the personal computer.

Modem Protocol

The most commonly used protocol of this type is the Modem protocol designed by Ward Christiansen in the mid-1970's for use between two machines utilizing the Digital Research CP/M operating system. This protocol is essentially a byte count protocol, but with an implicit, fixed byte count. All packets contain 128 bytes of data, which

is one CP/M logical record. Because it is fixed, the byte count is not contained in the packet header information. The header consists of a SOH, the packet number modulo 256, and the one's complement of the packet number. Exactly 128 bytes of data follow. The trailing block check character may be either a simple checksum, or optionally a two-byte cyclic redundancy check (CRC) based on the CRC-16 algorithm. This protocol is of the stop and wait variety, utilizing single characters as the acknowlege (ACK) or negative acknowlege (NAK) return packets.

While the Modem protocol solves the problem of polled communication ports, it does not address the problem of the non-transparency of terminal networks or of the front end processors of remote hosts. Unfortunately, the sequence number, the sequence number's one's complement, and the block check character may have any value between 00H and 0FFH. Terminal networks which implement a flow control mechanism, such as Xon/Xoff control, immediately interfere with the data transfer when the sequence number, its one's complement or the block check character is a flow control character. A similar problem can occur when any of these is a character used in the control of the front end processor of the remote machine, or for control of the network. Because the Modem protocol was designed for direct communication between CP/M-based machines in which the program has complete control of the

communications port, it does not manipulate the data within the packet in any way. Therefore, any characters in the data which happen to be utilized as control characters by either the network or the remote host will probably not be transferred correctly, and may actually effect changes in the network or the remote host's front end processor.

The network used at The University of Tennessee at Knoxville is an example of the problem with flow control. The network is designed to connect terminals to the various remote hosts. In order that network buffers not interfere with flow control of data to the terminals, the network itself implements Xon/Xoff flow control, and does not pass either of these characters to the remote host. A Modem protocol data transfer using this network does not continue beyond the eighteenth packet as the sequence number of the nineteenth packet is absorbed by the network flow control mechanism. Similarly, two consecutive SO (shift out, control-N) characters are used as a network disconnect command to break the connection between a terminal and the remote host. The Modem protocol would allow these characters in packet data fields, resulting in disconnection and loss of the data transferred.

Kermit Protocol

The Kermit (Da Cruz, 1983) protocol is intended to circumvent these problems. This protocol was recently designed at Columbia University and is currently rapidly developing as a major personal computer to remote host protocol. In this protocol only graphic (printable) characters are transmitted in a packet as header, data, or block check characters, with the exception of a single character used as a start of packet marker. This is a byte count protocol explicitly based on the ASCII character code, and dealing indirectly with the IBM EBCDIC character code. One of the unique features of this protocol is that almost any part of it may be dynamically configured at the beginning of a data transfer. Each machine may define its own set of parameters which the other machine uses for packets sent to that machine. This configuration capability to some extent includes the single non-graphic start of packet character. The length of packets, quoting characters used to mark a non-graphic character converted to a graphic character in the data portion of the packet, type of block check character or characters, fill characters, and end of line terminators, if any, are configurable.

Sequence numbers and other packet header information, along with trailing block check characters are designed to be graphic characters. Any non-graphic characters in the

data itself are converted to a graphic character for transmission and restored to the original value when received. An attempt is also made to compress the size of the actual data being transmitted, by reducing a repeated sequence of a character to a special escape indicator, a repeat count, and the character.

The conversion of non-graphic data characters to printable characters necessarily involves an amount of overhead in the packets. Currently, each character is individually treated in the process of making it printable. For the two cases where a character is either below the ASCII printable range or above the printable range, a "quote" character is inserted into the data stream preceding the character, and the character is then modified to make it printable. Separate quote characters are used for the two cases. It is these quote characters which account for the overhead in the data portion of the packets. Additional overhead is necessarily incurred in the header and block check portions of the packet.

The current version of the Kermit protocol defines a repeat sequence for strings of repeated characters. Utilizing this mechanism, a string of repeated characters is replaced by a sequence consisting of a repeat escape flag character, defaulting to tilde (~), a repeat count transformed by the CHAR function (20H added to the count), followed by any appropriate quotes for the character and

lastly the character. Minimally, a repeated string must be at least four characters long before any savings accrue. In the worst case, where the character is in the range of 80H to 9FH, the string must be at least six characters long before any savings in data length occur, as both of the individual quote characters are required.

For data containing non-trivial amounts of non-graphic characters, overhead in the Kermit protocol may range to over 50 percent of the characters in the data portion of the packets. Any attempt at reducing the overhead incurred by these quoting mechanisms, or at reducing the size of the transmitted data itself by utilizing repeat sequences, will necessarily be greatly affected by the actual characteristics of the data.

CHAPTER III

PROPOSED PROTOCOL CHANGES

The Kermit protocol was designed such that it would be essentially a "generic" protocol, and therefore usable between most machines. The primary feature which enables this generality is that only one non-graphic (non-printing) character needs to be used--the start of packet character. This necessitates that all non-graphic characters be converted to graphic characters before transmission, and restored to the original value after reception. While this is a quite satisfactory solution to the problem of non-transparency in either networks or machine front end processors, it is also a primary source of overhead in the packets themselves.

A second significant source of overhead is the packet headers and block check characters. The Kermit protocol was originally designed to function well with the Columbia University DECsystem-20 machine. The DECsystem-20 and the DECsystem-10 machines have a limited capability for high speed input from terminal ports--both in input buffer size and also in that each input character interrupts the central processing unit. As a consequence of this, the Kermit protocol was designed to use packets of a limited size in order to minimize the periods of continuous high speed input. The Kermit protocol engenders a higher level of

overhead than protocols which allow much larger packet sizes because, with small packet sizes, the overhead in headers and block check characters is a larger percentage of the entire packet.

The combination of overhead due to the conversion to graphic characters, and to the smaller packet size results in data transfers which are significantly slower than similar stop and wait protocols such as the Modem protocol. In non-definitive tests conducted by this investigator, Kermit data transfers ranged to as much as twice the time for similar Modem protocol transfers. Small packet size is deeply ingrained in the Kermit protocol, and as such is not easily modified without a major protocol re-definition. Consequently that source of overhead will not be addressed here.

The overhead due to the conversion of non-graphic characters to graphic equivalents however is amenable to reduction without a major re-definition of the protocol. The current Kermit protocol uses two methods for rendering a data character printable. One is used for characters which fall in the range below that of ASCII graphic characters--hexadecimal (H) character values between 00H and 1FH. The other mechanism is used for characters above the range of printable ASCII characters, those which have the high-order bit set, hexadecimal values from 80H to 0FFH.

Characters below the graphic range, less than 20H, are converted to their graphic equivalents by turning on the "control" bit. (A control character is exclusive ORed with 40H.) A 00H becomes 40H; 01H becomes 41H. The reverse conversion is identical--41H is exclusive ORed with 40H to become 01H. Each time a character is converted to an equivalent graphic, a "quote" character is inserted into the data stream preceding it. It is the insertion of this quote character, defaulting to '#', which creates the overhead for control case characters in the packet data stream. One quote character is inserted for each character converted to the graphic range. Several such characters in a row in the data stream result in an extra quote character for each such character converted.

Similarly, characters which fall above the ASCII graphic range are also converted to a graphic equivalent. In this case, 80H is subtracted from the original character, and a different quote character, defaulting to '&', is inserted into the data stream preceding it. The receiving machine deletes the '&' from the data, and adds 80H to the character to return it to the original value.

A problem arises for characters which fall into the range of 80H to 9FH; the converted character (00H - 1FH) is not an ASCII graphic character. In this case, the control character quoting mechanism is applied to the

resulting character. In this way a character which had the original value of 81H becomes successively '&<01H>' then '&#A', with two quoting characters preceding it. This method doubles the overhead for this particular type of character.

In pathological cases, where data is largely non-graphic, the overhead may be 66 percent of the data portion of the packet. The following notation is used in the example below: upper case characters (A) represent values 80H to FFH. Lower case characters (a) represent characters in the range 00H to 7FH. Characters which, if taken as normal seven-bit ASCII characters and ignoring the high order bit, would be control characters are underlined. A pathological string to the current protocol is:

ABCDEFGH

The current quoting operators triple its length to:

&#A &#B &#C &#D &#E &#F &#G &#H

Clearly, quoting each character individually results in a large amount of overhead for files which have more than a small proportion of non-graphic characters. A method of deriving the desired result--only graphic characters in the data packets--without requiring that each character be quoted individually would result in less overhead and in less time required for the data transfer.

Protocol Modifications

It is proposed that the protocol be modified to include three more quoting characters which would be used to allow the mass quoting of strings of characters. These string quoting mechanisms would function as shift-out/shift-in commands in the data stream. In this system, the presence of one of these shift characters in the data stream would flag the beginning of a string of characters which had been converted to graphic equivalents. Individual characters within the string would not be separately marked as converted. The string would be defined to continue until the next occurrence of the shift character.

There would be two string shift quoting characters, one for characters below the graphic range ('|', 7CH), and one for characters above the graphic range ('?', 60H). A string of length n > 2 would result in a net transmitted character savings of of n - 2. The functionality of this system is in files with non-trivial numbers of strings of length two characters or greater, which are not graphic.

A third quoting character is needed in this system due to the effect of string shift quoting characters above the graphic range. The shifting of characters from above the graphic range can result in any character, either in the control case or the graphic range of characters. The third new quoting character (`, 60H) would be used as an escape quote, such that any character following it is no

longer treated as a quoting character. This eliminates the problem of a converted character being one of the quoting characters, either individual or shift quoting. This character would be used to turn off the quoting power of a quote character which should be treated as a normal data character. In this way the normal individual character quoting and also control case shift quoting mechanisms may be used within a larger string of characters converted from above the graphic range.

Precedence, or binding order, based on individual quoting of characters, is described in the Kermit Protocol Manual (Da Cruz, 1983). Additional precedences for the proposed new shift-out/shift-in operators and the escape quoting operator have been added to the original Kermit binding order. The new binding order is discussed in Chapter IV.

CHAPTER IV

BINDING ORDER

In order to implement the proposed string shifting quoting operators and the escape quote operator, a new binding order for the quoting operators must be defined. The term "binding order" is used here in order to maintain consistency with the Kermit Protocol Manual, Fourth Edition (Da Cruz, 1983).

Binding order is used to denote the relative precedence of the quoting operators, and is a parallel concept to the operator precedence definitions used in most computing languages. The closeness of operator binding in a protocol is equivalent to relative operator precedence in a computer language. In a functioning protocol this means that in cases where more than one quoting operator is applied to a character, the more closely bound (higher precedence) is applied first.

Current Kermit Binding Order

The current Kermit protocol defines the three quoting operators: '#', '&', and '~' for control characters, characters with the high bit set, and for repeated characters, respectively. The binding order is from the most closely bound '#' operator to the least closely bound '~' operator.

Using this binding order, the quoting operators in the transmitted data stream bear a fixed relationship. The least closely bound operator ('~') is inserted first into the transmitted data stream, and the most closely bound operator last, just prior to the resultant character. An 82H would become &#B. A repeated sequence of such characters would become ~n&#B, where n is the repeat count transformed by the CHAR function (x = x+32) to give a graphic character. It should be noted that this places an upper limit on the lengths of repeat sequences representable by a single repeat operator if the result of the CHAR function is to be in the ASCII graphic range.

Proposed Binding Order

A new binding order must be implemented which includes the string quoting operators and the escape quote operator ('`'). The following order is proposed: Strongest binding (highest precedence) is given to the escape quote operator. The next strongest binding is given to the single control character operator. Third in binding order is the single high bit operator. The control string shift operator ('|') is fourth in binding, followed by the high bit string operator ('?'). The least bound operator remains the repeat sequence operator. From most closely to least closely bound they are:

most `

#

&

|

?

least ~

There are two results of the new additions to the binding order. One is the obvious change made to the binding order itself. The second is that operators may be in effect when a character is processed, but not immediately preceed it in the the data stream. This is, of course, the intended result of the new string shift operators. These two operators are applied differently for transmission and reception. On reception, they are only effective after more closely bound, explicitly included operators have been applied. On transmission, any string operators are applied first, before any other operators are effective. The result is that operators are applied in data stream order on transmission, and reverse data stream order on reception.

Several examples of the string quoting operators are shown below. The same notation as in Chapter III is used: Upper case characters (A) represent values of 80H to FFH. Lower case characters (a) represent characters in the range 00H to 7FH. Characters which, if taken as normal seven-bit ASCII and ignoring the high order bit, would

be control characters are underlined. Spaces are shown in the strings for clarity, but would not be in the strings as transmitted or received.

In this example single characters are quoted, as in the current Kermit protocol. The string

a b c d E f G

is converted to the following string for transmission:

a b #c d &e f &#g

and restored to the original form upon reception. In the second example, a sequence containing repeated characters but no non-repeated strings of non-graphic characters would be transformed from:

a b c d EEEE f GGGGG

to the following string for transmission:

a b c d ~$&e f ~%&#g

The string quoting operators would operate on strings as shown below. The string:

A B C D E F G H I I I I I J K L M n o p

is converted to the string:

? a b c d | e f g h ~%i j k | l m ? n o p

The original 20 character string expands to 22 characters under the string shift operators, about a 9 percent increase. However, using the current protocol, quoting each character individually would result in the following 34 character string:

a b c d &#d &#f &#g &#h ~%&#i &#j &#k &l &m n o p

This is a 41 percent increase in transmitted string length over the original string, and significantly longer than the string processed under the string shift operators.

It is important to note that these operators remain transparent to the packet driver routines in a Kermit program in the same manner that the currently defined operators are transparent. Quoting operators in both the currently defined protocol and in the proposed changes are the lowest level of the protocol, preprocessing the data before it is packetized for transmission, and post-processing it after the packets have been received.

Implementation

The processing routines used to implement these operators would be somewhat more complicated than those required for the standard Kermit protocol as it is currently defined. The processors used in the transmission section must look ahead in the input file to ascertain if a string is beginning which would be long enough

to use the string shift operators. Routines in both the transmission and reception sections must determine if the character currently being processed is within one or more on-going string shift sequences. Routines in both sections must also add or delete the escape quoting operator if a processed character is one of the quoting characters. One might expect that globally defined flags would be used to indicate that a string shift operator was in effect. The look ahead and flag system would operate in a similar manner to the one for repeated character processing defined in the current Kermit protocol. Because of this, little extra processing overhead would need to be added to that already implemented for repeated character sequences. Each transmission routine would apply any string shift operator or operators currently in effect before proceeding with its processing. On reception, string shift operators would be applied after the routine's normal processing.

Example C language functions for a repeated character processor utilizing the string shift operators are given in Figures C-1 and C-2, Appendix C. In order to clearly show the logic for the string shift operators, escape quoting is not included in the C language functions.

CHAPTER V

ANALYSIS

The most pervasive trait shared among data compression algorithms is dependence on the composition of the data itself. Two of the most common mechanisms used for data compression are replacing sequences of repeated characters with short sequences indicating repeated characters and various forms of the Huffman coding algorithm. Both of these are very dependent on the type of data to be compressed. The Kermit protocol repeat sequence compression mechanism (CUCCA, 1983) is typical of techniques used for compression of repeated characters. Its efficacy is proportional to the frequency of repeated strings, and to the length of the repeated strings. Clearly, if there are no sequences of repeated characters, then incoroprating a defined repeat sequence replacement will have little effect. Similarly, Huffman coding will result in little compression in cases where the data is distributed in a uniform or near-uniform manner. The two shift-in/shift-out mechanisms proposed here are dependent on the frequency and length of strings of characters both below and above the ASCII graphic range. Another compression technique is the set of algorithms developed by De Maine (De Maine, 1972). These also are quite specific for the type of data to be compressed, utilizing entirely different

methods for alphanumeric data from those used with digitized analog data.

Analysis Program

A program, COUNT.C, was developed to analyze the individual files. The C language source for this program is presented in Figure D-1, Appendix D. It should be noted that the program does not output lines of the distribution which contain only zero values. The program reads each byte of a file and creates a matrix of the frequencies of each value read. It further analyzes the frequencies of strings of control case (below the graphic range) characters by string length. A similar analysis is made for strings of bytes with values greater than 7FH (above the graphic range). The output of one analysis is shown in Figure A-1, Appendix A. Only lines containing non-zero values are shown.

Data Selection and Composition

The compositon of approximately two hundred files has been invesitgated. The files were selected to be typical of those transferred between personal computers and mini or mainframe computers in a university environment. The frequency of occurrence of byte values in these files was determined. An analysis of the frequencies of occurrence of strings of control case characters and characters with

values over 7F hexadecimal (H) was done by string length. The files used were collected from a variety of sources including: student source language programs, student listings files, a working Mumps language interpreter (O'Kane, 1983), various remote CP/M systems, and the investigator's personal files.

File Groupings

After collection, the files were grouped into several categories based on type. The groups used for final data reduction were: source files, text files, object files, lineprinter output files, and Huffman coded files from the public domain program Squeeze developed by Greenlaw (Greenlaw, 1981). The Huffman coded files were included as this mechanism has become the de facto standard for file compression under the CP/M operating system, and is currently used under other microcomputer operating systems as well. Personal computer generated object files were included in the analysis as these represent a significant portion of the files transferred between personal machines and the mainframes at The University of Tennessee. Personal computer users often utilize the large storage capacity of the mainframe machines for archival storage and backup of files from the floppy disks of the smaller machines. Source, text, and lineprinter files were

included as these are perhaps the most frequently transferred groups of files in a student environment.

A condensed, more representative, subset of the original files was selected. The data generated by the COUNT.C program for this subset of the original group was entered into a commercial electronic spread-sheet program for final analysis by file type group. The analyses from this program are presented in Figures B-1, B-2, B-3, B-4, and B-5, Appendix B arranged by file type group.

Results

Source Files

The source files summarized in Figure B-1, Appendix B are perhaps, as a group, the most inconsistent. The total potential savings varies from 1 percent to 44 percent. The average of the potential savings due to a shift-in/shift-out mechanism for control case character quoting is about the same as the standard deviation for this mechanism. The standard deviation for the repeat mechanism (10.009646) is actually larger than the average itself (8.5207417).

There appears to be no clear relationship between the language in which the file is written and the savings. The last five, Fortran, files show the largest total savings (29 - 44 percent) but GLOBAL.FOR is the fifth least compressable at 2.6 percent. It is interesting that the

last five and GLOBAL.FOR were collected from the same source. Pascal files appear in the list with low (4.9 percent), medium (11.5 percent), and high (22.5 percent) compressability, as do Basic language files (1.1 to 22 percent).

There is no clear pattern of one of the mechanisms resulting in greater savings over the others. There are several instances in which files have almost the same total savings, but the savings derives from different mechanisms. GRADE.BAS would compress only about 1 percent from a repeated character system, but about 7 percent from a shift mechanism for control case characters. However, CYPHER.SNO shows almost the opposite trend. The largest average savings is from the repeat sequences (8.5 percent), with a much smaller average savings due to the contol case shift mechanism (3.4 percent). None of the files show any savings from the shift-in/shift-out mechansim for characters above the graphic range (values over 7FH), as would be expected. Overall for the source files, using a combination of all three mechanisms, the average savings would be about 12 percent.

Text Files

The results from the text files, shown in Figure B-2, Appendix B, are similar to the source files. There are no files which show savings due to the high order bit shift

mechanism. Only one file, however is a formatted output of the Wordstar (c) word processing program, and it does include bytes with values over 7FH. In general, these files achieve the most savings due to repeated character compression. One of the interesting aspects of this particular group of files is that they all exhibit relatively large proportions of space characters, from 15 to 25 percent, while having few repeated strings of spaces available for compression.

There are, however, at least two general types of text files represented. Several are very dense text, with little white space. These all fall into the very lowest savings group (less than 5 percent), represented by the five documentation files at the top of the list. Text files which have a larger amount of white space tend to fall into the group with the greatest savings, such as F1410.FIN with 23 percent savings. This file is a final examination for a lower level Fortran class. W2610.SCH, with 27 percent savings, is a schedule for a class, and largely white space. In general, files which show low control shift mechanism savings, and high repeated character savings are forms and open spaced documents.

Listing Files

The results of the listing files analysis, in Figure B-3, Appendix B, are rather interesting in that they show

clear grouping around the systems on which they were generated. The files from the DECsystem-10 are as a group the ones which are least amenable to savings using any of the mechanisms. The listing files from the IBM system, marked Assist, show the greatest savings of the three groups. The CP/M listing files fall between the other two groups. While there is quite a bit of variability within each group, there is a clear 5 to 6 percent gap between the groups. The Decsystem-10 files, with 4 to 6 percent savings, have no location column in the listing, and are essentially only a copy of the input source file, with maximum entabulation of each line. The CP/M assembler files, with 17 to 22 percent savings, have a location column followed by several spaces, and are highly entabulated beyond that point on each line. The IBM output files show the most savings, as much as 47 percent. These IBM files are generated in a fixed record length format, and padded with spaces to fill out any unused columns. No tab characters are used in the IBM files. It is these spaces which account for most of the savings achievable with these files. The file P1.LPO is 49 percent space characters, while NEW2.LPO is 57 percent spaces.

Object Files

The object file group, shown in Figure B-4, Appendix B, results in the least total savings as a group, all

exhibiting low to medium savings. This group is also more consistent as a group than the source and listing groups, showing a standard deviation (3.0984249) about one third of the average total savings, about 11 percent. The largest part of the savings arises from the shift mechanism for strings with values below 20H (5 percent). Object files generated on mainframe computers were not included in this group as these are seldom transferred via the Kermit protocol.

Huffman Coded Files

The Huffman coded files in Figure B-5, Appendix B exhibited the most consistent savings of any of the groups. The average total savings was almost 16 percent, with a standard deviation of 0.8907943, an order of magnitude lower. This is somewhat surprising, as the files are a mixed group, consisting of documentation, assembler source, IBM lineprinter output, and C language source files. Little to no savings were due to the repeat mechanism (0.0002275 percent) or to the control character shift mechanism (0.4506778 percent). Almost all the savings were due to the shift mechanism for bytes with values over 80H. It should be remembered that these files represent data which are no longer byte oriented, but have been converted to a bit stream. Therefore, there is a 0.5

probability that any given byte will appear to have a value above 80H.

While there is consistent moderate savings in this group, this may mask an interesting situation. The Squeeze programs were developed as a mechanism to decrease the required disk space needed for storage on floppy disks, and to reduce the time necessary for the transfer of files from CP/M system to CP/M system using the Modem protocol. Unfortunately, under the Kermit protocol, with it's attendant quoting of individual isolated characters with values over 80H, the squeezed files may take quite a bit longer than the unzqueezed versions even with the shift mechanisms implemented. In the file PROTECT.AQM there are a total of 8695 characters with values over 80H, 1932 isolated characters with values over 80H, but only 135 instances of strings of length five. The current Kermit protocol would add 8695 characters to the outgoing data stream for this file, which is only 16,128 characters long. Potential savings of high bit quoting characters in this file is 2359 characters, more than the isolated character count. Even though a large number of characters are saved by using the shift mechanism, the isolated characters alone would add almost 2000 characters to the size of the file in the out going data stream. The value of the high bit shift mechanism for squeezed files is clear. The

outgoing data streams for these files would be significantly larger without this mechanism.

Validity of the Data

The attempt was made to create a collection of files which would be as representative as possible of the file transmitted in a university environment. Although no definitive statement regarding the representativeness of this particular selection can be made, some general observations are pertinent.

The source files as a group are quite representative of the files transferred at the University of Tennessee at Knoxville. An informal survey was undertaken of Kermit users on the DECsystem-10. The mixture of files which were reported transferred using Kermit was similar to the group of source files selected for final analysis.

The text files are perhaps somewhat less representative of the actual mixture of files transferred using Kermit. There was only one file utilized in the final analysis group which was created by the Wordstar word processing editor. This probably represents some amount of skewing of the distribution of text files, as files produced by Wordstar are commonly transmitted to the University's laser printer for printing, although this is more commonly done with the ASCII file mode of Kermit,

which strips high order bits normally in the course of the transfer.

The group of listing files is perhaps the most representative of all the ASCII files examined. The mixture of files reported by Kermit users in the informal survey mentioned above was used as the basis of the final selection of files for this group.

The object files as a group may have resulted in the most skewing of the selection. All the files used are files generated for the 8080/Z-80 cpu's running in the CP/M operating system environment. Files for other microcomputers using other operating systems were unavailable for analysis. It is expected however, that similar results would be obtained for those machines, as the machine level instruction sets are similarly byte oriented.

The Huffman coded files appear to be quite representative. This is supported by the relatively small standard deviation of this group. The files were selected to represent a wide variety of types of files, but the resulting compression data was similar for all the files in the group.

CHAPTER VI

CONCLUSIONS

There were two primary purposes for this investigation. The first of these was to analyze the composition of files actually transferred between computers in a university environment using the Kermit protocol. The second was to develop data compression mechanisms which would decrease the amount of overhead necessary to quote non-graphic characters. This would then increase the effective speed of a Kermit protocol file transfer. A large number of files which are thought to be representative of those transferred in a university environment using the Kermit protocol were analyzed. Two compression mechanisms were developed. During the course of this investigation, a repeated character operator was developed to accompany the two string shift operators. However, the repeated character operator was simultaneously developed and implemented in the Kermit protocol by the original Kermit developers at Columbia University.

Analysis of the data files indicated that only a relatively small amount of data compression (5 to 15 percent) was feasable in most files. Few files contain any characters above the ASCII graphic range. The files which do contain characters above the ASCII graphic range are generally limited to object files and to Huffman coded

files. The Huffman coded files were the only group to show significant benefits from the string shift operator for characters having values above the normal ASCII range. Strings of control case characters are much less frequent than originally expected, with frequency assymptotically approaching zero as string length increases. The only clear exception to this is in isolated object files.

String shift operators were developed and shown to function properly for strings of characters which are either control case characters, or fall above the ASCII graphic range. The amount of data compression accomplished by these operators in conjunction with the repeat operator varies significantly with file composition, (from almost none to almost 50 percent).

One unexpected result of the analysis was that some computing systems and some people generate files which are initially somewhat compressed, by replacing spaces with tab characters. Data streams generated from these files allow little compression. Another result which was unanticipated was that there are few very long repeated character strings, although these repeated strings are more common than the non-graphic strings for which the string shift operators were designed.

The current Kermit protocol is subject to worst cases when a file contains significant amounts of non-graphic characters. The proposed changes reduce this problem for

files with strings of non-graphic characters, but do not eliminate the problem when file contains large numbers of isolated non-graphic characters separated by graphic characters.

Further Research

This investigation should be followed by an analysis of a much larger group of files very carefully selected to be representative of the files transferred outside the university environment. A significantly more sophisticated analysis program should be developed to account for the effect on the data stream of the specific operators being studied. The current version of the file analysis program does not deal adequately with these effects. It would be useful to investigate the differences in the data stream when characters above the graphic range are not quoted. The input preprocessors of many computers may be easily programmed to allow such characters. This would reduce the impact of the worst case where two quote operators are required.

A different approach to data compression well worth investigation is a design for the protocol which would allow the 128 characters above the graphic range to represent the 128 most common words in the file. This approach might generate the largest data stream savings of all the mechanisms.

LIST OF REFERENCES

LIST OF REFERENCES

Da Cruz, Frank and Catchings, William. Kermit Protocol Manual. Fourth Edition. New York: Columbia University Center for Computing Activities, 1983.

de Maine, P.A.D. The Integral Family of Reversible Compressors. State College: Computer Science Department, The Pennsylvania State University, 1972.

Greenlaw, Richard. Squeeze.C. Available in source and compiled form from most remote CP/M systems. 1981.

IBM. Systems Reference Library, General Information - Binary Synchronous Communications. Research Triangle Park: International Business Machines Corporation, 1970.

McNamara, John E. Technical Aspects of Data Communication. Bedford: Digital Press, 1977.

O'Kane, Kevin C. Mumps Interpreter. For further information contact O'Kane directly at University of Tennessee, Knoxville, Computer Science Department. 1983.

Tanenbaum, Andrew S. Computer Networks. Englewood Cliffs: Prentice-Hall, 1981.

APPENDIXES

APPENDIX A

SAMPLE FILE ANALYSIS OUTPUT

```
count        version 3.1     15 September 83

Character count for file B:CH2.WS
(control-Z used as eof.)

            Frequencies of characters by hex value

hex         0     1     2     3     4     5     6     7
value       8     9     A     B     C     D     E     F
-------------------------------------------------------
 8:         0     0   306     0     0    75     0     0
20:      2473     0     4     0     0     0     0    13
28:        17     2     0     0    10    25    56     5
30:         5    18     5     3     0     2     3     7
38:         4    10     0     0     0     0     0     4
40:         0    10    10    31    16     8     6     2
48:         8    25     0     3     9    18     9    12
50:        11     0     4    21    48     5     0     3
58:         2     2     0     2     0     2     1     0
60:         0   815   169   590   210   968   140    76
68:       460   716     8    52   292   303   668   879
70:       285    22   661   397   986   241    62   107
78:        14    50     8     0     0     0     0     0
88:         0     0     5     0     0   236     0     0
A0:       753     0     2     0     0     0     0     0
A8:         0    15     0     0    96     0    87     0
B0:         0     0     0     0     0     0     1     0
B8:         2     0     0     1     0     0     0     0
C0:         0     5     2     1     0     0     1     0
C8:         3     1     0     0     0     5     1     2
D0:         1     0     2     0     1     0     0     0
D8:         2     0     0     0     0     0     0     0
E0:         0   104     2     4   154   422   108    51
E8:        42     0     0    42    96    30   148    77
F0:         8     0   128   262   183     0     0    11
F8:         2    85     0     0     0     0     0     0

TOTAL CHARACTERS:  15678
```

Figure A-1. Example file analysis output.

Control character frequencies by string length

(0xff value is for strings longer than 0xff)

hex value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0:	0	231	17	0	1	0	16	0
8:	2	0	0	0	0	0	0	0

Total number of control characters: 381
amount of potential savings: 78, percent:

Frequencies of chracters >= 0x80 by string length

(0xff value is for strings longer than 0xff)

hex value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0:	0	1719	658	47	2	0	0	0

Total number of high-bit characters: 3184
amount of potential savings: 51, percent:
total potential saving: 129, percent:

Repeated character frequencies by string length

(0xff value is for strings longer than 0xff)

hex value	0 8	1 9	2 A	3 B	4 C	5 D	6 E	7 F
0:	0	0	339	5	3	18	1	0
18:	1	1	0	0	0	0	0	0

Total number of repeated characters: 850
amount of potential savings: 85, percent:
total potential saving: 214, total percent:

Figure A-1. (continued)

APPENDIX B

SUMMARY OF ANALYSIS DATA

FROM SPREADSHEET

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	Name	Type	Chars.	Control	> 80H	Repeated	Savings	% Savings	Savings	% Savings	Savings	% Savings	Savings	% Savings
2	File	File	Total	Total	Total	Total	Control	Control	High bit	High bit	Repeated	Repeated	Total	Total
3	Abbsutl.bas	basic	9422	456	0	747	0	0	0	0	104	1.1037996	104	1.1037996
4	Clbel.bas	10/bas	10404	468	0	808	0	0	0	0	204	1.9238023	204	1.9238023
5	Dishowl6.asm	cpm/asm	12246	1902	0	1093	260	2.1231423	0	0	0	0	260	2.1231423
6	Zcpr1.asm	cpm/asm	51450	7880	0	3911	1149	2.2332362	0	0	160	0.3109815	1309	2.5442177
7	Global.for	10/fort	20105	2603	0	1200	514	2.556578	0	0	18	0.08953	532	2.6461079
8	Zcmd.asm	cpm/asm	45096	8883	0	5438	1263	1.9402114	0	0	670	1.0292491	1933	2.9694605
9	Ccsrom.asm	cpm/asm	41666	7317	0	3052	1238	2.9712475	0	0	13	0.0312085	1251	3.002448
10	Dve79.asm	cpm/asm	82730	12779	0	6358	1920	2.3200826	0	0	1343	1.4233531	3263	3.9441557
11	Purple.sno	snobol	1905	267	0	125	70	3.6745407	0	0	9	0.4724409	79	4.1469816
12	Cnvtap.pas	10/pascal	1625	153	0	172	51	3.1384615	0	0	28	1.7230769	79	4.8615385
13	CCC.asm	cpm/asm	24346	4295	0	1124	1050	4.3128235	0	0	165	0.6777294	1215	4.9995529
14	Dv-v79.asm	cpm/asm	43778	9757	0	2431	1787	4.081959	0	0	400	0.9137009	2187	4.9956599
15	Xmodem73.asm	cpm/asm	59786	9873	0	6550	1404	2.4821845	0	0	1746	2.9204162	3230	5.4026026
16	Dve2-15a.asm	cpm/asm	60670	9259	0	6161	1379	2.272203	0	0	2063	3.399242	3442	5.671445
17	Icky.sno	snobol	2610	327	0	230	64	2.4521073	0	0	93	3.5632184	157	6.0153257
18	Loan.pas	10/pascal	1775	98	0	267	8	0.4507042	0	0	101	5.6901408	109	6.1408451
19	Coins.pas	10/pascal	1430	125	0	194	37	2.5874126	0	0	33	3.8461538	92	6.4335664
20	DB.3	360 asmbr.	58320	7227	0	7795	994	1.7043896	0	0	2838	4.8642551	3832	6.5706447
21	Stdlibl.c	C	7975	1267	0	645	483	6.0544263	0	0	43	0.539185	526	6.5956113
22	Zilch.sno	snobol	4860	485	0	765	92	1.8930041	0	0	235	4.8353909	327	6.7283951
23	Averag.pas	10/pascal	1080	110	0	175	29	2.6851852	0	0	44	4.0740741	73	6.7592593
24	Count.c	C	9046	1124	0	745	557	6.1574176	0	0	58	0.6411674	615	6.798585
25	Tri.pas	10/pascal	1870	179	0	315	41	2.1925134	0	0	101	5.4010695	142	7.5935829
26	DB.1	360 asmbr	21965	2085	0	3903	239	1.0880947	0	0	1511	6.8791259	1750	7.9672206
27	Linear.bas	10/bas	280	43	12	51	19	6.7857143	0	0	4	1.4285714	23	8.2142857
28	Substr.bas	10/bas	1125	130	30	170	70	6.2222222	0	0	24	2.1333333	94	8.3555556
29	DB.2	360 asmbr.	43200	6410	0	7615	494	1.1435185	0	0	3201	7.4097222	3695	8.5532407
30	Brode.bas	10/bas	475	99	21	90	36	7.5789474	0	0	5	1.0526316	41	8.6315789
31	Cypher.sno	snobol	3055	273	0	549	40	1.3711948	0	0	245	8.0196399	293	9.5908347
32	Kermit.c	C	31570	2912	0	5867	619	1.9607222	0	0	2457	7.7827051	3076	9.7434273
33	Mot4.pli	10/pli	26550	2687	0	4379	904	3.4048964	0	0	1844	6.9453861	2748	10.350282
34	mot1.pli	10/pli	17870	1873	0	2923	569	3.1841074	0	0	1285	7.1908226	1854	10.37493
35	Teldir.pli	10/pli	33195	3124	0	5731	1092	3.2896521	0	0	2412	7.2661545	3504	10.555807
36	Codd.pli	pl/i	43775	3882	0	15409	2094	3.2834183	0	0	5074	7.9560956	7168	11.239314
37	Span.pas	10/pascal	6780	731	0	1089	307	4.5280236	0	0	470	6.9321534	777	11.460177
38	Mmmm.bas	10/bas	295	69	13	71	30	10.169492	0	0	4	1.3559322	34	11.525424
39	Test.sno	snobol	600	38	0	144	2	0.3333333	0	0	68	11.333333	70	11.666667
40	Copy.pas	10/pascal	1720	193	0	359	25	1.4534884	0	0	178	10.348837	203	11.802326
41	P6670.pas	10/pascal	21755	1239	0	5034	84	0.3861181	0	0	2939	13.509538	3023	13.895656
42	Bubble.bas	10/bas	585	112	22	186	48	8.2051282	0	0	34	5.8119658	82	14.017094
43	Graph.for	10/fort	13630	847	0	3782	5	0.0366838	0	0	2069	15.179751	2074	15.216434
44	Smorl.for	10/fort	2410	159	0	696	2	0.0829876	0	0	402	16.680498	404	16.763485
45	Play.sno	snobol	4715	1210	134	1019	571	12.110286	0	0	255	5.4082715	826	17.518538
46	Drev.pli	10/pli	40030	2513	0	15083	213	0.4434728	0	0	8859	18.444722	9072	18.888195
47	Errmod.for	10/fort	4245	329	0	1444	6	0.1413428	0	0	777	18.775029	883	18.916372
48	Cpa.pas	10/pascal	18910	4086	559	4598	2138	11.411951	0	0	1824	9.6454791	3782	21.057641
49	Short.pas	10/pascal	14270	2573	0	2827	1474	10.329362	0	0	1628	11.408549	3102	21.737912
50	Check.bas	10/bas	7280	1005	192	2195	426	5.8516484	0	0	1175	16.14011	1601	21.991758
51	Hexfle.pas	10/pascal	14260	2872	0	2721	1722	12.075736	0	0	1490	10.448808	3212	22.524544
52	Libs.for	10/fort	6015	835	0	1768	134	2.2277639	0	0	1632	27.13217	1766	29.359933
53	Fcn.for	10/fort	11535	786	0	5136	22	0.1907239	0	0	3884	33.671435	3906	33.862159
54	Nmps.for	10/fort	70288	5705	0	28126	885	1.2591054	0	0	24050	34.216367	24935	35.475472
55	Parse.for	10/fort	32990	2682	0	14447	434	1.3155502	0	0	11859	35.947257	12293	37.262807
56	Dver.for	10/fort	12330	748	0	6445	35	0.2838605	0	0	5424	43.990268	5459	44.274128
57	>> AVERAGE	ROW <<	20557.741	2576.4074	18.203704	3394.9259	541.42593	3.3821313	0	0	1807.2963	8.5207417	2348.7222	11.902873
58	>> STD.	DEV. <<						3.2286958		0		10.009646		9.6430599

Figure B-1. Source file data.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	File	File	Total	Total	Total	Total	Control	Control	High bit	High bit	Repeated	Repeated	Total	Total
2	Name	Type	Chars.	Control	> 80H	Repeated	Savings	% Savings	Savings	% Savings	Savings	% Savings	Savings	% Savings
3	Sortmodm.doc	text	24579	1506	0	2914	130	0.5289068	0	0	107	0.435331	237	0.9642378
4	Redstick.doc	text	9716	444	0	559	97	0.9983532	0	0	0	0	97	0.9983532
5	Rbbsuse.doc	text	9663	414	0	672	76	0.7865052	0	0	24	0.2483701	100	1.0348753
6	Squeezer.doc	fat-ws	22840	1072	0	2703	176	0.7705779	0	0	254	1.1120841	430	1.882662
7	Bsort.doc	text	3126	184	0	94	59	1.887396	0	0	0	0	59	1.887396
8	Preprsl.txt	ws	3016	98	565	140	32	1.061008	0	0	49	1.6246684	81	2.6856764
9	Xmodem.inf	text	3727	201	0	351	73	1.9586799	0	0	48	1.2878991	121	3.246579
10	Batch.doc	text	35546	1770	0	3177	484	1.3616159	0	0	700	1.9692792	1184	3.3308952
11	Diszilog.doc	text	3255	254	0	211	120	3.6866359	0	0	41	1.2596006	161	4.9462366
12	Answer.txt	txt	11850	673	0	1361	130	1.0970464	0	0	501	4.2278481	631	5.3248945
13	Letter.jan	letter	1390	128	0	142	68	4.8920863	0	0	10	0.7194245	78	5.6115108
14	Janis.res	resume	1605	166	0	181	89	5.5451713	0	0	6	0.3738318	95	5.9190031
15	W2610.syl	syllabus	1915	164	0	221	73	3.8120104	0	0	42	2.1932115	115	6.0052219
16	Sed.hlp	text	4750	421	0	788	67	1.4105263	0	0	268	5.6421053	335	7.0526316
17	@.doc	text	4610	290	0	960	64	1.3882863	0	0	334	7.2451193	398	8.6334056
18	Ques.txt	txt	4095	473	0	397	310	7.5702076	0	0	47	1.1477411	357	8.7179487
19	F3520.t2	test	3895	702	0	271	394	10.115533	0	0	44	1.1296534	438	11.245186
20	Samp.swi	swit.ini	4160	271	0	711	6	0.1442308	0	0	520	12.5	526	12.644231
21	Track.hlp	text	8620	360	0	1932	44	0.5104408	0	0	1151	13.352668	1195	13.863109
22	S2610.rev	text	1960	302	0	519	158	8.0612245	0	0	119	6.0714286	277	14.132653
23	F1410.mp3	asgt.	2490	330	56	501	180	7.2289157	0	0	196	7.8714859	376	15.100402
24	Zcpr1.doc	ws/fmt	45229	2776	0	11829	1130	2.498397	0	0	5803	12.830264	6933	15.328661
25	S2610.t1	test	7085	869	0	1045	524	7.3959068	0	0	616	8.6944248	1140	16.090332
26	Xmodem.hlp	text	1150	71	0	311	25	2.173913	0	0	188	16.347826	213	18.521739
27	F1410.mid	test	4810	647	0	1304	291	6.049896	0	0	628	13.056133	919	19.106029
28	F1410.fin	test	8345	1124	0	2316	555	6.650689	0	0	1363	16.333134	1918	22.983823
29	W2610.ev1	text	2130	110	0	689	24	1.1267606	0	0	476	22.347418	500	23.474178
30	F1410.xtr	text	1375	207	0	389	129	9.3818182	0	0	225	16.363636	354	25.745455
31	Hand.ass	text	1040	254	0	316	149	14.326923	0	0	122	11.730769	271	26.057692
32	Hand2.ass	text	1075	243	0	368	140	13.023256	0	0	143	13.302326	283	26.325581
33	W2610.sch	schedule	1280	82	0	474	35	2.734375	0	0	316	24.6875	351	27.421875
34	F1410.rev	text	1160	97	0	473	11	0.9482759	0	0	327	28.189655	338	29.137931
35	>> AVERAGE	ROW <<	7546.4688	521.96875	19.40625	1197.4688	182.59375	4.097674	0	0	458.375	7.9467136	640.96875	12.044388
36	>> STD.	DEV. <<						3.8507069		0		7.9608546		9.1154869

Figure B-2. Text file data.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	FILE	file	total	total	total	total	control	control	high bit	high bit	repeated	repeated	total	total
2	NAME	type	chars.	control	> 80H	repeated	savings	% savings	savings	% savings	savings	% savings	savings	% savings
3	Newh.lst	dec/fort	12430	1786	0	2166	251	2.0193081	0	0	285	2.2928399	536	4.312148
4	Help20.prn	cpm/asm	62581	5737	0	19151	226	0.361132	0	0	4089	6.533932	4315	6.895064
5	Span.lst	dec/pasc	8830	774	0	2406	116	1.3137033	0	0	525	5.9456399	641	7.2593431
6	Newtry.lst	dec/fort	14490	2007	0	3033	331	2.284334	0	0	761	5.2518979	1092	7.5362319
7	Cpm.lst	dec/pasc	19070	1915	0	5623	280	1.4682748	0	0	1264	6.6282119	1544	8.0964866
8	Maxflo.lst	dec/pasc	20280	2964	0	6522	803	3.9595661	0	0	1604	7.9092702	2407	11.868836
9	Short.lst	dec/pasc	19885	2655	0	6384	667	3.3542872	0	0	1731	8.7050541	2398	12.059341
10	Du-v77.prn	cpm/asm	84013	9508	0	25332	83	0.0987942	0	0	14304	17.025936	14387	17.124731
11	Ccsrom.prn	cpm/asm	72577	7725	0	21804	122	0.1680973	0	0	12484	17.201042	12606	17.369139
12	Userpw37.lpt	cpm/asm	31668	2609	0	9299	34	0.1073639	0	0	5670	17.904509	5704	18.011873
13	Boot48.prn	cpm/asm	3407	397	0	1269	5	0.1467567	0	0	639	18.755503	644	18.90226
14	Findbd55.prn	cpm/asm	53683	4322	0	16268	83	0.1546113	0	0	10092	18.799247	10175	18.953859
15	Lrun20.prn	cpm/asm	42237	4138	0	13778	56	0.1325852	0	0	8623	20.415749	8679	20.548334
16	Init40.prn	cpm/asm	9810	1111	0	3226	167	1.7023445	0	0	2019	20.58104	2186	22.283384
17	COMPIL.LP0	assist	37420	911	0	16920	39	0.1042223	0	0	10530	28.140032	10569	28.244254
18	compio.lp0	assist	8750	225	0	5185	15	0.1714286	0	0	3304	37.76	3319	37.931429
19	comp2.lp0	assist	25035	533	0	14108	10	0.0399441	0	0	9676	38.64989	9686	38.689834
20	New1.lp0	assist	112819	4583	0	60770	573	0.5078932	0	0	43144	38.241786	43717	38.749679
21	P1.lp0	assist	47335	1562	0	26313	85	0.1795711	0	0	18267	38.590895	18352	38.770466
22	P2.lp0	assist	42060	1405	0	23402	63	0.149786	0	0	16599	39.46505	16662	39.614836
23	P3.lp0	assist	29005	951	0	16556	84	0.2896052	0	0	12110	41.751422	12194	42.041027
24	New4.lp0	assist	63640	2448	0	37101	192	0.301697	0	0	27302	42.900691	27494	43.202388
25	New2.lp0	assist	73104	2572	0	44851	188	0.2571679	0	0	34089	46.630827	34277	46.887995
26	Samp2.lp0	assist	1305	48	0	900	6	0.4597701	0	0	607	46.51341	613	46.97318
27	Samp.lp0	assist	1365	49	0	948	3	0.2197802	0	0	640	46.886447	643	47.106227
28	>> AVERAGE		35871.96	2517.4	0	15332.6	179.28	0.798081	0	0	9614.32	24.779213	9793.6	25.577294
29	>> STD DEV							1.0835834		0		15.45882		14.85

Figure B-3. Listing file data.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	File	File	Total	Total	Total	Total	Control	Control	High bit	High bit	Repeated	Repeated	Total	Total
2	Name	Type	Chars.	Control	> 80H	Repeated	Savings	% Savings	Savings	% Savings	Savings	% Savings	Savings	% Savings
3	Disintel.com	asm?	8448	1683	1834	787	249	2.9474432	230	2.7225379	88	1.0416667	567	6.7116477
4	Kermit.com	asm	9261	2437	2193	432	459	4.9562682	75	0.8098477	95	1.0258071	629	6.7919231
5	Sm5.com	asm	11008	2108	2439	1271	325	2.9523983	244	2.2165698	377	3.424782	946	8.59375
6	Cc1.com	asm	13696	3343	4801	553	337	2.4605724	752	5.4906542	113	0.8250584	1202	8.776285
7	Sql6.com	C	15872	4164	4343	827	687	4.328377	773	4.8702117	146	0.9198589	1606	10.118448
8	Bsort.com	asm	3840	1262	931	199	201	5.234375	85	2.2135417	122	3.1770833	408	10.625
9	Count.com	C	9344	2876	2344	459	674	7.2131849	323	3.4567637	39	0.4173801	1036	11.087329
10	Xmodem73.com	asm	4096	1107	1031	367	218	5.3222656	94	2.2949219	153	3.7353516	465	11.352539
11	Lu.com	C	17664	4900	5705	1307	1019	5.7687953	937	5.3045743	177	1.002038	2133	12.075408
12	L2.com	C	17152	4966	5451	1192	1187	6.9204757	915	5.3346549	81	0.4722481	2183	12.727379
13	Mbasic.com	asm?	24320	5992	9556	1827	1120	4.6052632	1335	5.4893092	733	3.0139803	3188	13.108553
14	M80.com	asm?	20096	5876	6764	3028	1210	6.0210987	737	3.6673965	1010	5.0258758	2957	14.714371
15	Paste.com	asm?	20480	5159	7209	1836	1345	6.5673828	856	4.1796875	1138	5.5566406	3339	16.303711
16	L80.com	asm?	10752	3076	3865	960	684	6.3616071	550	5.1153274	522	4.8549107	1756	16.331845
17	>> AVERAGE	ROW <<	13287.786	3496.3571	4176.1429	1074.6429	693.92857	5.1185343	564.71429	3.7975713	342.42857	2.463763	1601.0714	11.379871
18	>> STD.	DEV. <<						1.5179947		1.5411583		1.8532608		3.0984249

Figure B-4. Object file data.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	File	File	Total	Total	Total	Total	Control	Control	High bit	High bit	Repeated	Repeated	Total	Total
2	Name	Type	Chars.	Control	> 80H	Repeated	Savings	% Savings	Savings	% Savings	Savings	% Savings	Savings	% Savings
3	P3.lq0	asst/lpt	11392	1333	6030	74	49	0.4301264	1589	13.948385	0	0	1638	14.378511
4	Protect.aqm	asm/src	16128	1847	8595	202	55	0.3410218	2359	14.626736	0	0	2414	14.967758
5	Sd50.aqm	asm/src	44800	4781	23812	272	73	0.1629464	6718	14.995536	0	0	6791	15.158482
6	Lrun20.aqm	asm/src	14848	1635	7821	87	57	0.3838901	2219	14.944774	0	0	2276	15.328664
7	Rcpm-038.lqt	text	13184	1409	6948	120	68	0.5157767	1978	15.003034	0	0	2046	15.518811
8	P1.lq0	asst/lpt	19328	2106	10326	90	68	0.3518212	2949	15.257657	0	0	3017	15.609478
9	Bishowl4.aqm	asm/src	7296	823	3909	60	40	0.5482456	1116	15.296053	0	0	1156	15.844298
10	New2.lq0	asst/lpt	27776	2789	14888	241	81	0.2916187	4334	15.603399	0	0	4415	15.895017
11	Xref30.aqm	asm/src	22272	2178	11835	310	93	0.4175647	3461	15.539691	0	0	3554	15.957256
12	P2.lq0	asst/lpt	17024	1782	8996	76	72	0.4229323	2646	15.542763	0	0	2718	15.965695
13	New4.lq0	asst/lpt	25856	2761	13802	184	81	0.3132735	4059	15.698484	1	0.0038676	4141	16.015625
14	Redstick.dqc	text	6400	698	3373	28	49	0.765625	984	15.375	0	0	1033	16.140625
15	New1.lq0	asst/lpt	48128	5354	25643	295	140	0.290891	7638	15.87018	0	0	7778	16.16107
16	L2.cq	C/src	14080	1543	7484	77	74	0.5255682	2225	15.802557	0	0	2299	16.328125
17	Disk76.aqm	asm/src	33280	3373	17809	734	82	0.2463942	5523	16.595553	0	0	5605	16.841947
18	Squeezer.dqc	text	13952	1349	7639	56	69	0.4945528	2392	17.144495	0	0	2461	17.639048
19	Zcpr1.mqg	text	3968	448	2150	33	46	1.1592742	667	16.809476	0	0	713	17.96875
20	>> AVERAGE	ROW <<	19983.059	2129.9412	10650.588	172.88235	70.411765	0.4506778	3109.2353	15.532575	0.0588235	0.0002275	3179.7059	15.98348
21	>> STD.	DEV. <<						0.022986		0.7897444		0.000938		0.8907943

Figure B-5. Huffman file data.

APPENDIX C

SAMPLE STRING COMPRESSION ROUTINES

```
        /* sample repeat processor output routine */

extern int hi_str,      /* high bit shift flag */
           ctl_str;     /* control shift flag */
extern char char(),
            *outbuf,  /* output buffer pointer */
            *inbuf;   /* input buffer pointer */

r_out()
{
  int count;

  while ( (last = *inbuf) == ++*inbuf) {
    count++;
    if (count++ = 76) break;  /* string too long? */
  }
  *outbuf++ = '~';
  *outbuf++ = char(count);
  if (hi_str || ctl_str) {
    if (hi_str) last -= 0x80;
    if (ctl_str) last ^= 0x40;
    *outbuf++ = last;
    return (SUCCEED);
  }
  else {
    if (last > 0x7f) {
      *outbuf++ = '&';
      last -= 0x80;
    }
    if (last < 0x20) {
      *outbuf++ = '#';
      last ^= 0x40;
    }
    *outbuf = last;
    return (SUCCEED);
  }
}
```

Figure C-1. Sample repeat processor output routine.

```
        /* sample repeat processor input routine */

extern int hi_str,      /* high bit shift flag */
           ctl_str;     /* control shift flag */
extern char unchar(),
            *outbuf,  /* output buffer pointer */
            *inbuf;   /* input buffer pointer */

r_in()
{
  int count,                    /* repeat count */
      loc_hi;                   /* local high bit flag */

  inbuf++;                      /* get past the ~ */
  count = *inbuf++;
  if (*inbuf == '&') {          /* single hi-bit quote? */
    loc_hi = TRUE;
    inbuf++;
  }
  if (*inbuf == '#') {      /* single control quote? */
    inbuf++;
    *inbuf ^= 40H;
  }
  if (loc_hi) *inbuf += 0x80;           /* 1 high quote? */
  if (ctl_str) *inbuf = unchar(*inbuf); /* ctl string? */
  if (hi_str) *inbuf +=0x80;            /* hi string? */
  for ( i = 0 ; i < count ; i++) {      /* output */
    *outbuf++ = *inbuf;
  }
}
```

Figure C-2. Sample repeat processor input routine.

APPENDIX D

FILE ANALYSIS PROGRAM

COUNT.C

```
/*
 *              count.c         j.bray
 *              copyright (c) 1983 1984 john miller bray
 *
 *      counts the incidence of each character in a file, incidence
 *      of repeated strings, control characters, and strings of
 *      characters with the high bit set.  calculates potential savings
 *      in Kermit output string for case shift operators.  stopping on
 *      cpm eof mark (ascii), printer output, and collapsing null pro-
 *      tions of the arrays are user options.
 *
 */

/*
 *      revision history
 *
 *      ver 3.1 - 15 sept 1983 - fix fommand line option parser; add
 *                               over 64 k messages.
 *          3.0 - 30 august 83 - add command line options for sub files.
 *          2.9 - 11 july 83 - add warnings for over 64 k files.
 *          2.8 -    "         - add as 2.7 to rep. section for
 *                               repeat sequ's. of 3 chars.
 *          2.7 - 6 jul 83 -    add totals to ctl and hi bit distribs. for
 *                               calc'g savings.
 *          2.1 - 2.6 jul 83 - fixes for 2.0
 *          2.0 - jun 83 -      add distribs for ctl and hi bit chars
 *          1.0 - aug 82 - original version.
```

Figure D-1. File analysis program.

```
 *
 */

#include "b:count.h"
#define main_vers 3
#define rev_no 1



main(argc,argv)
int argc;
char *argv[];

{

int     array_index, ctl_in_row, hi_bit_in_row, repeat_in_row;
char    ascii,collapse,printer,first_flag,m_vers,r_no,last_char;

char    cmd_line_options;


        m_vers = main_vers;
        r_no = rev_no;

        printf("\ncount    version %d.%d  15 september 83\n\n",
                                                m_vers, r_no);
```

Figure D-1. File analysis program. (continued)

```
/*
        check invocation line and see if there is such a file.
        complain and quit if there are probs
*/

        if (argc < 2) {
                printf("no file specified for input \n");
                invoke();
                exit();
        }
        if (argc > 5) {
                printf(" improper invocation \n");
                invoke();
                exit();
        }
        if (fopen(argv[1],ibuf) == ERROR)
                exit(printf(">>> SOURCE FILE NOT ON DISK <<<<"));

/*
        set up our options flags per user request -- either
        from interactive or command line
*/

        if ((argc < 3) ) {              /* no switches, prompt */
                printf("ascii file (quit on ctrl-Z)? (y or <cr>/n)");
```

Figure D-1. File analysis program. (continued)

```
                ascii = toupper(getchar());
                if ((ascii == 'Y') || (ascii == '')) ascii = TRUE;
                else ascii = FALSE;

                printf("collapse the distributions? (y or <cr>/n) ");
                collapse = toupper(getchar());
                if ((collapse == 'Y') || (collapse == '')) collapse=TRUE;
                else collapse = FALSE;


                printf("printer output also? (y or <cr>/n) ");
                printer = toupper(getchar());
                if ((printer == 'Y') || (printer == '')) printer=TRUE;
                else printer = FALSE;
        }
        else {
                ascii = FALSE;
                printer = FALSE;
                collapse = FALSE;
                for ( i = 2 ; i < argc ; ++i) {
                        if (strcmp(argv[i], "-Z") == 0) ascii = TRUE;
                        if (strcmp(argv[i], "-C") == 0) collapse = TRUE;
                        if (strcmp(argv[i], "-P") == 0) printer = TRUE;
                        if (strcmp(argv[i], "-O") == 0) {
                                ascii = FALSE;
                                collapse = TRUE;
                                printer = TRUE;
```

Figure D-1. File analysis program. (continued)

```
                        break;
                }
                if (strcmp(argv[i],"-A") == 0) {
                        ascii = TRUE;
                        collapse = TRUE;
                        printer = TRUE;
                        break;
                }
        }
}

/*
        sign on
*/

        printf("character count for file %s", argv[1]);
        if (printer) {
                lprintf("\n\n\n");
                lprintf("\t\t\tcount  version %d.%d  15 sept 83",
                                                m_vers,r_no);
                lprintf("character count for file %s", argv[1]);
                lprintf("control-Z ");
                if (ascii) lprintf("used as eof.)");
                else lprintf("ignored. physical eof used.)");
        }

/*
```

Figure D-1. File analysis program. (continued)

```
        zero the counters in all the arrays
*/

        array_index = 0;
        while (array_index <= 0xff) {
                ctl_char_sets[array_index] = 0;
                hi_bit_sets[array_index] = 0;
                distrib[array_index] = 0;
                repeated_sets[array_index] = 0;
                ++array_index;
        }

/*
        init some variables
*/

        total = 0;                      /* total chars in file */
        ctl_in_row = 0;                 /* # ctl chars in a row */
        hi_bit_in_row = 0;                      /* #hi bits in row */
        control_count = 0;              /* # ctl chars */
        high_bit_count = 0;             /* # hi bit chars */
        ff = 0x0c;                      /* form feed */
        line_count = 0;                 /* #lines printed so far */
        first_flag = TRUE;              /* first time through flag */
        repeat_count = 0;
```

Figure D-1. File analysis program. (continued)

```
        repeat_in_row = 1;      /* # of repeated characters */
/*
        now do the file and count each character
*/

        while ((c = getc(ibuf)) != ERROR) {
                if ((ascii) && (c == 0x1a)) break;
                ++distrib[c];
                ++total;

                if (distrib[c] == 0xffff) {
                        printf(" warning: distrib[%3x] = 0xffff",c);
                        if (printer)
                          lprintf(" warning: distrib[%3x] = 0xffff",c);
                }
                if (total == 0xffff) {
                        printf("  warning: total = 0xffff");
                        printf("savings may be invalid");
                        if (printer) {
                          printf("  warning: total = 0xffff");
                          printf("savings may be invalid");
                        }
                }
                /* control case character? */
                if (c < 0x20) {
                        ++ctl_in_row;
```

Figure D-1. File analysis program. (continued)

```
            ++control_count;
    }
    /* else if not ctl and is end of ctl string */
    else if ( (c >= 0x20) && (ctl_in_row != 0) ) {
            if (ctl_in_row < 0xff)
                    ++ctl_char_sets[ctl_in_row];
            else ++ctl_char_sets[0xff];
            ctl_in_row = 0;
    }
    /* high bit set or end of hi bit string? */
    if (c >= 0x80) {
            ++hi_bit_in_row;
            ++high_bit_count;
    }
    else if (hi_bit_in_row != 0) {
            if(hi_bit_in_row < 0xff)
                    ++high_bit_sets[hi_bit_in_row];
            else ++hi_bit_sets[0xff];
            hi_bit_in_row = 0;
    }
    /* check for rpt characters.  note that this count is
            always one behind the real count and must be
            inc'd again at the end.
    */
    if (! first_flag) {
            if (c == last_char) {
                    ++repeat_count;
```

Figure D-1. File analysis program. (continued)

```
                        ++repeat_in_row;
                }
                else if (repeat_in_row != 1) {
                        ++repeat_count;
                        if (repeat_in_row < 0xff)
                           ++repeated_sets[repeat_in_row];
                        else ++repeated_sets[0xff];
                        repeat_on_row = 1;
                }
        }
        last_char = c;
        if (first_flag) first_flag = FALSE;
}

/* any left over strings? */

if (ctl_in_row != 0) {
        if (ctl_in_row < 0xff)
                ++ctl_char_sets[ctl_in_row];
        else ++ctl_char_sets[0xff];
        ctl_in_row = 0;
}
if (hi_bit_in_row != 0) {
        if (hi_bit_in_row < 0xff)
                ++hi_bit_sets[hi_bit_in_row];
        else ++hi_bit_sets[0xff];
        hi_bit_in_row = 0;
```

Figure D-1. File analysis program. (continued)

```
        }
        if (repeat_in_row != 1) {
                ++repetat_count;
                if (repeat_in_row < 0xff)
                        ++repeated_sets[repeat_in_row];
                else ++repeated_sets[0xff];
                repeat_in_row = 1;
        }

/*
        now print out data
*/

        printf("\n\n\t Frequencies of characters by hex value\n\n");
        if (printer)
          lprintf("\n\n\t Frequencies of characters by hex value\n\n");
        print_distrib(distrib,printer,collapse);
        printf("\n\nTOTAL CHARACTERS: \t%u\n\n",total1);
        if (printer) {
          lprintf("\n\nTOTAL CHARACTERS: \t%u\n\n",total1);
          if (line_count > 10) {
                lprintf("%c\n\n\n",ff);
                line_count = 0;
          }
          else lprintf("\n\n\n");
        }
        printf(\t Frequencies of control characters by string length\n\n");
```

Figure D-1. File analysis program. (continued)

```
printf("\t  (0xff value is for strings longer than 0xff alse)\n");
if (printer) {
  lprintf(\t Frequency of control characters by string length\n\n");
  lprintf("\t  (0xff value is for strings longer than 0xff also)\n");
}
print_distrib(ctl_char_sets,printer,collapse);
printf("\t Total number of control characters %u",control_count);
ctl_sav - saving(ctl_char_sets,2);
printf("\tamount of potential savings: %6u, percent: ",ctl_sav);
if (printer) {
  lprintf("\t Total number of control characters %u",control_count);
  lprintf("\tamount of potential savings: %6u, percent: ",ctl_sav);
  if (line_count > 10) {
        lprintf("%c\n\n\n",ff);
        line_count = 0;
  }
  else lprintf("\n\n\n");
}
printf("\tFrequencies of characters >= 0x80 by string length\n\n");
printf("\t  (0xff value is for strings longer than 0xff also)\n");
if (printer) {
  printf("\tFrequency of characters >= 0x80 by string length\n\n");
  printf("\t (0xff value is for strings longer than 0xff also)\n");
}
print_distrib(hi_bit_sets,printer,collapse);
hi_sav = saving(hi_bit_sets,2);
ttl_sav = ctl_sav + hi_sav;
```

Figure D-1. File analysis program. (continued)

```
printf("\nTotal number of high_bit characters: %u",
        high_bit_count);
printf("\tamount of potential saving: %6u, percent",
        high_sav);
printf("\ttotal potential saving: %6u, total percent",
        ttl_sav);
  if (printer) {
  printf("\nTotal number of high_bit characters: %u",
        high_bit_count);
  printf("\tamount of potential saving: %6u, percent",
        high_sav);
  printf("\ttotal potential saving: %6u, total percent",
        ttl_sav);
  if (line_count > 10) {
        lprintf("%c\n\n\n",ff);
        line_count = 0;
  }
  else lprintf("\n\n\n");
}
printf("\t Frequencies of repeated characters by string length\n\n");
printf("\t  (0xff value is for strings longer than 0xff)\n");
if(printer) {
  lprintf("\tRepeated character frequencies by string length\n\n");
  lprintf("\t (0xff values is for strings longer than 0xff)\n");
}
print_distrib(repeated_sets,printer,collapse);
rpt_sav = saving(repeated_sets,3);
```

Figure D-1. File analysis program. (continued)

```
        ttl_sav = ctl_sav + hi_sav + rpt_sav;
        printf("\nTotal number of repeated characters: %u",repeat_count);
        printf("\tamount of potential saving: %6u, percent:",
                rpt_sav);
        printf("total potential saving: %6u, total percent:",
                ttl_sav);
        if(printer) {
                lprintf("Total number of repeated characters: %u",
                                        repeat_count);
                lprintf("amount of potential saving: %6u, percent:",
                                rpt_sav);
                lprintf("total potential saving: %6u, total percent:",
                                ttl_sav);
        }


        /*
         * close our input file for neatness's sake
         * and go home to mama
         */

        fclose(ibuf);
        exit();
} /* end main */
```

Figure D-1. File analysis program. (continued)

```
invoke()        /* tells how to properly invoke the program */

{
        printf("\nProper usage is:");
        printf("\n[d:]count[xx] fn.ft [-z | -c | -p | -a]");
        printf("\nwhere: \t items in [] are optional.");
        printf("\n\t-z is stop on cpm ^Z eof");
        printf("\n\t-c is collapse distributions");
        printf("\n\t-p output to printer also");
        printf("\n\t-a do all above");
        printf("\n\t-o is -c -p but not -z");
        printf("\n\t-z -c -p may be used in any combination");
        printf("\n\t-a -o are used in lieu of the others");
        printf("\n\tand take precedence over -z -c -p");
        printf("\n");
}
```


Figure D-1. File analysis program. (continued)

VITA

John Miller Bray was born in Johnson City, Tennessee on January 15, 1948. He attended elementary and high schools there, receiving a high school diploma in June 1966 from Science Hill High School. The years from 1966 to 1971 were spent at The University of Tennessee, Knoxville, culminating in a Bachelor of Science degree in Physics, specialization in radiation safety.

After spending the years from 1971 to 1973 pursuing graduate studies in chemical physics, he worked for a year as a psychiatric technician. Returning to The University of Tennessee in 1974, he was granted a Master of Science degree in Educational Psychology in 1975. From then until 1982 he was a practicing college student personnel administrator. During that time he held positions as international admissions officer, director of residence halls, assistant dean of students, counselor, and director of international student affairs. He re-entered The University of Tennessee in 1982, and received the Master of Science degree in Computer Science in August 1984.