# KERMIT PROTOCOL MANUAL

*Sixth Edition*

Frank da Cruz

Columbia University Center for Computing Activities
New York, New York 10027

June 1986

## Preface to the Sixth Edition

The sixth edition (June 1986) of the *Kermit Protocol Manual* is being issued for two major reasons: to correct minor errors in the fifth edition, and to include new sections on two major protocol extensions: long packets and sliding windows. No attempt has been made to reorganize, rewrite, or otherwise improve the protocol manual. The Kermit protocol has been presented in an entirely different -- hopefully more thorough, organized, coherent, and useful (if not more formal) -- manner in the book, *Kermit, A File Transfer Protocol*, by Frank da Cruz, Digital Press, Bedford MA (1987), ISBN 0-932376-88-6, DEC order number EY-6705E-DP. If you have the book, you won't need this protocol manual. On the other hand, if you don't have the book, this manual should still contain all the necessary information. The *Kermit Protocol Manual* will continue to be freely distributed in perpetuity.

The bare-bones C-language Kermit program that appeared as an appendix in previous editions has been removed. It was not a particularly good example of how to write a Kermit program, and made the manual unnecessarily thick. For sample Kermit programs, see the source code for any of the hundreds of Kermit implementations, or follow the program fragments in the book.

## Preface to the Fifth Edition

The fifth edition (March 1984) attempts to clarify some fine points that had been left ambiguous in the 4th edition, particularly with respect to when and how prefix encoding is done, and when it is not, and about switching between block check types. A mechanism is suggested (in the Attributes section) for file archiving, and several attributes have been rearranged and some others added (this should do no harm, since no one to date has attempted to implement the attributes packet). A more complete protocol state table is provided, a few minor additions are made to the collection of packet types.

## Preface to the Fourth Edition

The fourth edition (November 1983) of the Kermit Protocol Manual incorporates some new ideas that grew from our experience in attempting to implement some of the features described in earlier editions, particularly user/server functions. These include a mechanism to allow batch transfers to be interrupted gracefully for either the current file or the entire batch of files; a "capability mask"; a protocol extension for passing file attributes. In addition, numbers are now written in decimal notation rather than octal, which was confusing to many readers. Also, several incompatible changes were made in minor areas where no attempts at an implementation had yet been made; these include:

- The format and interpretation of the operands to the server commands.
- Usurpation of the reserved fields 10-11 of the Send-Init packet, and addition of new reserved fields.

Most of the remaining material has been rewritten and reorganized, and much new material added, including a section on the recommended vocabulary for documentation and commands.

The previous edition of the Protocol Manual attempted to define "protocol version 3"; this edition abandons that concept. Since Kermit development is an unorganized, disorderly, distributed enterprise, no requirement can be imposed on Kermit implementors to include a certain set of capabilities in their implementations. Rather, in this edition we attempt to define the basic functionality of Kermit, and then describe various optional functions.

The key principle is that any implementation of Kermit should work with any other, no matter how advanced the one or how primitive the other. The capability mask and other Send-Init fields attempt to promote this principle.

## Acknowledgements

Bill Catchings and I designed the basic Kermit protocol at Columbia University in 1981. For ideas, we looked at some of the ANSI models (X3.57, X3.66), the ISO OSI model, some real-world "asynchronous protocols" (including the Stanford Dialnet and TTYFTP projects, the University of Utah Small FTP project), as well as at file transfer on full-blown networks like DECnet and ARPAnet.

Bill wrote the first two programs to implement the protocol, one for the DEC-20, one for a CP/M-80 microcomputer, and in the process worked out most of the details and heuristics required for basic file transfer. Meanwhile, Daphne Tzoar and Vace Kundakci, also of Columbia, worked out the additional details necessary for IBM mainframe communication, while writing IBM VM/CMS and PC-DOS versions.

Much credit should also go to Bernie Eiben of Digital Equipment Corporation for promoting widespread use of Kermit and for adding many insights into how it should operate, to Nick Bush and Bob McQueen of Stevens Institute of Technology, for many contributions to the "advanced" parts of the protocol, and for several major Kermit implementations, and to Leslie Spira and her group at The Source Telecomputing for adding full-duplex sliding window capability to the Kermit protocol.

Thanks to the many people all over the world who have contributed new Kermit implementations, who have helped with Kermit distribution through various user groups, and who have contributed to the quality of the protocol and its many implementations by reporting or fixing problems, criticizing the design, or suggesting new features. In particular, thanks to Ted Toal of Nevada City, CA, for a detailed list of corrections to the fifth edition of this manual.

And above all, thanks to Christine Gianone for taking charge of Kermit at Columbia; for keeping it alive, healthy, and strong; for promoting its development and use all over the world; for setting its tone and direction; for fostering its spirit. Without her guidance and perserverance, Kermit might have faded from the scene years ago.

The Kermit protocol was named after Kermit the Frog, star of the television series THE MUPPET SHOW. The name is used by permission of Henson Associates, Inc., New York City.

## Disclaimer

# 1. Introduction

This manual describes the Kermit protocol. It is assumed that you understand the purpose and operation of the Kermit file transfer facility, described in the *Kermit Users Guide*, and basic terminology of data communications and computer programming.

## 1.1. Background

The Kermit file transfer protocol is intended for use in an environment where there may be a diverse mixture of computers -- micros, personal computers, workstations, laboratory computers, timesharing systems -- from a variety of manufacturers. All these systems need have in common is the ability to communicate in ASCII over ordinary serial telecommunication lines.

Kermit was originally designed at Columbia University to meet the need for file transfer between our DECSYSTEM-20 and IBM 370-series mainframes and various microcomputers. It turned out that the diverse characteristics of these three kinds of systems resulted in a design that was general enough to fit almost any system. The IBM mainframe, in particular, strains most common assumptions about how computers communicate.

## 1.2. Overview

The Kermit protocol is specifically designed for character-oriented transmission over serial telecommunication lines. The design allows for the restrictions and peculiarities of the medium and the requirements of diverse operating environments -- buffering, duplex, parity, character set, file organization, etc. The protocol is carried out by Kermit programs on each end of the serial connection sending "packets" back and forth; the sender sends file names, file contents, and control information; the receiver acknowledges (positively or negatively) each packet.

The packets have a layered design, more or less in keeping with the ANSI and ISO philosophies, with the outermost fields used by the data link layer to verify data integrity, the next by the session layer to verify continuity, and the data itself at the application level.

Connections between systems are established by the ordinary user. In a typical case, the user runs Kermit on a microcomputer, enters terminal emulation, connects to a remote host computer (perhaps by dialing up), logs in, runs Kermit on the remote host, and then issues commands to that Kermit to start a file transfer, "escapes" back to the micro, and issues commands to that Kermit to start its side of the file transfer. Files may be transferred singly or in groups.

Basic Kermit provides only file transfer, and that is provided for *sequential files only*, though the protocol attempts to allow for various types of sequential files. Microcomputer implementations of Kermit are also expected to provide terminal emulation, to facilitate the initial connection.

More advanced implementations simplify the "user interface" somewhat by allowing the Kermit on the remote host to run as a "server", which can transfer files in either direction upon command from the local "user" Kermit. The server can also provide additional functionality, such as file management, messages, mail, and so forth. Other optional features also exist, including a variety of block check types, a mechanism for passing 8-bit data through a 7-bit communication link, a way to compressing a repeated sequence of characters, and so forth.

As local area networks become more popular, inexpensive, and standardized, the demand for Kermit and similar protocols may dwindle, but will never wither away entirely. Unlike hardwired networks, Kermit gives the ordinary user the power to establish reliable error-free connections between *any two* computers; this may always be necessary for one-shot or long-haul connections.

## 1.3. General Terminology

TTY: This is the term commonly used for a device which is connected to a computer over an EIA RS-232 serial telecommunication line. This device is most commonly an ASCII terminal, but it may be a microcomputer or even a large multi-user computer emulating an ASCII terminal. Most computers provide hardware (RS-232 connectors and UARTs) and software (device drivers) to support TTY connections; this is what makes TTY-oriented file transfer protocols like Kermit possible on almost any system at little or no cost.

LOCAL: When two machines are connected, the LOCAL machine is the one which you interact with directly, and which is in control of the terminal. The "local Kermit" is the one that runs on the local machine. A local Kermit always communicates over an external device (the micro’s communication port, an assigned TTY line, etc).

REMOTE: The REMOTE machine is the one on the far side of the connection, which you must interact with "through" the local machine. The "remote Kermit" runs on the remote machine. A remote Kermit usually communicates over its own "console", "controlling terminal", or "standard i/o" device.

HOST: Another word for "computer", usually meaning a computer that can provide a home for multiple users or applications. This term should be avoided in Kermit lore, unless preceded immediately by LOCAL or REMOTE, to denote which host is meant.

SERVER: An implementation of remote Kermit that can accept commands in packet form from a local Kermit program, instead of directly from the user.

USER: In addition to its usual use to denote the person using a system or program, "user" will also be used refer to the local Kermit program, when the remote Kermit is a server.

## 1.4. Numbers

All numbers in the following text are expressed in decimal (base 10) notation unless otherwise specified.

Numbers are also referred to in terms of their bit positions in a computer word. Since Kermit may be implemented on computers with various word sizes, we start numbering the bits from the "right" -- bit 0 is the least significant. Bits 0-5 are the 6 least significant bits; if they were all set to one, the value would be 63.

A special quirk in terminology, however, refers to the high order bit of a character as it is transmitted on the communication line, as the "8th bit". More properly, it is bit 7, since we start counting from 0. References to the "8th bit" generally are with regard to that bit which ASCII transmission sets aside for use as a parity bit. Kermit concerns itself with whether this bit can be usurped for the transmission of data, and if not, it may resort to "8th-bit prefixing".

## 1.5. Character Set

All characters are in ASCII (American national Standard Code for Information Interchange) representation, ANSI standard X3.4-1968. All implementations of Kermit transmit and receive characters only in ASCII. The ASCII character set is listed in Appendix III.

ASCII character mnemonics:

| | |
|---|---|
| NUL | Null, idle, ASCII character 0. |
| SOH | Start-of-header, ASCII character 1 (Control-A). |
| SP | Space, blank, ASCII 32. |
| CR | Carriage return, ASCII 13 (Control-M). |
| LF | Linefeed, ASCII 10 (Control-J). |

CRLF A carriage-return linefeed sequence.
DEL Delete, rubout, ASCII 127.

A control character is considered to be any byte whose low order 7 bits are in the range 0 through 31, or equal to 127. In this document, control characters are written in several ways:

Control-A
: This denotes ASCII character 1, commonly referred to as "Control-A". Control-B is ASCII character 2, and so forth.

CTRL-A
: This is a common abbreviation for "Control-A". A control character is generally typed at a computer terminal by holding down the key marked CTRL and pressing the corresponding alphabetic character, in this case "A".

`^A`
: "Uparrow" notation for CTRL-A. Many computer systems "echo" control characters in this fashion.

A printable ASCII character is considered to be any character in the range 32 (SP) through 126 (tilde).

## 1.6. Conversion Functions

Several conversion functions are useful in the description of the protocol and in the program example. The machine that Kermit runs on need operate only on integer data; these are functions that operate upon the numeric value of single ASCII characters.

`tochar(x) = x+32`
: Transforms the integer *x*, which is assumed to lie in the range 0 to 94, into a printable ASCII character; 0 becomes SP, 1 becomes "`!`", 3 becomes "`#`", etc.

`unchar(x) = x-32`
: Transforms the character *x*, which is assumed to be in the printable range (SP through tilde), into an integer in the range 0 to 94.

`ctl(x) = x XOR 64`
: Maps between control characters and their printable representations, preserving the high-order bit. If *x* is a control character, then

```
x = ctl(ctl(x))
```

that is, the same function is used to controllify and uncontrollify. The argument is assumed to be a true control character (0 to 31, or 127), or the result of applying CTL to a true control character (i.e. 63 to 95). The transformation is a mnemonic one -- `^A` becomes A and vice versa.

## 1.7. Protocol Jargon

A Packet is a clearly delimited string of characters, comprised of "control fields" nested around data; the control fields allow a Kermit program to determine whether the data has been transmitted correctly and completely. A packet is the unit of transmission in the Kermit protocol.

ACK stands for "Acknowledge". An ACK is a packet that is sent to acknowledge receipt of another packet. Not to be confused with the ASCII character ACK.

NAK stands for "Negative Acknowledge". A NAK is a packet sent to say that a corrupted or incomplete packet was received, the wrong packet was received, or an expected packet was not received. Not to be confused with the ASCII character NAK.

A timeout is an event that can occur if expected data does not arrive within a specified amount of time. The program generating the input request can set a "timer interrupt" to break it out of a nonresponsive read, so that recovery procedures may be activated.

# 2. Environment

## 2.1. System Requirements

The Kermit protocol requires that:

- The host can send and receive characters using 7- or 8-bit ASCII encoding over an EIA RS-232 physical connection, either hardwired or dialup.
- All printable ASCII characters are acceptable as input to the host and will not be transformed in any way[1]. Similarly, any intervening network or communications equipment ("smart modems", TELENET, terminal concentrators, port selectors, etc) must not transform or swallow any printable ASCII characters.
- A single ASCII *control character* can pass from one system to the other without transformation. This character is used for packet synchronization. The character is normally Control-A (SOH, ASCII 1), but can be redefined.
- If a host requires a line terminator for terminal input, that terminator must be a single ASCII control character, such as CR or LF, distinct from the packet synchronization character.
- When using a job's controlling terminal for file transfer, the system must allow the Kermit program to set the terminal to no echo, infinite width (no "wraparound" or CRLF insertion by the operating system), and no "formatting" of incoming or outgoing characters (for instance, raising lowercase letters to uppercase, transforming control characters to printable sequences, etc). In short, the terminal must be put in "binary" or "raw" mode, and, hopefully, restored afterwards to normal operation.
- The host's terminal input processor should be capable of receiving a single burst of 40 to 100 characters at normal transmission speeds. This is the typical size of packet.

Note that most of these requirements rule out the use of Kermit through IBM 3270 / ASCII protocol converters, except those (like the Series/1 or 7171 running the Yale ASCII package) that can be put in "transparant mode."

Kermit does *not* require:

- That the connection run at any particular baud rate.
- That the system can do XON/XOFF or any other kind of flow control. System- or hardware-level flow control can help, but it's not necessary. See section 3.7.
- That the system is capable of full duplex operation. Any mixture of half and full duplex systems is supported.
- That the system can transmit or receive 8-bit bytes. Kermit will take advantage of 8-bit connections to send binary files; if an 8-bit connection is not possible, then binary files may be sent using an optional prefix encoding.

[1] If they are translated to another character set, like EBCDIC, the Kermit program must be able to reconstruct the packet as it appeared on the communication line, before transformation.

## 2.2. Printable Text versus Binary Data

For transmission between unlike systems, files must be assigned to either of two catagories: *printable text* or *binary*.

A printable text file is one that can make sense on an unlike system -- a document, program source, textual data, etc. A binary file is one that will not (and probably can not) make sense on an unlike system -- an executable program, numbers stored in internal format, etc. On systems with 8-bit bytes, printable ASCII files will have the high order bit of each byte set to zero[2] (since ASCII is a 7-bit code) whereas binary files will use the high order bit of each byte for data, in which case its value can vary from byte to byte.

Many computers have no way to distinguish a printable file from a binary file -- especially one originating from an unlike system -- so the user may have to give an explicit command to Kermit to tell it whether to perform these conversions.

### 2.2.1. Printable Text Files

A primary goal of Kermit is for printable text files to be useful on the target system after transfer. This requires a standard representation for text during transmission. Kermit's standard is simple: 7-bit ASCII characters, with "logical records" (lines) delimited by CRLFs. It is the responsibility of systems that do not store printable files in this fashion to perform the necessary conversions upon input and output. For instance, IBM mainframes might strip trailing blanks on output and add them back on input; UNIX would prepend a CR to its normal record terminator, LF, upon output and discard it upon input. In addition, IBM mainframes must do EBCDIC/ASCII translation for text files.

No other conversions (e.g. tab expansion) are performed upon text files. This representation is chosen because it corresponds to the way text files are stored on most microcomputers and on many other systems. In many common cases, no transformations are necessary at all.

### 2.2.2. Binary Files

Binary files are transmitted as though they were a sequence of characters. The difference from printable files is that the status of the "8th bit" must be preserved. When binary files are transmitted to an unlike system, the main objective is that they can be brought back to the original system (or one like it) intact; no special conversions should be done during transmission, except to make the data fit the transmission medium.

For binary files, eight bit character transmission is permissible as long as the two Kermit programs involved can control the value of the parity bit, and no intervening communications equipment will change its value. In that case, the 8th bit of a transmitted character will match that of the original data byte, after any control-prefixing has been done. When one or both sides cannot control the parity bit, a special prefix character may be inserted, as described below.

Systems that do not store binary data in 8-bit bytes, or whose word size is not a multiple of 8, may make special provisions for "image mode" transfer of binary files. This may be done within the basic protocol by having the two sides implicitly agree upon a scheme for packing the data into 7- or 8-bit ASCII characters, or else the more flexible (but optional) file attributes feature may be used. The former method is used on PDP-10 36-bit word machines, in which text is stored five 7-bit bytes per word; the value of the "odd bit" is sent as the parity bit of every 5th word.

[2]There are some exceptions, such as systems that store text files in so-called "negative ASCII", or text files produced by word processors that use the high order bit to indicate underline or boldface attributes.

# 3. File Transfer

The file transfer protocol takes place over a *transaction*. A transaction is an exchange of packets beginning with a Send-Init (S) packet, and ending with a Break Transmission (B) or Error (E) packet[3], and may include the transfer of one or more files, all in the same direction. In order to minimize the unforseen, Kermit packets do not contain any control characters except one specially designated to mark the beginning of a packet. Except for the packet marker, only printable characters are transmitted. The following sequence characterizes basic Kermit operation; the *sender* is the machine that is sending files; the *receiver* is the machine receiving the files.

1. The sender transmits a Send-Initiate (S) packet to specify its parameters (packet length, timeout, etc; these are explained below).

2. The receiver sends an ACK (Y) packet, with its own parameters in the data field.

3. The sender transmits a File-Header (F) packet, which contains the file's name in the data field. The receiver ACKs the F packet, with no data in the data field of the ACK (optionally, it may contain the name under which the receiver will store the file).

4. The sender sends the contents of the file, in Data (D) packets. Any data not in the printable range is prefixed and replaced by a printable equivalent. Each D packet is acknowledged before the next one is sent.

5. When all the file data has been sent, the sender sends an End-Of-File (Z) packet. The receiver ACKs it.

6. If there is another file to send, the process is repeated beginning at step 3.

7. When no more files remain to be sent, the sender transmits an End-Of-Transmission (B) packet. The receiver ACKs it. This ends the transaction, and closes the logical connection (the physical connection remains open).

Each packet has a *sequence number*, starting with 0 for the Send Init. The acknowledgment (ACK or NAK) for a packet has the same packet number as the packet being acknowledged. Once an acknowledgment is successfully received the packet number is increased by one, modulo 64.

If the sender is remote, it waits for a certain amount of time (somewhere in the 5-30 second range) before transmitting the Send-Init, to give the user time to escape back to the local Kermit and tell it to receive files.

Each transaction starts fresh, as if no previous transaction had taken place. For example, the sequence number is set back to zero, and parameters are reset to their default or user-selected values.

## 3.1. Conditioning the Terminal

Kermit is most commonly run with the user sitting at a microcomputer, connected through a communications port to a remote timesharing system. The remote Kermit is using its job's own "controlling terminal" for file transfer. While the microcomputer's port is an ordinary device, a timesharing job's controlling terminal is a special one, and often performs many services that would interfere with normal operation of Kermit. Such services include echoing (on full duplex systems), wrapping lines by inserting carriage return linefeed sequences at the terminal width, pausing at the end of a screen or page full of text, displaying system messages, alphabetic case conversion, control character intepretation, and so forth. Mainframe Kermit programs should be prepared to disable as many of these services as possible before packet communication begins, and to restore them to their original condition at the end of a transaction. Disabling these services is usually known as "putting the terminal in binary mode."

[3]A transaction should also be considered terminated when one side or the other has stopped without sending an Error packet.

Kermit's use of printable control character equivalents, variable packet lengths, redefinable markers and prefixes, and allowance for any characters at all to appear between packets with no adverse effects provide a great deal of adaptability for those systems that do not allow certain (or any) of these features to be disabled.

## 3.2. Timeouts, NAKs, and Retries

If a Kermit program is capable of setting a timer interrupt, or setting a time limit on an input request, it should do so whenever attempting to read a packet from the communication line, whether sending or receiving files. Having read a packet, it should turn off the timer.

If the sender times out waiting for an acknowledgement, it should send the same packet again, repeating the process a certain number of times up to a retry limit, or until an acknowledgement is received. If the receiver times out waiting for a packet, it can send either a NAK packet for the expected packet or another ACK for the last packet it got. The latter is preferred.

If a packet from the sender is garbled or lost in transmission (the latter is detected by a timeout, the former by a bad checksum), the receiver sends a NAK for the garbled or missing packet. If an ACK or a NAK from the receiver is garbled or lost, the sender ignores it; in that case, one side or the other will time out and retransmit.

A retry count is maintained, and there is a retry threshold, normally set around 5. Whenever a packet is resent -- because of a timeout, or because it was NAK'd -- the counter is incremented. When it reaches the threshold, the transaction is terminated and the counter reset.

If neither side is capable of timing out, a facility for manual intervention must be available on the local Kermit. Typically, this will work by sampling the keyboard (console) periodically; if input, such as a CR, appears, then the same action is taken as if a timeout had occurred. The local Kermit keeps a running display of the packet number or byte count on the screen to allow the user to detect when traffic has stopped. At this point, manual intervention should break the deadlock.

Shared systems which can become sluggish when heavily used should adjust their own timeout intervals on a per-packet basis, based on the system load, so that file transfers won't fail simply because the system was too slow.

Normally, only one side should be doing timeouts, preferably the side with the greatest knowledge of the "environment" -- system load, baud rate, and so forth, so as to optimally adjust the timeout interval for each packet. If both sides are timing out, their intervals should differ sufficiently to minimize collisions.

## 3.3. Errors

During file transfer, the sender may encounter an i/o error on the disk, or the receiver may attempt to write to a full or write-protected device. Any condition that will prevent successful transmission of the file is called a "fatal error". Fatal errors should be detected, and the transfer shut down gracefully, with the pertinent information provided to the user. Error packets provide a mechanism to do this.

If a fatal error takes place on either the sending or receiving side, the side which encountered the error should send an Error (E) packet. The E packet contains a brief textual error message in the data field. Both the sender and receiver should be prepared to receive an Error packet at any time during the transaction. Both the sender and receiver of the Error packet should halt, or go back into into user command mode (a server should return to server command wait). The side that is local should print the error message on the screen.

There is no provision for sending nonfatal error messages, warnings, or information messages during a transaction. It would be possible to add such a feature, but this would require both sides agree to use it through setting of a bit in the capability mask, since older Kermits that did not know about such a feature would encounter an unexpected packet type and would enter the fatal error state. In any case, the utility of such a feature is questionable, since there is no guarantee that the user will be present to see such messages at the time they are sent; even if they are saved up for later perusal in a "message box", their significance may be long past by the time the user reads them. See the section on Robustness, below.

## 3.4. Heuristics

During any transaction, several heuristics are useful:

1. A NAK for the current packet is equivalent to an ACK for the previous packet (modulo 64). This handles the common situation in which a packet is successfully received, and then ACK'd, but the ACK is lost. The ACKing side then times out waiting for the next packet and NAKs it. The side that receives a NAK for packet *n+1* while waiting for an ACK for packet *n* simply sends packet *n+1*.

2. If packet *n* arrives more than once, simply ACK it and discard it. This can happen when the first ACK was lost. Resending the ACK is necessary *and* sufficient -- don't write the packet out to the file again!

3. When opening a connection, discard the contents of the line's input buffer before reading or sending the first packet. This is especially important if the other side is in receive mode (or acting as a server), in which case it may have been sending out periodic NAKs for your expected SEND-INIT or command packet. If you don't do this, you may find that there are sufficient NAKs to prevent the transfer -- you send a Send-Init, read the response, which is an old NAK, so you send another Send-Init, read the next old NAK, and so forth, up to the retransmission limit, and give up before getting to the ACKs that are waiting in line behind all the old NAKs. If the number of NAKs is below the cutoff, then each packet may be transmitted multiply.

4. Similarly, before sending a packet, you should clear the input buffer (after looking for any required handshake character). Failure to clear the buffer could result in propogation of the repetition of a packet caused by stacked-up NAKs.

5. If an ACK arrives for a packet that has already been ACK'd, simply ignore the redundant ACK and wait for the next ACK, which should be on its way.

## 3.5. File Names

The syntax for file names can vary widely from system to system. To avoid problems, it is suggested that filenames be represented in the File Header (F) packet in a "normal form", by default (that is, there should be an option to override such conversions).

1. Delete all pathnames and attributes from the file specification. The file header packet should not contain directory or device names; if it does, it may cause the recipient to try to store the file in an inaccessible or nonexistent area, or it may result in a very strange filename.

2. After stripping any pathname, convert the remainder of the file specification to the form "*name*`.`*type*", with no restriction on length (except that it fit in the data field of the F packet), and:

    a. Include no more than one dot.
    b. Not begin or end with a dot.
    c. The *name* and *type* fields contain digits and uppercase letters.

Special characters like `"$"`, `"_"`, `"-"`, `"&"`, and so forth should be disallowed, since they're sure to cause problems on one system or another.

The recipient, of course, cannot depend upon the sender to follow this convention, and should still take precautions. However, since most file systems embody the notion of a file name and a file type, this convention will allow these items to be expressed in a way that an unlike system can understand. The particular notation is chosen simply because it is the most common.

The recipient must worry about the length of the name and type fields of the file name. If either is too long, they must be truncated. If the result (whether truncated or not) is the same as the name of a file that already exists in the same area, the recipient should have the ability to take some special action to avoid writing over the original file.

Kermit implementations that convert file specifications to normal form by default should have an option to override this feature. This would be most useful when transferring files between like systems, perhaps used in conjunction with "image mode" file transfer. This could allow, for instance, one UNIX system to send an entire directory tree to another UNIX system.

## 3.6. Robustness

A major feature of the Kermit protocol is the ability to transfer multiple files. Whether a particular Kermit program can actually send multiple files depends on the capabilities of the program and the host operating system (any Kermit program can receive multiple files).

If a Kermit program can send multiple files, it should make every attempt to send the entire group specified. If it fails to send a particular file, it should not terminate the entire batch, but should go on the the next one, and proceed until an attempt has been made to send each file in the group.

Operating in this robust manner, however, gives rise to a problem: the user must be notified of a failure to send any particular file. Unfortunately, it is not sufficient to print a message to the screen since the user may not be physically present. A better solution would be to have the sender optionally keep a log of the transaction, giving the name of each file for which an attempt was made, and stating whether the attempt was successful, and if not, the reason. Additional aids to robustness are described in the Optional Features section, below.

## 3.7. Flow Control

On full duplex connections, XON/XOFF flow control can generally be used in conjunction with Kermit file transfer with no ill effects. This is because XOFFs are sent in the opposite direction of packet flow, so they will not interfere with the packets themselves. XON/XOFF, therefore, need not be implemented by the Kermit program, but can done by the host system. If the host system provides this capability, it should be used -- if both sides can respond XON/XOFF signals, then buffer overruns and the resulting costly packet retransmissions can be avoided.

Beware, however, of the following situation: remote Kermit is sending periodic NAKs, local system is buffering them on the operating system level (because the user has not started the local end of the file transfer yet); local line buffer becomes full, local systems sends XOFF, remote starts buffering them up on its end, user finally starts file transfer on local end, clears buffer, local operating system sends XON, and then all the remotely buffered NAKs show up, causing the packet echoing problem described above, despite the buffer clearing.

Flow control via modem signals can also be used when available.

Note that flow control should not be confused with "handshake" or "line turnaround" techniques that are used on simplex or half-duplex communication lines. In fact, the two techniques are mutually exclusive.

## 3.8. Basic Kermit Protocol State Table

The Kermit protocol can be described as a set of states and transitions, and rules for what to do when changing from one state to another. State changes occur based on the type of packets that are sent or received, or errors that may occur. Packets always go back and forth; the sender of a file always sends data packets of some kind (init, header, data) and the receiver always returns ACK or NAK packets.

Upon entering a given state, a certain kind of packet is either being sent or is expected to arrive -- this is shown on top of the description of that state. As a result of the action, various responses may occur; these are shown in the EVENT column. For each event, an appropriate ACTION is taken, and the protocol enters a NEW STATE.

The following table specifies basic Kermit operation. Timeouts and error conditions have been omitted from the following table for simplicity, but the action is as described above. Server operation and some of the advanced features are also omitted. A full-blown state table is given subsequently.

```
STATE   EVENT           ACTION                        NEW STATE

     -- SEND STATES --

Send Send-Init Packet:
S       Get NAK,bad ACK (None)                            S
        Get good ACK    Set remote's params, open file    SF
        (Other)         (None)                            A

Send File-Header Packet
SF      Get NAK,bad ACK (None)                            SF
        Get good ACK    Get bufferful of file data        SD
        (Other)         (None)                            A

Send File-Data Packet
SD      Get NAK,bad ACK (None)                            SD
        Get good ACK    Get bufferful of file data        SD
        (End of file)   (None)                            SZ
        (Other)         (None)                            A

Send EOF Packet
SZ      Get NAK,bad ACK (None)                            SZ
        Get good ACK    Get next file to send             SF
        (No more files) (None)                            SB
        (Other)         (None)                            A

Send Break (EOT) Packet
SB      Get NAK,bad ACK (None)                            SB
        Get good ACK    (None)                            C
        (Other)         (None)                            A

     -- RECEIVE STATES --

Wait for Send-Init Packet
R       Get Send-Init   ACK w/local params                RF
        (Other)         (None)                            A

Wait for File-Header Packet
RF      Get Send-Init   ACK w/local params
                         (previous ACK was lost)          RF
        Get Send-EOF    ACK (prev ACK lost)               RF
        Get Break       ACK                               C
        Get File-Header Open file, ACK                    RD
        (Other)         (None)                            A

Wait for File-Data Packet
RD      Get previous
         packet(D,F)    ACK it again                      RD
        Get EOF         ACK it, close the file            RF
        Get good data   Write to file, ACK                RD
        (Other)         (None)                            A

     -- STATES COMMON TO SENDING AND RECEIVING --

C       (Send Complete)                                   start
A       ("Abort")                                         start
```

# 4. Packet Format

## 4.1. Fields

The Kermit protocol is built around exchange of packets of the following format:

```
+------+------------+------------+------+-----------+-------+
| MARK | tochar(LEN) | tochar(SEQ) | TYPE |    DATA    | CHECK |
+------+------------+------------+------+-----------+-------+
```

where all fields consist of ASCII characters. The fields are:

MARK The synchronization character that marks the beginning of the packet. This should normally be CTRL-A, but may be redefined.

LEN The number of ASCII characters within the packet that follow this field, in other words the packet length minus two. Since this number is transformed to a single character via the `tochar()` function, packet character counts of 0 to 94 (decimal) are permitted, and 96 (decimal) is the maximum total packet length. The length does not include end-of-line or padding characters, which are outside the packet and are strictly for the benefit of the operating system or communications equipment, but it does include the block check characters.

SEQ The packet sequence number, modulo 64, ranging from 0 to 63. Sequence numbers "wrap around" to 0 after each group of 64 packets.

TYPE The packet type, a single ASCII character. The following packet types are required:

| | |
|---|---|
| D | Data packet |
| Y | Acknowledge (ACK) |
| N | Negative acknowledge (NAK) |
| S | Send initiate (exchange parameters) |
| B | Break transmission (EOT) |
| F | File header |
| Z | End of file (EOF) |
| E | Error |
| Q | *Reserved for internal use* |
| T | *Reserved for internal use* |

The NAK packet is used only to indicate that the expected packet was not received correctly, never to supply other kinds of information, such as refusal to perform a requested service. The NAK packet *always* has an empty data field. The T "packet" is used internally by many Kermit programs to indicate that a timeout occurred.

DATA The "contents" of the packet, if any contents are required in the given type of packet, interpreted according to the packet type. Control characters (bytes whose low order 7 bits are in the ASCII control range 0-31, or 127) are preceded by a special prefix character, normally "#", and "uncontrollified" via `ctl()`. A prefixed sequence may not be broken across packets. Logical records in printable files are delimited with CRLFs, suitably prefixed (e.g. "`#M#J`"). Logical records need not correspond to packets. Any prefix characters are included in the count. Optional encoding for 8-bit data and repeated characters is described later. The data fields of all packets are subject to prefix encoding, *except* the S, I, and A packets and their acknowledgements, which must *not* be encoded.

CHECK A block check on the characters in the packet between, but not including, the mark and the block check itself. The check for each packet is computed by both hosts, and must agree if a packet is to be accepted. A single-character arithmetic checksum is the normal and required block check. Only six bits of the arithmetic sum are included. In order that all the bits of each data character contribute to this quantity, bits 6 and 7 of the final value are added to the quantity formed by bits 0-5. Thus if *s* is the arithmetic sum of the ASCII characters, then

$$check = \mathrm{tochar}((s + ((s \text{ AND } 192)/64)) \text{ AND } 63)$$

This is the default block check, and all Kermits must be capable of performing it. Other optional block check types are described later.

The block check is based on the ASCII values of all the characters in the packet, including control fields and prefix characters. Non-ASCII systems must translate to ASCII before performing the block check calculation.

## 4.2. Terminator

Any line terminator that is required by the system may be appended to the packet; this is carriage return (ASCII 15) by default. Line terminators are not considered part of the packet, and are not included in the count or checksum. Terminators are not necessary to the protocol, and are invisible to it, as are any characters that may appear between packets. If a host cannot do single character input from a TTY line, then a terminator will be required when sending to that host. The terminator can be specified in the initial connection exchange.

Some Kermit implementations also use the terminator for another reason -- speed. Some systems are not fast enough to take in a packet and decode it character by character at high baud rates; by blindly reading and storing all characters between the MARK and the EOL, they are able to absorb the incoming characters at full speed and then process them at their own rate.

## 4.3. Other Interpacket Data

The space between packets may be used for any desired purpose. Handshaking characters may be necessary on certain connections, others may require screen control or other sequences to keep the packets flowing.

## 4.4. Encoding, Prefixing, Block Check

MARK, LEN, SEQ, TYPE, and CHECK are *control fields*. Control fields are always literal single-character fields, except that the CHECK field may be extended by one or two additional check characters. Each control field is encoded by `tochar()` or taken literally, but never prefixed. The control fields never contain 8-bit data.

The DATA field contains a string of data characters in which any control characters are encoded printably and preceded with the control prefix. The decision to prefix a character in this way depends upon whether its low order 7 bits are in the ASCII control range, i.e. 0-31 or 127. Prefix characters that appear in the data must themselves be prefixed by the control prefix, but unlike control characters, these retain their literal value in the packet. The character to be prefixed is considered a prefix character if its low-order 7 bits corresponds to an active prefix character, such as # (ASCII 35), *regardless of the setting of its high-order bit*.

During decoding, any character that follows the control prefix, but is not in the control range, is taken literally. Thus, it does no harm to prefix a printable character, even if that character does not happen to be an active prefix.

The treatment of the high order ("8th") bit of a data byte is as follows:

- If the communication channel allows 8 data bits per character, then the original value of the 8th bit is retained in the prefixed character. For instance, a data byte corresponding to a Control-A with the 8th bit set would be send as a control prefix, normally "#", without the 8th bit set, followed by `ctl(^A)` *with* the 8th bit set. In binary notation, this would be

  ```
  00100011 11000001
  ```

  In this case, the 8th bit is figured into all block check calculations.

- If the communication channel or one of the hosts requires parity on each character, and both sides are capable of 8th-bit prefixing, then the 8th bit will be used for parity, and must *not* be included in the block check. 8th bit prefixing is an option feature described in greater detail in Section 6, below.

- If parity is being used but 8th-bit prefixing is *not* being done, then the value of the 8th bit of each data byte will be lost and binary files will not be transmitted correctly. Again, the 8th bit does not figure into the block check.

The data fields of all packets are subject to prefix encoding, *except* S, I, and A packets, and the ACKs to those packets (see below).

# 5. Initial Connection

Initial connection occurs when the user has started up a Kermit program on both ends of the physical connection. One Kermit has been directed (in one way or another) to send a file, and the other to receive it.

The receiving Kermit waits for a "Send-Init" packet from the sending Kermit. It doesn't matter whether the sending Kermit is started before or after the receiving Kermit (if before, the Send-Init packet should be retransmitted periodically until the receiving Kermit acknowledges it). The data field of the Send-Init packet is optional; trailing fields can be omitted (or left blank, i.e. contain a space) to accept or specify default values.

The Send-Init packet contains a string of configuration information in its data field. The receiver sends an ACK for the Send-Init, whose data field contains its own configuration parameters. The data field of the Send-Init and the ACK to the Send-Init are *literal*, that is, there is no prefix encoding. This is because the two parties will not know *how* to do prefix encoding until *after* the configuration data is exchanged.

It is important to note that newly invented fields are added at the right, so that old Kermit programs that do not have code to handle the new fields will act as if they were not there. For this reason, the default value for any field, indicated by blank, should result in the behavior that occurred before the new field was defined or added.

```
  1      2      3      4      5      6      7      8      9      10...
+------+------+------+------+------+------+------+------+------+-------
| MAXL | TIME | NPAD | PADC | EOL  | QCTL | QBIN | CHKT | REPT | CAPAS
+------+------+------+------+------+------+------+------+------+-------
```

The fields are as follows (the first and second person "I" and "you" are used to distinguish the two sides). Fields are encoded printably using the `tochar()` function unless indicated otherwise.

1. MAXL The maximum length packet I want to receive, a number up to 94 (decimal). (This really means the biggest value I want to see in a LEN field.) You respond with the maximum you want me to send. This allows systems to adjust to each other's buffer sizes, or to the condition of the transmission medium.

2. TIME The number of seconds after which I want you to time me out while waiting for a packet from me. You respond with the amount of time I should wait for packets from you. This allows the two sides to accommodate to different line speeds or other factors that could cause timing problems. Only one side needs to time out. If both sides time out, then the timeout intervals should not be close together.

3. NPAD The number of padding characters I want to precede each incoming packet; you respond in kind. Padding may be necessary when sending to a half duplex system that requires some time to change the direction of transmission, although in practice this situation is more commonly handled by a "handshake" mechanism.

4. PADC The control character I need for padding, if any, transformed by `ctl()` (*not* `tochar()`) to make it printable. You respond in kind. Normally NUL (ASCII 0), some systems use DEL (ASCII 127). This field is to be ignored if the value NPAD is zero.

5. EOL The character I need to terminate an incoming packet, if any. You respond in kind. Most systems that require a line terminator for terminal input accept carriage return for this purpose (note, because there is no way to specify that no EOL should be sent, it would have been better to use `ctl()` for this field rather than `tochar()`, but it's too late now).

6. QCTL (verbatim) The printable ASCII character I will use to quote control characters, normally and by default "`#`". You respond with the one you will use.

*The following fields relate to the use of OPTIONAL features of the Kermit protocol, described in section 6.*

7. QBIN (verbatim) The printable ASCII character I want to use to quote characters which have the 8th bit set, for transmitting binary files when the parity bit cannot be used for data. Since this kind of quoting increases both processor and transmission overhead, it is normally to be avoided. If used, the quote character must be in the range ASCII 33-62 ("`!`" through "`>`") or 96-126 ("`‘`" through "`~`"), but different from the control-quoting character. This field is interpreted as follows:

   `Y` I agree to 8-bit quoting if you request it (I don't need it).

N I will not do 8-bit quoting (I don't know how).

& (or any other character in the range 33-62 or 96-126) I need to do 8-bit quoting using this character (it will be done if the other Kermit puts a Y in this field, or responds with the same prefix character, such as &). The recommended 8th-bit quoting prefix character is "&".

*Anything Else* : 8-bit quoting will not be done.

Note that this scheme allows either side to initiate the request, and the order does not matter. For instance, a micro capable of 8-bit communication will normally put a "Y" in this field whereas a mainframe that uses parity will always put an "&". No matter who sends first, this combination will result in election of 8th-bit quoting.

8. CHKT (Verbatim) Check Type, the method for detecting errors. "1" for single-character checksum (the normal and required method), "2" for two-character checksum (optional), "3" for three-character CRC-CCITT (optional). If your response agrees, the designated method will be used; otherwise the single-character checksum will be used.

9. REPT The prefix character I will use to indicate a repeated character. This can be any printable character in the range ASCII 33-62 or 96-126, but different from the control and 8th-bit prefixes. SP (32) denotes no repeat count processing is to be done. Tilde ("~") is the recommended and normal repeat prefix. If you don't respond identically, repeat counts will not be done. Groups of at least 3 or 4 identical characters may be transmitted more efficiently using a repeat count, though an individual implementation may wish to set a different threshhold.

10-?. CAPAS

A bit mask, in which each bit position corresponds to a capability of Kermit, and is set to 1 if that capability is present, or 0 if it is not. Each character contains a 6-bit field (transformed by `tochar()`), whose low order bit is set to 1 if another capability byte follows, and to 0 in the last capability byte. The capabilities defined so far are:

#1 *Reserved*
#2 *Reserved*
#3 Ability to accept "A" packets (file attributes)
#4 Ability to do full duplex sliding window protocol
#5 Ability to transmit and receive extended-length packets

The capability byte as defined so far would then look like:

```
 bit5 bit4 bit3 bit2 bit1 bit0
+----+----+----+----+----+----+
| #1 | #2 | #3 | #4 | #5 |  0 |
+----+----+----+----+----+----+
```

If all these capabilities were "on", the value of the byte would be 76 (octal). When capability 6 is added, the capability mask will look like this:

```
 bit5 bit4 bit3 bit2 bit1 bit0    bit5 bit4 bit3 bit2 bit1 bit0
+----+----+----+----+----+----+   +----+----+----+----+----+----+
| #1 | #2 | #3 | #4 | #5 |  1 |   | #6 | -- | -- | -- | -- |  0 |
+----+----+----+----+----+----+   +----+----+----+----+----+----+
```

CAPAS+1. WINDO
Window size (see section 7.2).

CAPAS+2. MAXLX1
Extended packet length (see section 7.1).

CAPAS+3. MAXLX2
Extended packet length (see section 7.1).

The receiving Kermit responds with an ACK ("Y") packet in the same format to indicate its own preferences, options, and parameters. The ACK need not contain the same number of fields as the the Send-Init. From that point, the two Kermit programs are "configured" to communicate with each other for the remainder of the transaction. In the case of 8th-bit quoting, one side must specify the character to be used, and the other must agree with a "Y" in the same field, but the order in which this occurs does not matter. Similarly for checksums -- if one side requests 2 character checksums and the other side responds with a "1" or with nothing at all, then single-character checksums will be done, since not all implementations can be expected to do 2-character checksums or CRCs. And for repeat counts; if the repeat field of the send-init and the ACK do not agree, repeat processing will not be done.

All Send-Init fields are optional. The data field may be left totally empty. Similarly, intervening fields may be defaulted by setting them to blank. Kermit implementations should know what to do in these cases, namely apply appropriate defaults. The defaults should be:

MAXL: 80
TIME: 5 seconds
NPAD: 0, no padding
PADC: 0 (NUL)
EOL: CR (carriage return)
QCTL: the character "#"
QBIN: space, can't do 8-bit quoting
CHKT: "1", single-character checksum
REPT: No repeat count processing
CAPAS: All zeros (no special capabilities)
WINDO: Blank (zero) - no sliding windows
MAXLX1:
Blank (zero) - no extended length packets
MAXLX2:
Blank (zero) - no extended length packets

There are no prolonged negotiations in the initial connection sequence -- there is one Send-Init and one ACK in reply. Everything must be settled in this exchange.

The very first Send-Init may not get through if the sending Kermit makes wrong assumptions about the receiving host. For instance, the receiving host may require certain parity, some padding, handshaking, or a special end of line character in order to read the Send-Init packet. For this reason, there should be a way for the user the user to specify whatever may be necessary to get the first packet through.

A parity field is not provided in the Send-Init packet because it could not be of use. If the sender requires a certain kind of parity, it will also be sending it. If the receiver does not know this in advance, i.e. *before* getting the Send-Init, it will not be able to read the Send-Init packet.

# 6. Optional Features

The foregoing sections have discussed basic, required operations for any Kermit implementation. The following sections discuss optional and advanced features.

## 6.1. 8th-Bit and Repeat Count Prefixing

Prefix quoting of control characters is mandatory. In addition, prefixing may also be used for 8-bit quantities or repeat counts, when both Kermit programs agree to do so. 8th-bit prefixing can allow 8-bit binary data pass through 7-bit physical links. Repeat count prefixing can improve the throughput of certain kinds of files dramatically; binary files (particularly executable programs) and structured text (highly indented or columnar text) tend to be the major beneficiaries.

When more than one type of prefixing is in effect, a single data character can be preceded by more than one prefix character. Repeat count processing can only be requested by the sender, and will only be used by the sender if the receiver agrees. 8th-bit prefixing is a special case because its use is normally not desirable, since it increases both processing and transmission overhead. However, since it is the only straightforward mechanism for binary file transfer available to those systems that usurp the parity bit, a receiver must be able to request the sender to do 8th-bit quoting, since most senders will not normally do it by default.

The repeat prefix is followed immediately by a single-character repeat count, encoded printably via `tochar()`, followed by the character itself (perhaps prefixed by control or 8th bit prefixes, as explained below). The repeat count may express values from 0 to 94. If a character appears more than 94 times in a row, it must be "cut off" at 94, emitted with all appropriate prefixes, and "restarted". The following table should clarify Kermit’s prefixing mechanism (the final line shows how a sequence of 120 consecutive NULs would be encoded):

| Character | Prefixed Representation | With Repeat Count for 8 |
|---|---|---|
| A | A | ~(A ["(" is ASCII 40 - 32 = 8] |
| ^A | #A | ~(#A |
| ’A | &A | ~(&A |
| ’^A | &#A | ~(&#A |
| # | ## | ~(## |
| ’# | &## | ~(&## |
| & | #& | ~(#& |
| ’& | &#& | ~(&#& |
| ~ | #~ | ~(#~ |
| ’~ | &#~ | ~(&#~ |
| NUL | #@ | ~~#@~:#@ [120 NULs] |

`A` represents any printable character, `^A` represents any control character, `’x` represents any character with the 8th bit set. The # character is used for control-character prefixing, and the & character for 8-bit prefixing. The repeat count must always precede any other prefix character. The repeat count is taken literally (after transformation by `unchar()`; for instance "#" and "&" immediately following a "~" denote repeat counts, not control characters or 8-bit characters. The control prefix character "#" is most closely bound to the data character, then the 8-bit prefix, then the repeat count; in other words, the order is: repeat prefix and count, 8-bit prefix, control prefix, and the data character itself. To illustrate, note that `&#A` is *not* equivalent to `#&A`.

When the parity bit is available for data, then 8th-bit prefixing should not be done, and the 8th bit of the prefixed character will have the same value as the 8th bit of the original data byte. In that case, the table looks like this:

```
               Prefixed          With
Character      Representation    Repeat Count for 8
   'A             'A              ~('A
   '^A            #'A             ~(#'A
   '#             #'#             ~(#'#
   '&             '&              ~('&
   '~             #'~             ~(#'~
```

Note that since 8th bit prefixing is not being done, "&" is not being used as an 8th bit prefix character, so it does not need to be prefixed with "#". Also, note that the 8th bit is set on the final argument of the repeat sequence, no matter how long, and not on any of the prefix characters.

Finally, remember the following rules:

- *Prefixed sequences must not be broken across packets.*
- *Control, 8th-bit, and repeat count prefixes must be distinct.*
- *Data fields of all packets must pass through the prefix encoding mechanism, except for S, I, and A packets, and ACKs to those packets, whose data fields must not be encoded.*

In the first rule above, note that a prefixed sequence means a single character and all its prefixes, like `~%&#X`, *not* a sequence like `#M#J`, which is *two* prefixed sequences.

## 6.2. Server Operation

A Kermit server is a Kermit program running remotely with no "user interface". All commands to the server arrive in packets from the local Kermit. SERVER operation is much more convenient than basic operation, since the user need never again interact directly with the remote Kermit program after once starting it up in server mode, and therefore need not issue complementary SEND and RECEIVE commands on the two sides to get a file transfer started; rather, a single command (such as SEND or GET) to the local Kermit suffices. Kermit servers can also provide services beyond file transfer.

Between transactions, a Kermit server waits for packets containing server commands. The packet sequence number is always set back to 0 after a transaction. A Kermit server in command wait should be looking for packet 0, and command packets sent to servers should also be packet 0. Certain server commands will result in the exchange of multiple packets. Those operations proceed exactly like file transfer.

A Kermit server program waiting for a command packet is said to be in "server command wait". Once put into server command wait, the server should never leave it until it gets a command packet telling it to do so. This means that after any transaction is terminated, either normally or by any kind of error, the server must go back into command wait. While in command wait, a server may elect to send out periodic NAKs for packet 0, the expected command packet. Since the user may be disconnected from the server for long periods of time (hours), the interval between these NAKs should be significantly longer than the normal timeout interval (say, 30-60 seconds, rather than 5-10). The periodic NAKs are useful for breaking the deadlock that would occur if a local program was unable to time out, and sent a command that was lost. On the other hand, they can cause problems for local Kermit programs that cannot clear their input buffers, or for systems that do XON/XOFF blindly, causing the NAKs to buffered in the server's host system output buffer, to be suddenly released en masse when an XON appears. For this reason, servers should have an option to set the command-wait wakeup interval, or to disable it altogher.

Server operation must be implemented in two places: in the server itself, and in any Kermit program that will be communicating with a server. The server must have code to read the server commands from packets and respond to them. The user Kermit must have code to parse the user's server-related commands, to form the server command packets, and to handle the responses to those server commands.

### 6.2.1. Server Commands

Server commands are listed below. Not all of them have been implemented, and some may never be, but their use should be reserved. Although server-mode operation is optional, certain commands should be implemented in every server. These include Send-Init (S), Receive-Init (R), and the Generic Logout (GL) and/or Finish (GF) commands. If the server receives a command it does not understand, or cannot execute, it should respond with an Error (E) packet containing a message like "Unimplemented Server Command" and both sides should set the packet sequence number back to 0, and the server should remain in server command wait. Only a GL or GF command should terminate server operation.

Server commands are as follows:

S Send Initiate (exchange parameters, server waits for a file).
R Receive Initiate (ask the server to send the specified files).
I Initialize (exchange parameters).
X Text header. Allows transfer of text to the user's screen in response to a generic or host command. This works just like file transfer except that the destination "device" is the screen rather than a file. Data field may contain a filename, title, or other heading.
C Host Command. The data field contains a string to be executed as a command by the host system command processor.
K Kermit Command. The data field contains a string in the interactive command language of the Kermit server (normally a SET command) to be executed as if it were typed in at command level.
G Generic Kermit Command. Single character in data field (possibly followed by operands, shown in {braces}, optional fields in [brackets]) specifies the command:

  I Login [{*user[*password[*account]]}]
  C CWD, Change Working Directory [{*directory[*password]}]
  L Logout, Bye
  F Finish (Shut down the server, but don't logout).
  D Directory [{*filespec}]
  U Disk Usage Query [{*area}]
  E Erase (delete) {*filespec}
  T Type {*filespec}
  R Rename {*oldname*newname}
  K Copy {*source*destination}
  W Who's logged in? (Finger) [{*user ID or network host[*options]}]
  M Send a short Message {*destination*text}
  H Help [{*topic}]
  Q Server Status Query
  P Program {*[program-filespec][*program-commands]}
  J Journal {*command[*argument]}
  V Variable {*command[*argument[*argument]]}

  Asterisk as used above ("`*`") represents a single-character length field, encoded using `tochar()`, for the operand that follows it; thus lengths from 0 to 94 may be specified. This allows multiple operands to be clearly delimited regardless of their contents.

Note that field length encoding is used within the data field of all Generic command packets, but not within the data fields of the other packets, such as S, I, R, X, K, and C.

All server commands that send arguments in their data fields should pass through the prefix encoding mechanism. Thus if a data character or length field happens to correspond to an active prefix character, it must itself be prefixed. The field length denotes the length of the field *before* prefix encoding and (hopefully) *after* prefix decoding. For example, to send a generic command with two fields, "ABC" and "ZZZZZZZZ", first each field would be prefixed by `tochar()` of its length, in this case `tochar(3)` and `tochar(8)`, giving "`#ABC(ZZZZZZZZ`". But "`#`" is the normal control prefix character so it must be prefixed itself, and the eight Z's can be condensed to 3 characters using a repeat prefix (if repeat counts are in effect), so the result after encoding would be "`##ABC(~(Z`" (assuming the repeat prefix is tilde ("`~`"). The recipient would decode this back into the original "`#ABC(ZZZZZZZZ`" before attempting to extract the two fields.

Since a generic command must fit into a single packet, the program sending the command should ensure that the command actually fits, and should not include length fields that point beyond the end of the packet. Servers, however, should be defensive and not attempt to process any characters beyond the end of the data field, even if the argument length field would lead them to do so.

### 6.2.2. Timing

Kermit does not provide a mechanism for suspending and continuing a transaction. This means that text sent to the user's screen should not be frozen for long periods (i.e. not longer than the timeout period times the retry threshold).

Between transactions, when the server has no tasks pending, it may send out periodic NAKs (always with type 1 checksums) to prevent a deadlock in case a command was sent to it but was lost. These NAKs can pile up in the local "user" Kermit's input buffer (if it has one), so the user Kermit should be prepared to clear its input buffer before sending a command to a server. Meanwhile, servers should recognize that some systems provide no function to do this (or even when they do, the process can be foiled by system flow control firmware) and should therefore provide a way turn off or slow down the command-wait NAKs.

### 6.2.3. The R Command

The R packet, generally sent by a local Kermit program whose user typed a GET command, tells the server to send the files specified by the name in the data field of the R packet. Since we can't assume that the two Kermits are running on like systems, the local (user) Kermit must parse the file specification as a character string, send it as-is (but encoded) to the server, and let the server take care of validating its syntax and looking up the file. If the server can open and read the specified file, it sends a Send-Init (S) packet -- *not an acknowledgement!* -- to the user, and then completes the file-sending transaction, as described above.

If the server cannot send the file, it should respond with an error (E) packet containing a reason, like "File not found" or "Read access required".

Thus, the only two valid responses to a successfully received R packet are an S packet or an E packet. The R packet is not ACK'd.

### 6.2.4. The K Command

The K packet can contain a character string which the server interprets as a command in its own interactive command language. This facility is useful for achieving the same effect as a direct command without having to shut down the server, connect back to the remote system, continue it (or start a new one), and issue the desired commands. The server responds with an ACK if the command was executed successfully, or an error packet otherwise. The most likely use for the K packet might be for transmitting SET commands, e.g. for switching between text and binary file modes.

### 6.2.5. Short and Long Replies

Any request made of a server may be answered in either of two ways, and any User Kermit that makes such a request should be prepared for either kind of reply:

- *A short reply*. This consists of a single ACK packet, which may contain text in its data field. For instance, the user might send a disk space query to the server, and the server might ACK the request with a short character string in the data field, such as "12K bytes free". The user Kermit should display this text on the screen.
- *A long reply*. This proceeds exactly like a file transfer (and in some cases it may be a file transfer). It begins with one of the following:
    - A File-Header (F) packet (optionally followed by one or more Attributes packets; these are

discussed later);

- A Text-Header (X) packet.
- A Send-Init (S) Packet, followed by an X or F packet.

After the X or F packet comes an arbitrary number of Data (D) packets, then an End-Of-File (Z) packet, and finally a Break-Transmission (B) packet, as for ordinary file transfer.

A long reply should begin with an S packet unless an I-packet exchange has already taken place, *and* the type 1 (single-character) block check is being used.

### 6.2.6. Additional Server Commands

The following server commands request the server to perform tasks other than sending or receiving files. Almost any of these can have either short or long replies. For instance, the Generic Erase (GE) command may elicit a simple ACK, or a stream of packets containing the names of all the files it erased (or didn't erase). These commands are now described in more detail; arguments are as provided in commands typed to the user Kermit (subject to prefix encoding); no transformations to any kind of normal or canonic form are done -- filenames and other operands are in the syntax of the server's host system.

I Login. For use when a Kermit server is kept perpetually running on a dedicated line. This lets a new user obtain an identity on the server's host system. If the data field is empty, this removes the user's identity, so that the next user does not get access to it.

L Logout, Bye. This shuts down the server entirely, causing the server itself to log out its own job. This is for use when the server has been started up manually by the user, who then wishes to shut it down remotely. For a perpetual, dedicated server, this command simply removes the server's access rights to the current user's files, and leaves the server waiting for a new login command.

F Finish. This is to allow the user to shut down the server, putting its terminal back into normal (as opposed to binary or raw) mode, and putting the server's job back at system command level, still logged in, so that the user can connect back to the job. For a perpetual, dedicated server, this command behaves as the L (BYE) command.

C CWD. Change Working Directory. This sets the default directory or area for file transfer on the server's host. With no operands, this command sets the default area to be the user's own default area.

D Directory. Send a directory listing to the user. The user program can display it on the terminal or store it in a file, as it chooses. The directory listing should contain file sizes and creation dates as well as file names, if possible. A wildcard or other file-group designator may be specified to ask the server list only those files that match. If no operand is given, all files in the current area should be shown.

U Disk Usage Query. The server responds with the amount of space used and the amount left free to use, in K bytes (or other units, which should be specified).

E Erase (delete). Delete the specified file or file group.

T Type. Send the specified file or file group, indicating (by starting with an X packet rather than an F packet, or else by using the Type attribute) that the file is to be displayed on the screen, rather than stored.

R Rename. Change the name of the file or files as indicated. The string indicating the new name may contain other attributes, such as protection code, permitted in file specifications by the host.

K Copy. Produce a new copy of the file or file group, as indicated, leaving the source file(s) unmodified.

W Who's logged in? (Finger). With no arguments, list all the users who are logged in on the server's host system. If an argument is specified, provide more detailed information on the specified user or network host.

M Short Message. Send the given short (single-packet) message to the indicated user's screen.

P Program. This command has two arguments, program name (filespec), and command(s) for the program. The first field is required, but may be left null (i.e. zero length). If it is null, the currently loaded program is "fed" the specified command. If not null, the specified program is loaded and started; if a program command is given it is fed to the program as an initial command (for instance, as a command line argument on systems that support that concept). In any case, the output of the program is sent back in packets as either a long or short reply, as described above.

J Journal. This command controls server transaction logging. The data field contains one of the following:

- `+` Begin/resume logging transactions. If a filename is given, close any currently open transaction and then open the specified file as the new transaction log. If no name given, but a log file was already open, resume logging to that file. If no filename was given and no log was open, the server should open a log with a default name, like `TRANSACTION.LOG`.
- `-` Stop logging transactions, but don't close the current transaction log file.
- `C` Stop logging and close the current log.
- `S` Send the transaction log as a file. If it was open, close it first.

Transaction logging is the recording of the progress of file transfers. It should contain entries showing the name of each file transferred, when the transfer began and ended, whether it completed successfully, and if not, why.

V Set or Query a variable. The *command* can be S or Q. The first argument is the variable name. The second argument, if any, is the value.

- S Set the specified variable to the specified value. If the value is null, then undefine the variable. If the variable is null then do nothing. If the variable did not exist before, create it. The server should respond with an ACK if successful, and Error packet otherwise.
- Q Query the value of the named variable. If no variable is supplied, display the value of all active variables. The server responds with either a short or long reply, as described above. If a queried variable does not exist, a null value is returned.

Variables are named by character strings, and have character string values, which may be static or dynamic. For instance, a server might have built-in variables like "system name" which never changes, or others like "mail status" which, when queried, cause the server to check to see if the user has any new mail.

### 6.2.7. Host Commands

Host commands are conceptually simple, but may be hard to implement on some systems. The C packet contains a text string in its data field which is simply fed to the server's host system command processor; any output from the processor is sent back to the user in Kermit packets, as either a short or long reply.

Implementation of this facility under UNIX, with its forking process structure and i/o redirection via pipes, is quite natural. On other systems, it could be virtually impossible.

### 6.2.8. Exchanging Parameters Before Server Commands

In basic Kermit, the Send-Init exchange is always sufficient to configure the two sides to each other. During server operation, on the other hand, some transactions may not begin with a Send-Init packet. For instance, when the user sends an R packet to ask the server to send a file, the server chooses what block check option to use. Or if the user requests a directory listing, the server does not know what packet length to use.

The solution to this problem is the "I" (Init-Info) packet. It is exactly like a Send-Init packet, and the ACK works the same way too. However, receipt of an I packet does not cause transition to file-send state. The I-packet exchange simply allows the two sides to set their parameters, in preparation for the next transaction.

Servers should be able to receive and ACK "I" packets when in server command wait. User Kermits need not send "I" packets, however; in that case, the server will assume all the defaults for the user listed on page 21, or whatever parameters have been set by other means (e.g. SET commands typed to the server before it was put in server mode).

User Kermits which send I packets should be prepared to receive and ignore an Error packet in response. This could happen if the server has not implemented I packets.

The I packet, together with its ACK, constitute a complete transaction, separate from the S-packet or other exchange that follows it. The packet number remains at zero after the I-packet exchange.

## 6.3. Alternate Block Check Types

There are two optional kinds of block checks:

Type 2
: A two-character checksum based on the low order 12 bits of the arithmetic sum of the characters in the packet (from the LEN field through the last data character, inclusive) as follows:

```
                1              2
--------+----------------+---------------+
...data | tochar(b6-b11) | tochar(b0-b5) |
--------+----------------+---------------+
```

For instance, if the 16-bit result is 154321 (octal), then the 2 character block check would be "`C1`".

Type 3
: Three-character 16-bit CRC-CCITT. The CRC calculation treats the data it operates upon as a string of bits with the low order bit of the first character first and the high order bit of the last character last. The initial value of the CRC is taken as 0; the 16-bit CRC is the remainder after dividing the data bit string by the polynomial $X^{16}+X^{12}+X^{5}+1$ (this calculation can actually be done a character at a time, using a simple table lookup algorithm). The result is represented as three printable characters at the end of the packet, as follows:

```
                1                2               3
--------+-----------------+----------------+---------------+
...data | tochar(b12-b15) | tochar(b6-b11) | tochar(b0-b5) |
--------+-----------------+----------------+---------------+
```

For instance, if the 16-bit result is 154321 (octal), then the 3 character block check would be "`-C1`". The CRC technique chosen here agrees with many hardware implementations (e.g. the VAX CRC instruction).

Here is an algorithm for Kermit's CRC-CCITT calculation:

```
    crc = 0                                   Start CRC off at 0
    i = <position of LEN field>               First byte to include

A:  c = <byte at position i>                  Get current byte
    if (parity not NONE) then c = c AND 127;  Mask off any parity bit
    q = (crc XOR c) AND 15;                   Do low-order 4 bits
    crc = (crc / 16) XOR (q * 4225);
    q = (crc XOR (c / 16)) AND 015;           And high 4 bits
    crc = (crc / 16) XOR (q * 4225);
    i = i + 1                                 Position of next byte
    LEN = LEN - 1                             Decrement packet length
    if (LEN > 0) goto A                       Loop till done

At this point, the crc variable contains the desired quantity.
```

Thanks to Andy Lowry of Columbia's CS department for this "tableless" CRC algorithm (actually, it uses a table with one entry -- 4225). `AND` is the bitwise AND operation, `XOR` the bitwise exclusive OR, "`*`" is multiplication, and "`/`" signifies integer division ("`crc / 16`" is equivalent to shifting the `crc` quantity 4 bits to the right).

The single-character checksum has proven quite adequate in practice. The other options can be used only if both sides agree to do so via Init packet (S or I) exchange. The 2 and 3 character block checks should only be used under conditions of severe line noise and packet corruption.

Since type 2 and 3 block checks are optional, not all Kermits can be expected to understand them. Therefore, during initial connection, communication must begin using the type 1 block check. If type 2 or 3 block checks are agreed to during the "I" or "S" packet exchange, the switch will occur *only after* the Send-Init has been sent and ACK'd with a type 1 block check. This means that the first packet with a type 2 or 3 block check must always be an "F" or

"X" packet. Upon completion of a transaction, both sides must switch back to type 1 (to allow for the fact that neither side has any way of knowing when the other side has been stopped and restarted). The transaction is over *after* a "B" or "E" packet has been sent and ACK'd, or after any error that terminates the transaction prematurely or abnormally.

A consequence of the foregoing rule is that if a type 2 or 3 block check is to be used, a long reply sent by the server *must* begin with a Send-Init (S) packet, even if an I packet exchange had already occurred. If type 1 block checks are being used, the S packet can be skipped and the transfer can start with an X or F packet.

A server that has completed a transaction and is awaiting a new command may send out periodic NAKs for that command (packet 0). Those NAKs must have type 1 block checks.

The use of alternate block check types can cause certain complications. For instance, if the server gets a horrible error (so bad that it doesn't even send an error packet) and reverts to command wait, sending NAKs for packet 0 using a type 1 block check, while a transfer using type 2 or 3 block checks was in progress, neither side will be able to read the other's packets. Communication can also grind to a halt if A sends a Send-Init requesting, say, type 3 block checks, B ACKs the request, switches to type 3 and waits for the X or F packet with a type 3 block check, but the ACK was lost, so A resends the S packet with a type 1 block check. Situations like this will ultimately resolve themselves after the two sides retransmit up to their retry threshhold, but can be rectified earlier by the use of two heuristics:

- The packet reader can assume that if the packet type is "S", the block check type is 1.
- A NAK packet never has anything in its data field. Therefore, the block check type can always be deduced by the packet reader from the length field of a NAK. In fact, it is the value of the length field minus 2. A NAK can therefore be thought of as a kind of "universal synchronizer".

These heuristics tend to violate the layered nature of the protocol, since the packet reader should normally be totally unconcerned with the packet type (which is of interest to the application level which invokes the packet reader). A better design would have had each packet include an indicator of the type of its own block check; this would have allowed the block check type to be changed dynamically during a transaction to adapt to changing conditions. But it's too late for that now...

## 6.4. Interrupting a File Transfer

This section describes an optional feature of the Kermit protocol to allow graceful interruption of file transfer. This feature is unrelated to server operation.

To interrupt sending a file, send an EOF ("Z") packet in place of the next data packet, including a "D" (for Discard) in the data field. The recipient ACKs the Z packet normally, but does not retain the file. This does not interfere with older Kermits on the receiving end; they will not inspect the data field and will close the file normally. The mechanism can be triggered by typing an interrupt character at the console of the sending Kermit program. If a (wildcard) file group is being sent, it is possible to skip to the next file or to terminate the entire batch; the protocol is the same in either case, but the desired action could be selected by different interrupt characters, e.g. CTRL-X to skip the current file, CTRL-Z to skip the rest of the batch.

To interrupt receiving a file, put an "X" in the data field of an ACK for a Data packet. To interrupt receiving an entire file group, use a "Z". The user could trigger this mechanism by typing an interrupt character, say, CTRL-X and CTRL-Z, respectively, at the receiving Kermit's console. A sender that was aware of the new feature, upon finding one of these codes, would act as described above, i.e. send a "Z" packet with a "D" code; a sender that did not implement this feature would simply ignore the codes and continue sending. In this case, and if the user wanted the whole batch to be cancelled (or only one file was being sent), the receiving Kermit program, after determining that the sender had ignored the "X" or "Z" code, could send an Error (E) packet to stop the transfer.

The sender may also choose to send a Z packet containing the D code when it detects that the file it is sending cannot be sent correctly and completely -- for instance, after sending some packets correctly, it gets an i/o error reading the file. Or, it notices that the "8th bit" of a file byte is set when the file is being sent as a text file and no provision has been made for transmitting the 8th bit.

## 6.5. Transmitting File Attributes

The optional Attributes (A) packet provides a mechanism for the sender of a file to provide additional information about it. This packet can be sent if the receiver has indicated its ability to process it by setting the Attributes bit in the capability mask. If both sides set this bit in the Kermit capability mask, then the sender, after sending the filename in the "F" packet and receiving an acknowledgement, may (but does not have to) send an "A" packet to provide file attribute information.

Setting the Attributes bit in the capability mask does *not* indicate support for any particular attributes, only that the receiver is prepared to accept the "A" packet.

The attributes are given in the data field of the "A" packet. The data field consists of 0 or more subfields, which may occur in any order. Each subfield is of the following form:

```
+-----------+---------------+------+
| ATTRIBUTE | tochar(LENGTH) | DATA |
+-----------+---------------+------+
```

where

ATTRIBUTE
: is a single printable character other than space,

LENGTH
: is the length of the data characters (0 to 94), with 32 added to produce a single printable character, and

DATA
: is *length* characters worth of data, all printable characters.

No quoting or prefixing is done on any of this data.

More than one attribute packet may be sent. The only requirement is that all the A packets for a file must immediately follow its File header (or X) packet, and precede the first Data packet.

There may be 93 different attributes, one for each of the 93 printable ASCII characters other than space. These are assigned in ASCII order.

! (ASCII 33)
: Length. The data field gives the length in K (1024) bytes, as a printable decimal number, e.g. "!#109". This will allow the receiver to determine in advance whether there is sufficient room for the file, and/or how long the transfer will take.

" (ASCII 34)
: Type. The data field can contain some indicator of the nature of the file. Operands are enclosed in {braces}, optional items in [brackets]. The braces and brackets do not actually appear in the packet.

A[{xx}] ASCII text, containing no 8-bit quantities, logical records (lines) delimited by the (quoted) control character sequence {xx}, represented here by its printable counterpart (MJ = CRLF, J = LF, etc). For instance AMJ means that the appearance of `#M#J` (the normal prefixed CRLF sequence) in a file data packet indicates the end of a record, assuming the current control prefix is "#". If {xx} is omitted, MJ will be assumed.

B[{xx}] Binary. {xx} indicates in what manner the file is binary:

8 (default) The file is a sequence of 8-bit bytes, which must be saved as is. The 8th bit may be sent "bare", or prefixed according to the Send-Init negotiation about 8th-bit prefixing.

36 The file is a PDP-10 format binary file, in which five 7-bit bytes are fit into one 36-bit word, with the final bit of each word being represented as the "parity bit" of every 5th

character (perhaps prefixed).

D{x} *Moved from here to FORMAT attribute*

F{x} *Moved from here to FORMAT attribute*

I[{x}] Image. The file is being sent exactly as it is represented on the system of origin. For use between like systems. There are {x} usable bits per character, before prefixing. For instance, to send binary data from a system with 9-bit bytes, it might be convenient to send three 6-bit characters for every two 9-bit bytes. Default {x} is 8.

# (ASCII 35)
: Creation Date, expressed as `"[yy]yymmdd[ hh:mm[:ss]]"` (ISO standard date format), e.g. `831009 23:59`. The time is optional; if given, it should be in 24-hour format, and the seconds may be omitted, and a single space should separate the time from the date.

`$` (ASCII 36)
: Creator's ID, expressed as a character string of the given length.

`%` (ASCII 37)
: Account to charge the file to, character string.

`&` (ASCII 38)
: Area in which to store the file, character string.

`'` (ASCII 39)
: Password for above, character string.

`(` (ASCII 40)
: Block Size. The file has, or is to be stored with, the given block size.

`)` (ASCII 41)
: Access:

- N New, the normal case -- create a new file of the given name.
- S Supersede (overwrite) any file of the same name.
- A Append to file of the given name.

`*` (ASCII 42)
: Encoding:

- A ASCII, normal ASCII encoding with any necessary prefixing, etc.
- H Hexadecimal "nibble" encoding.
- E EBCDIC (sent as if it were a binary file).
- X Encrypted.
- Q{x}
  Huffman Encoded for compression. First *x* bytes of the file are the key.

`+` (ASCII 43)
: Disposition (operands are specified in the syntax of the receiver's host system):

M{user(s)} Send the file as Mail to the specified user(s).

O{destination} Send the file as a lOng terminal message to the specified destination (terminal, job, or user).

S[{options}] Submit the file as a batch job, with any specified options.

P[{options}] Print the file on a system printer, with any specified options, which may specify a particular printer, forms, etc.

T Type the file on the screen.

L[{aaa}] Load the file into memory at the given address, if any.

X[{aaa}] Load the file into memory at the given address and eXecute it.

A Archive the file; save the file together with the attribute packets that preceded it, so that it can be sent back to the system of origin with all its attributes intact. A file stored in this way should be specially marked so that the Kermit that sends it back will recognize the attribute information as distinct from the file data.

, (ASCII 44)
: Protection. Protection code for the file, in the syntax of the receiver's host file system. With no operand, store according to the system's default protection for the destination area.

- (ASCII 45)
: Protection. Protection code for the file with respect to the "public" or "world", expressed generically in a 6-bit quantity (made printable by `tochar()`), in which the bits have the following meaning:

b0: Read Access
b1: Write Access
b2: Execute Access
b3: Append Access
b4: Delete Access
b5: Directory Listing

A one in the bit position means allow the corresponding type of access, a zero means prohibit it. For example, the letter "E" in this field would allow read, execute, and directory listing access (`unchar("E") = 69-32 = 37 = 100101` binary).

. (ASCII 46)
: Machine and operating system of origin. This is useful in conjunction with the archive disposition attribute. It allows a file, once archived, to be transferred among different types of systems, retaining its archive status, until it finds its way to a machine with the right characteristics to de-archive it. The systems are denoted by codes; the first character is the major system designator, the second designates the specific model or operating system. A third character may be added to make further distinctions, for instance operating system version. The systems below do not form a complete collection; many more can and probably will be added.

A Apple microcomputers

1 Apple II, DOS
2 Apple III
3 Macintosh
4 Lisa

B Sperry (Univac) mainframes

1 1100 series, EXEC
2 9080, VS9

C CDC mainframes

1 Cyber series, NOS
2 Cyber series, NOS-BE
3 Cyber series, NOS-VE
4 Cyber series, SCOPE

D DEC Systems

1 DECsystem-10/20, TOPS-10
2 DECsystem-10/20, TOPS-20
3 DECsystem-10/20, TENEX
4 DECsystem-10/20, ITS
5 DECsystem-10/20, WAITS
6 DECsystem-10/20, MAXC
7 VAX-11, VMS
8 PDP-11, RSX-11
9 PDP-11, IAS
A PDP-11, RSTS/E
B PDP-11, RT-11
C Professional-300, P/OS
D Word Processor (WPS or DECmate), WPS

E Honeywell mainframes

1 MULTICS systems
2 DPS series, running CP-6
3 DPS series, GCOS
4 DTSS

F Data General machines

1 RDOS
2 AOS
3 AOS/VS

G PR1ME machines, PRIMOS

H Hewlett-Packard machines

1 HP-1000, RTE
2 HP-3000, MPE

I IBM 370-series and compatible mainframes

1 VM/CMS
2 MVS/TSO
3 DOS
4 MUSIC
5 GUTS
6 MTS

J Tandy microcomputers, TRSDOS

K Atari computers

1 Home computers, DOS
2 ST series

L Commodore micros

1 Pet
2 64
3 Amiga

M Miscellaneous mainframes and minis with proprietary operation systems:

1 Gould/SEL minis, MPX
2 Harris, VOS
3 Perkin-Elmer minis, OS/32
4 Prime, Primos
5 Tandem, Nonstop
6 Cray, CTSS
7 Burroughs (subtypes may be necessary here)
8 GEC 4000, OS4000
9 ICL machines
A Norsk Data, Sintran III
B Nixdorf machines

N Miscellaneous micros and workstations:

1 Acorn BBC Micro
2 Alpha Micro
3 Apollo Aegis
4 Convergent, Burroughs, and similar systems with CTOS, BTOS
5 Corvus, CCOS
6 Cromemco, CDOS
7 Intel x86/3x0, iRMX-x86

8 Intel MDS, ISIS
9 Luxor ABC-800, ABCDOS
A Perq
B Motorola, Versados

O-T *Reserved*

U Portable Operating or File Systems

1 UNIX
2 Software Tools
3 CP/M-80
4 CP/M-86
5 CP/M-68K
6 MP/M
7 Concurrent CP/M
8 MS-DOS
9 UCSD p-System
A MUMPS
B LISP
C FORTH
D OS-9

/ (ASCII 47)
Format of the data within the packets.

A{xx} Variable length delimited records, terminated by the character sequence {xx}, where xx is a string of one or more control characters, represented here by their unprefixed printable equivalents, e.g. `MJ` for `^M^J` (CRLF).

D{x} Variable length undelimited records. Each logical record begins with an {x}-character ASCII decimal length field (similar to ANSI tape format "D"). For example, "`D$`" would indicate 4-digit length fields, like "0132".

F{xxxx} Fixed-length undelimited records. Each logical record is {xxxx} bytes long.

R{x} For record-oriented transfers, to be used in combination with one of the formats given above. Each record begins (in the case of D format, after the length field) with an x-character long position field indicating the byte position within the file at which this record is to be stored.

M{x} For record-oriented transfers, to be used in combination with one of the formats given above. Maximum record length for a variable-length record.

0 (ASCII 48)
Special system-dependent parameters for storing the file on the system of origin, for specification of exotic attributes not covered explicitly by any of the Kermit attribute descriptors. These are given as a character string in the system's own language, for example a list of DCB parameters in IBM Job Control Language.

1-@ (ASCII 49)
Exact byte count of the file as it is stored on the sender's system, before any conversions (e.g. to canonic form). Of limited usefulness when transferring text files between systems that represent text boundaries differently.

2-@ (ASCII 50-64)
*Reserved*

Other attributes can be imagined, and can be added later if needed. However, two important points should be noted:

- The receiver may have absolutely no way of honoring, or even recording, a given attribute. For instance, CP/M-80 has no slot for creation date or creator's ID in its FCB; the DEC-20 has no concept of block size, etc.

- The sender may have no way of determining the correct values of any of the attributes. This is particularly true when sending files of foreign origin.

The "A" packet mechanism only provides a way to send certain information about a file to the receiver, with no provision or guarantee about what the receiver may do with it. That information may be obtained directly from the file's directory entry (FCB, FDB, `...`), or specified via user command.

The ACK to the "A" packet may in turn have information in its data field. However, no complicated negotiations about file attributes may take place, so the net result is that the receiver may either refuse the file or accept it. The receiver may reply to the "A" packet with any of the following codes in the data field of the ACK packet:

<null> (empty data field) I accept the file, go ahead and send it.

N[{xxx}] I refuse the file as specified, don't send it; {xxx} is a string of zero or more of the attribute characters listed above, to specify what attributes I object to (e.g. "`!`" means it's too long, "&" means I don't have write access to the specified area, etc).

Y[{xxx}] I agree to receive the file, but I cannot honor attributes {xxx}, so I will store the file according to my own defaults.

Y (degenerate case of Y{xxx}, equivalent to <null>, above)

How the receiver actually replies is an implementation decision. A NAK in response to the "A" packet means, of course, that the receiver did not receive the "A" correctly, not that it refuses to receive the file.

## 6.6. Advanced Kermit Protocol State Table

The simple table presented previously is sufficient for a basic Kermit implementation. The following is a state table for the full Kermit protocol, including both server mode and sending commands to a server Kermit. It does not include handling of the file attributes packet (A). Note that states whose names start with "Send" always send a packet each time they are entered (even when the previous state was the same). States whose name starts with "Rec", always wait for a packet to be received (up to the timeout value), and process the received packet. States whose names do not include either send or receive do not process packets directly. These are states which perform some local operation and then change to another state.

The initial state is determined by the user's command. A "server" command enters at `Rec_Server_Idle`. A "send" command enters at `Send_Init`. A "receive" command (the old non-server version, not a "get" command) enters at `Rec_Init`. Any generic command, the "get" command, and the "host" command enter at either `Send_Server_Init` or `Send_Gen_Cmd`, depending upon the expected response.

Under "Rec'd Msg", the packet type of the incoming message is shown, followed by the packet number in parentheses; (n) means the current packet number, (n-1) and (n+1) mean the previous and next packet numbers (modulo 64), (0) means packet number zero. Following the packet number may be slash and a letter, indicating some special signal in the data field. For instance `Z(n)/D` indicates a Z (EOF) packet, sequence number *n*, with a "D" in the data field.

Under "Action", "`r+`" means that the retry count is incremented and compared with a threshhold; if the threshhold is exceeded, an Error packet is sent and the state changes to "Abort". "`n+`" means that the packet number is incremented, modulo 64, and the retry count, *r*, is set back to zero.

```
State   Rec'd Msg        Action                  Next state

Rec_Server_Idle -- Server idle, waiting for a message

    Set n and r to 0

        I(0)             Send ACK                Rec_Server_Idle
        S(0)             Process params,
                          ACK with params, n+    Rec_File
        R(0)             Save file name          Send_Init
```

```
        K, C or G(0)    Short reply:
                         ACK(0)/reply           Rec_Server_Idle
                        Long reply:
                         init needed            Send_Init
                         init not needed, n+    Open_File

        Timeout         Send NAK(0)             Rec_Server_Idle
        Other           Send E                  Rec_Server_Idle
```

Rec_Init -- *Entry point for non-server RECEIVE command*

```
    Set n and r to 0

        S(0)            Process params, send
                         ACK with params, n+    Rec_File
        Timeout         Send NAK(0), r+         Rec_Init
        Other           Send E                  Abort
```

Rec_File -- *Look for a file header or EOT message*

```
        F(n)            Open file, ACK, n+      Rec_Data
        X(n)            Prepare to type on
                         screen, ACK, n+        Rec_Data
        B(n)            ACK                     Complete
        S(n-1)          ACK with params, r+     Rec_File
        Z(n-1)          ACK, r+                 Rec_File
        Timeout         Resend ACK(n), r+       Rec_File
        Other           Send E                  Abort
```

Rec_Data -- *Receive data up to end of file*

```
        D(n)            Store data, ACK, n+;
                         If interruption wanted
                         include X or Z in ACK  Rec_Data
        D(n-1)          Send ACK, r+            Rec-Data
        Z(n)            Close file, ACK, n+     Rec_File
        Z(n)/D          Discard file, ACK, n+   Rec_File
        F(n-1)          Send ACK, r+            Rec_Data
        X(n-1)          Send ACK, r+            Rec_Data
        Timeout         Send ACK(n-1), r+       Rec_Data
        Other           Send E                  Abort
```

Send_Init -- *Also entry for SEND command*

```
    Set n and r to 0, send S(0) with parameters

        Y(0)            Process params, n+      Open_File
        N, Timeout      r+                      Send_Init
        Other           r+                      Send_Init
```

Open_File -- *Open file or set up text to send*

```
                                                Send_File
```

Send_File -- *Send file or text header*

```
    Send F or X(n)

        Y(n), N(n+1)    Get first buffer of     Send_Data or Send_Eof if
                         data, n+                empty file or text
        N, Timeout      r+                      Send_File
        Other                                   Abort
```

Send_Data -- *Send contents of file or textual information*

```
    Send D(n) with current buffer
```

```
        Y(n), N(n+1)    n+, Get next buffer      Send_Data or Send_Eof if
                                                  at end of file or text
        Y(n)/X or Z     n+                       Send_Eof
        N, Timeout      r+                       Send_Data
        Other                                    Abort

Send_Eof -- Send end of file indicator

    Send Z(n); if interrupting send Z(n)/D

        Y(n), N(n+1)    Open next file, n+       Send_File if more, or
                                                 Send_Break if no more
                                                  or if interrupt "Z".
        N, Timeout      r+                       Send_Eof
        Other                                    Abort

Send_Break -- End of Transaction

    Send B(n)

        Y(n), N(0)                               Complete
        N(n), Timeout                            Send_Break
        Other                                    Abort

Send_Server_Init - Entry for Server commands which expect large response.

    Send I(0) with parameters

        Y(0)            Process params           Send_Gen_Cmd
        N, Timeout      r+                       Send_Server_Init
        E               Use default params       Send_Gen_Cmd
        Other                                    Abort

Send_Gen_Cmd - Entry for Server commands which expect short response (ACK)

    Send G, R or C(0)

        S(0)            Process params,
                         ACK with params, n+     Rec_File
        X(1)            Setup to type on
                         terminal, n+            Rec_Data
        Y(0)            Type data on TTY         Complete
        N, Timeout      r+                       Send_Gen_Cmd
        Other                                    Abort

Complete -- Successful Completion of Transaction

        Set n and r to 0;
        If server, reset params, enter Rec_Server_Idle
        otherwise exit

Abort -- Premature Termination of Transaction

    Reset any open file, set n and r to 0

        If server, reset params, enter Rec_Server_Idle
        otherwise exit

Exit, Logout states
        Exit or Logout
```

Note that the generic commands determine the next state as follows:

1. If the command is not supported, an error packet is sent and the next state is "Abort".

2. If the command generates a response which can be fit into the data portion of an ACK, an ACK is sent with the text (quoted as necessary) in the data portion.

3. If the command generates a large response or must send a file, nothing is sent from the

Rec_Server_Idle state, and the next state is either Send_Init (if either no I message was received or if alternate block check types are to be used), or Open_File (if an I message was received and the single character block check is to be used).

4. If the command is Logout, an ACK is sent and the new state is Logout.

5. If the command is Exit, an ACK is sent and the new state is Exit.

# 7. Performance Extensions

The material in this chapter was added in 1985-86 to address the inherent performance problems of a stop-and-wait protocol like Kermit.

## 7.1. Long Packets

A method is provided to allow the formation of long Kermit packets. Questions as to the desirability or appropriateness of this extension to the Kermit protocol are not addressed. All numbers are in decimal (base 10) notation, all arithmetic is integer arithmetic.

In order for long packets to be exchanged, the sender must set the bit for Capability #5 (the LONGP bit) in the CAPAS field of the Send-Init (S or I) packet,

```
 bit5 bit4 bit3 bit2 bit1 bit0
+----+----+----+----+----+----+
| #1 | #2 | #3 | #4 | #5 |  0 |
+----+----+----+----+----+----+
                       ^
                       |
                     LONGP
```

and also furnish the MAXLX1 and MAXLX2 (extended length 1 and 2) fields, as follows:

```
    10              CAPAS+1  CAPAS+2  CAPAS+3
---+-------+-     -+--------+--------+--------+
   | CAPAS |  ...  | WINDO  | MAXLX1 | MAXLX2 |
---+-------+-     -+--------+--------+--------+
                    ^
                    |
```

(currently field 11, because CAPAS is still 1 byte)

where WINDO is the window size (a separate Kermit protocol extension), and MAXLX1 and MAXLX2 are each a printable ASCII character in the range SP (space, ASCII 32) to ~ (tilde, ASCII 126), formed as follows:

```
MAXLX1 = tochar(m / 95)
MAXLX2 = tochar(m MOD 95)
```

(where m is the intended maximum length, `/` signifies integer division, and `MOD` is the modulus operator), to indicate the longest extended-length packet it will accept as input. The receiver responds with an ACK packet having the same bit also set in the CAPAS field, and with the MAXLX1 and MAXLX2 fields set to indicate the maximum length packet it will accept.

The maximum length expressible by this construct is 95 x 94 + 94, or 9024.

Since the sender can not know in advance whether the receiver is capable of extended headers, the Send-Init MAXL field must also be set in the normal manner for compatibility.

If the receiver responds favorably to an extended-length packet bid (that is, if its ACK has the LONGP bit set in the CAPAS field), then the combined value of its MAXLX1,MAXLX2 fields is used. If the LONGP bit is set but the MAXLX1,MAXLX2 pair is missing, then the value 500 will be used by default.

If the response is unfavorable (the LONGP bit is not set in the receiver's CAPAS field), then extended headers will not be used and the MAXL field will supply the maximum packet length.

After the Send-Init has been sent and acknowledged with agreement to allow extended headers, all packets up to and including the B or E packet which terminates the transaction (and its acknowledgement) are allowed -- but not required -- to have extended headers; extended and normal packets may be freely mixed by both Kermits.

The normal Kermit packet length field (LEN) specifies the number of bytes to follow, up to and including the block check. Since at least 3 bytes must follow (SEQ, TYPE, and CHECK), a value of 0, 1, or 2 is never encountered in the LEN field of a valid unextended Kermit packet. When extended packets have been negotiated, the LEN field is treated as follows for the duration of the transaction:

- If `unchar(LEN)` > 2 then the packet is a normal, unextended packet.
- If `unchar(LEN)` = 0 then the packet has a "Type 0" extended header.
- If `unchar(LEN)` = 1 or 2, the packet is invalid and should cause an Error.

"Lengths" of 1 and 2 are reserved for future use in Type 1 and 2 extended headers, yet to be specified.

A Type 0 extended packet has the following layout:

```
+------+-----+-----+------+-------+-------+--------+----       ----+-------+
| MARK |     | SEQ | TYPE | LENX1 | LENX2 | HCHECK |  DATA ....    | CHECK |
+------+-----+-----+------+-------+-------+--------+----       ----+-------+
                          | Extended Header        |
```

The blank length field (SP = `tochar(0)`) indicates that the first 3 bytes of what is normally the data field is now an extended header of Type 0, in which the number of bytes remaining in the packet, up to and including the block check, is

    *extended-length* = (95 x unchar(LENX1)) + unchar(LENX2)

and HCHECK is a header checksum, formed exactly like a Type-1 Kermit block check, but from the sum of the ASCII values of the SEQ, TYPE, LENX1, and LENX2 fields:

    *s* = LEN + SEQ + TYPE + LENX1 + LENX2

    HCHECK = tochar((*s* + ((*s* & 192)/64)) & 63)

where & is the bitwise AND operator.

Since the value of the extended length field must be known accurately in order to locate the end of the packet and the packet block check, it is vital that this information not be corrupted before it is used. The header checksum prevents this.

The extended header, like the normal header itself, is *not* prefix-encoded. This is because it is used at datalink level, before decoding takes place. Therefore the entity responsible for building packets must leave 3 spaces at the beginning of the data field, and the datalink function (spack) fills in LENX1, LENX2, and HCHECK based upon the data actually entered into the packet, after encoding. The packet receiving datalink function (rpack) behaves accordingly.

The packet block check is formed in the usual manner, based on all packet bytes beginning with LEN and ending with the last character in the data field. The block check may be Type 1, 2, or 3, depending upon what was negotiated, but longer packets are more likely to be corrupted than shorter ones and should therefore have higher-order block checks if possible. This proposal does not change the way block check type is negotiated, and does not require that Type 2 or 3 block check be implemented.

With long packets, the possibility exists that the arithmetic sum of the characters in a packet will exceed $2^{15}$, and will overflow a 16-bit word, or become negative. The checksum function would have to be modified to guard against this, for instance by always setting the high four bits of the sum to zero before adding in the next byte.

Implementation can be a bit tricky. The Kermit program should be set up to use normal, untextended packets by default -- that is, to mimic the behavior of original, "classic" Kermit. Even when the program believes itself to be capable of sending and receiving long packets, it has no knowledge of what devices may lie along the communication path, whose buffers might not be long enough to accommodate bursts of data of the desired length. Long packets should be elected when the user has explicitly elected them with a SET command. The current SET SEND PACKET-LENGTH <n> command will do; if the number is larger than 94, then the program will -- transparently to the user -- try to negotiate long packets. A finer degree of control can be accomplished by included

SET commands to explicitly enable or disable the use of long packets.

Once long packets are successfully negotiated, the program should be prepared to back off when errors occur, since the very size of the packets may be the cause of the errors. Upon timeout or receipt of a NAK (or extra copies of the previous packet), the sender should be prepared to reconstruct the current packet at, say, half its size, down to some reasonable minimum, before retransmission. Even when the size itself is not the problem, this makes retransmission less painful under noisy conditions.

Long packets and sliding windows may be used at the same time, though the benefits from doing so may not be worth the trouble of coding the dynamic buffer allocation required (for n buffers of size m, negotiated at Send-Init time). It's also worth noting that the benefit/cost ratio of long packets declines after a length of about 1000, at which point the benefit of additional length is less than 1%, and the cost of retransmission is very high.

## 7.2. Sliding Windows

The sliding window extension to Kermit was proposed and developed by a group at The Source Telecomputing in McLean, Virginia, led by Leslie Spira and including Hugh Matlock and John Mulligan, who wrote the following material. Like other extensions, this one is designed for "upward compatibility" with Kermits that do not support this extension.

The windowing protocol as defined for the Kermit file transfer protocol is based on the main premise of continuously sending data packets up to the number defined by a set window size. These data packets are continuously acknowledged by the receive side and the ideal transfer occurs as long as they are transmitted with good checksums, they are transmitted in sequential order and there are no lost data packets or acknowledgements. The various error conditions define the details of the windowing protocol and are best examined on a case basis.

There are five stages that describe the overall sequence of events in the Kermit protocol. Three of these stages deviate from the original protocol in order to add the windowing feature. Stages 1 through 5 are briefly described on the following page. The three stages (1, 3 and 4) which deviate from the original protocol are then described in greater detail in the pages that follow.

### 7.2.1. Overall Sequence of Events

<u>STAGE 1 - Propose and Accept Windowing</u>

The send side requests windowing in the transmission of the Send-Initiate (S) packet. The receive side accepts windowing by sending an acknowledgement (ACK packet) for the Send-Initiate packet.

<u>STAGE 2 - Send and Accept File-Header Packet</u>

The send side transmits the File-Header (F) packet and waits for the receive side to acknowledge it prior to transmitting any data.

<u>STAGE 3 - Transfer Data</u>

The sending routine transmits Data (D) packets one after the other until the protocol window is closed. The receiving side ACKs good data, stores data to disk as necessary and NAKs bad data.

When the sender receives an ACK, the window may be rotated and the next packet sent. If the sender receives a NAK, the data packet concerned is retransmitted.

<u>STAGE 4 - Send and Accept End_of_File Packet</u>

As the sender is reading the file for data to send, it will eventually reach the end of the file. It then waits until all outstanding data packets have been acknowledged, and then sends an End-of_File (Z) packet.

When the receive side gets the End-of-File packet it stores the rest of the data to disk, closes the file, and ACKs the End-of_File packet.

The protocol then returns to Stage 2, sending and acknowledging any further File-Header (F) packets.

STAGE 5 - End of Transmission

Once the End-of-File packet has been sent and acknowledged and there are no more files to send, the sender transmits the End-of-Transmission (B) packet in order to end the ongoing transaction. Once the receiver ACKs this packet, the transaction is ended and the logical connection closed.

## Stage 1 - Propose and Accept Windowing

The initial connection as currently defined for the Kermit protocol will need to change only in terms of the contents of the Send-Initiate packet. The receiving Kermit waits for the sending Kermit to transmit the Send-Initiate (S) packet and the sending packet does not proceed with any additional transmission until the ACK has been returned by the receiver.

The contents of the Send-Init packet, however, will be slightly revised. The data field of the Send-Init packet currently contains all of the configuration parameters. The first six fields of the Send-Init packet are fixed as follows:

```
  1        2        3        4        5        6
 +--------+--------+--------+--------+--------+--------+
 | MAXL   | TIME   | NPAD   | PADC   | EOL    | QCTL   |
 +--------+--------+--------+--------+--------+--------+
```

Fields 7 through 10 are optional features of Kermit and fields 7 through 9 will also remain unchanged as defined for the existing protocol:

```
  7        8        9        10
 +--------+--------+--------+--------+
 | QBIN   | CHKT   | REPT   | CAPAS  |
 +--------+--------+--------+--------+
```

The windowing capability constitutes a fourth capability and the fourth bit of the capability field will be set to 1 if the Kermit implementation can handle windowing:

```
   bit5 bit4 bit3 bit2 bit1 bit0
  +----+----+----+----+----+----+
  | #1 | #2 | #3 | #4 | #5 |  0 |
  +----+----+----+----+----+----+
                    ^
                    |
                   SWC  (sliding window capability)
```

The remaining fields of the Send-Init packet are either reserved for future use by the standard Kermit protocol or reserved for local site implementations. The four fields following the capability field are reserved for the standard Kermit protocol. The field following the capability mask is used to specify the "Window Size":

```
      10             CAPAS+1  CAPAS+2  CAPAS+3
  ---+-------+-     -+--------+--------+--------+
     | CAPAS |  ...  | WINDO  | MAXLX1 | MAXLX2 |
  ---+-------+-     -+--------+--------+--------+
                      ^
                      |
                    (currently field 11, because CAPAS is still 1 byte)
```

WINDO is the window size to be used, encoded printably using the `tochar()` function. The window size may range from 1 to 31 inclusive.

The sender will specify the window size it wishes to use and the receiver will reply (in the ACK packet) with the window size it wishes to use. The window size actually used will be the minimum of the two. If the receiver replies with a window size of 0 then no windowing will be done.

## Stage 3 - Transfer Data

The sequence of events required for the transmission of data packets and confirmation of receipts constitute the main functions of the windowing protocol. There are four main functions which can be identified within this stage. These are:

- the sender's processing of the data packets,
- the receiver's handling of incoming packets,
- the sender's handling of the confirmations,
- the error handling on both sides.

The following discussion details the specific actions required for each of these functions. Refer to the state table at the end of this document for the specific action taken on a "received message" basis for the full protocol.

The Sender's Processing of Data Packets

The sender instigates the transmission by sending the first data packet and then operating in a cyclical mode of sending data until the defined window is closed.

Data to be sent must be read from the file, encoded into the Kermit Data packet, and saved in a Send-Table. A Send-Table entry consists of the data packet itself (which makes convenient the re-send of a NAK'd packet), a bit which keeps track of whether the packet has been ACK'd (the ACK'd bit), and a retry counter. The table is large enough to hold all the packets for the protocol window.

Before each transmission, the input buffer is checked and input is processed, as described below. Transmission is stopped if the protocol window "closes", that is, if the Send-Table is full.

The Receiver's Handling of Incoming Packets

The receiver keeps its own table as it receives incoming data packets. This allows the receiver to receive subsequent packets while it is waiting for a re-send of an erroneous or lost packet. In other words, the incoming packets do not have to be received in sequential order and can still be written to disk in order.

A Receive-Table entry consists of the data packet, a bit which keeps track of whether a good version of the packet has been received (the ACK'd bit), and a retry counter for the NAKs we send to request retransmissions of the packet. The table is large enough to hold all the packets for the protocol window.

The different possibilities for a received packet are:

1. A new packet, the next sequential one (the usual case)
2. A new packet, not the next sequential one (some were lost)
3. An old packet, retransmitted
4. An unexpected data packet
5. Any packet with a bad checksum

These are now discussed separately:

1. The next new packet has sequence number <one past the latest table entry>. The packet is ACK'd, and the Receive-Table is checked for space. If it is full (already contains window_size entries) then the oldest entry is written to disk. (This entry should have the ACK'd bit set. If not, the receiver aborts the file transfer.) The received packet is then stored in the Receive-Table, with the ACK'd bit set.

2. If the packet received has sequence number in the range <two past the latest table entry> to <window_size past the latest table entry> then it is a new packet, but some have been lost. (The upper limit here represents the highest packet the sender could send within its protocol window. Note that the requirement to test for this case is what limits the maximum window_size to half of the range of possible sequence numbers) We ACK the packet, and NAK all packets that were skipped. (The skipped packets are those from <one past the latest table entry> to <one before the received packet>)

The Receive-Table is then checked. The table may have to be rotated to accomodate the packet, as with case 1. (This time, several table entries may have to be written to disk. As before, if any do not have the ACK'd bit set, they will trigger an abort.) The packet is then stored in the table, and the ACK'd bit set.

3. A retransmitted packet will have sequence number in the range <the oldest table entry> to <the latest table entry>. The packet is ACK'd, then placed in the table, setting the ACK'd bit.

4. A packet with sequence number outside of the range from <the oldest table entry> to <window_size past the latest table entry> is ignored.

5. If the packet received has a bad checksum, we must decide whether to generate a NAK, and if so, with what sequence number. The best action may depend on the configuration and channel error rate. For now, we adopt the following heuristic: If there are unACK'd entries in our Receive-Table, we send a NAK for the oldest one. Otherwise we ignore the packet. (Notice that this will occur in a common case: when things have been going smoothly and one packet gets garbled. In this case, when we later receive the next packet we will NAK for this one as described under Case 2 above.)

### The Sender's Handling of Confirmations

The sender's receipt of confirmations controls the rotation of the Send-Table and normally returns the sender to a sending state. The sender's action depends on the packet checksum, the type of confirmation (ACK or NAK), and whether the confirmation is within the high and low boundaries of the Send-Table.

If the checksum is bad the packet is ignored.

When the sender receives an ACK, the sequence number is examined. If the sequence number is outside of the current table boundaries, then the ACK is also ignored. If the sequence number is inside of the current table boundaries then the ACK'd bit for that packet is marked. If the entry is at the low boundary, this enables a "rotation" of the table. The low boundary is changed to the next sequential entry for which the ACK'd bit is not set. This frees space in the table to allow further transmissions.

When the sender receives a NAK, the table boundaries are checked. A NAK outside of the table boundary is ignored and a NAK inside the table boundary indicates that the sender must re-send the packet. The sender first tests the packet's retry counter against the retry threshold. If the threshold has been reached, then the transfer is stopped (by going to the Abort state). Otherwise, the retry counter is incremented and the packet re-sent.

### Error Handling for Both Sides

Three situations are discussed here: Sender timeout, Receiver timeout, and invalid packets.

If certain packets are lost, each side may "hang", waiting for the other. To get things moving when this happens each may have a "timeout limit", the longest they will wait for something from the other side.

If the sender's timeout condition is triggered, then it will send the oldest unACK'd packet. This will be the first one in the Send-Table.

If the receiver's timeout condition is triggered, then it will send a NAK for the "most desired packet". This is defined as either the oldest unACK'd packet, or if none are unACK'd, then the next packet to be received (sequence number <latest table entry plus one>). The packet retry count is not incremented by this NAK; instead we depend on the timeout retry count, discussed next.

For either the sender or receiver, the timeout retry count is incremented each time a timeout occurs. If the timeout retry limit is exceeded then the side aborts the file transfer. Each side resets the retry count to zero whenever they receive a packet.

In addition, as with the existing Kermit, any invalid packet types received by either side will cause an Error packet and stop the file transfer.

## Stage 4 - Send and Accept End of File Packet

There are several ways to end the file transfer. The first is the normal way, when the sender encounters an end-of-file condition when reading the file to get a packet for transmission. The second is because of a sender side user interrupt. The third is because of a receiver side user interrupt. Both of these cause the received file to be discarded. In addition either side may stop the transfer with an Error packet if an unrecoverable error is encountered.

<u>Normal End of File Handling</u>

When the sender reaches the end of file, it must wait until all data packets have been acknowledged before sending the End-of-File (Z) packet. To do this it must be able to check the end-of-file status when it processes ACKs. If the ACK causes the Send-Table to be emptied and the end-of-file has been reached, then a transition is made to the Send_Eof state which sends the End_of_File packet.

When the receiver gets the End_of_File packet, it writes the contents of the Receive-Table to the file (suitably decoded) and closes the file. (If any entries do not have the ACK'd bit set, or if errors occur in writing the file, the receiver aborts the file transfer.) If the operation is successful, the receiver sends an ACK. It then sets its sequence number to the End_of_File packet sequence number and goes to Rcv_File state.

## File Transfer Interruptions

<u>Sender User Interrupt</u>

Whenever the sender checks for input from the data communications line, it should also check for user input. If that indicates that the file transfer should be stopped, the sender goes directly to the Send_Eof state and sends an End_of_File packet with the Discard indication. It will not have to wait for outstanding packets to be ACK'd.

When the receiver gets the End_of_File packet with the Discard indication it discards the file, sets its sequence number to the End_of_File packet sequence number, and goes to RcvFile state.

<u>Receiver User Interrupt</u>

Whenever the receiver checks for input from the data communications line, it also should check for user input. If that indicates that the file transfer should be stopped, the receiver sets an "interrupt indication" of X (for "stop this file transfer") or of Z (for "stop the batch of file transfers"). When the receiver later sends an ACK, it places an X or Z in the data field.

When the sender gets this ACK, it goes to the Send_Eof state and sends the End_of_File packet with the Discard indication, as above.

When the receiver gets the End_of_File packet with the Discard indication, it discards the file, sets its sequence number to the End_of_File packet sequence number, and goes to RcvFile state.

## Low Level Protocol Requirements

The windowing protocol makes certain assumptions about the underlying transmission and reception mechanism.

First, it must provide a full-duplex channel so that messages may be sent and received simultaneously.

Second, it will prove advantageous to be able to buffer several received messages at the low level before processing them at the Kermit level. This is for two reasons. The first is that the Kermit windowing level of the protocol may take a while to process one input, and meanwhile several others may arrive. The second reason is to support XON/XOFF flow control. If Kermit receives an XOFF from the data communications line, it must wait for an XON before sending its packet. While it is waiting, the low level receive must be able to accept input. Otherwise a deadlock situation could arise with each side flow controlled, waiting for the other.

### Kermit Windowing Protocol State Table

The following table shows the inputs expected, the actions performed, and the succeeding states for the Send_Data_Windowing and Rcv_Data_Windowing states.

If both sides agree on windowing in the Send Init exchange, then instead of entering the old Send_Data or Rcv_Data states from Send_File or Rcv_File, we enter the new Send_Data_Windowing or Rcv_Data_Windowing.

```
SEND_DATA_WINDOWING (SDW)

Rec'd Msg                 Action                             Next State

No input/Window closed   (1) Wait for input                      SDW
No input/Window open     (2) Read file, encode packet,           SDW
                             Place in table, mark unACK'd,
                             Send packet

ACK/ X or Z              (3) set interrupt indicator (X/Z)  Send_Eof
ACK/outside table         -ignore-                               SDW
ACK/inside table         (4) mark pkt ACK'd,         SDW or Send_Eof
                             if low rotate table,
                             if file eof & table empty
                                then goto Send_Eof

NAK/outside table         -ignore-                               SDW
NAK/inside table         (5) test retry limit,                   SDW
                             re-send DATA packet

Bad checksum             -ignore-                                SDW

Timeout                  (6) re-send oldest unACK'd pkt          SDW

User interrupt           (7) set interrupt indicator (X/Z)  Send_Eof

Other                    (8) send Error                         Quit
```

```
RCV_DATA_WINDOWING (RDW)

Rec'd Msg                 Action                             Next State

DATA/new                 (1) send ACK                            RDW
                             if table full: file & rotate
                             store new pkt in table
DATA/old                 (2) send ACK, store in table            RDW
DATA/unexpected          -ignore-                                RDW

Z/discard                (3) discard file                   Rcv_File
Z/                       (4) write table to file & close    Rcv_File
                             if OK send ACK, else Error     or Quit

Bad checksum             (5) send NAK for oldest unACK'd         RDW

Timeout                  (6) send NAK for most desired pkt       RDW

User Interrupt           (7) Set interrupt indicator X or Z      RDW

Other                    (8) send Error pkt                     Quit
```

### 7.2.2. Questions and Answers about Sliding Windows

**Q.** What is the purpose of the "windowing" extension?

**A.** The object is to speed up file transfers using Kermit. The increase will be especially noticeable over the data networks (such as Telenet and Tymnet) and over connections using satellite links. This is because there are long communications delays over these connections.

**Q.** How does it work?

**A.** Basically, it allows you to send several packets out in a row before getting the first acknowledgment back. The number of packets that can be sent out is set by the "window size", hence the name windowing.

**Q.** Could you explain in more detail?

**A.** Right now, a system sending a file transmits one packet of data, then does nothing more until it gets back an acknowledgment that the packet has been received. Once it gets an acknowledgment, it sends the next packet of data. Over standard direct-dial land-based phone lines, the transmission delays are relatively small. However, the public data networks or satellite links can introduce delays of up to several seconds round trip. As a result, the sending system ends up spending much more time waiting than actually sending data.

With the new windowing enhancement, the sending system will be able to keep sending data continuously, getting the acknowledgments back later. It only has to stop sending data if it reaches the end of the current "window" without getting an acknowledgment for the first packet in the current "window".

**Q.** What size is the "window"?

**A.** The window size can vary depending on what the two ends of the connection agree on. The suggested standard window size will be 8 packets. The maximum is 31 packets.

The Kermit sequence numbering is modulo 64 (it "wraps" back to the 1st sequence number after the 64th sequence number). It is helpful to limit the maximum window size to 31 to avoid problems (ambiguous sequence numbers) under certain error conditions.

**Q.** Is windowing in effect throughout a Kermit session?

**A.** No, it is only in effect during the actual data transfer (data packets) portion of a file transfer. Windowing begins with the first data packet (D packet type), and stops when you get an End-of-File packet (Z packet type).

**Q.** Why does it stop when you get to the End-of-File packet?

**A.** This is done primarily to avoid having more than one file open at once.

**Q.** Why will windowing be especially helpful at higher baud rates over communications paths that have delays?

**A.** As you increase the baud rate, the transmission speed of the data increases, but you do not change the delay caused by the communications path. As a result, the delay becomes more and more significant.

Assume, for example, that your communications path introduces a delay of 1 second each way for packets, for a total delay of 2 seconds round trip. Assume also that your packets have 900 bits in them so it takes you 3 seconds to send a packet at 300 baud (this is roughly equivalent to a typical Kermit packet).

WITHOUT windowing, here is what happens:

If at 300 baud you transmitted data for 3 seconds (sending 900 bits), then waited 2 seconds for each acknowledgment, your throughput would be roughly 180 baud. (Total time for each transmission = 5 seconds. 900/5 = 180).

However, if you went to 2400 baud, you would transmit data for 3/8 second, then wait 2 seconds for an acknowledgment. (Total time for each transmission = 2 and 3/8 seconds). The throughput would increase only to about 378 baud. (900 / 2.375 = 378).

The delay becomes the limiting factor; in this case, with this packet size, the delay sets an outside limit of 450 baud (900 / 2 second delay = 450), no matter how fast the modem speed.

WITH windowing, the throughput should be close to the actual transmission speed. It should be possible to send data nearly continuously. The exact speed will depend on the window size, length of transmission delays, and error rate.

**Q.** Are there any new packet types introduced by this extension?

**A.** No, the only change is to the contents of the Send-Init packet, to arrange for windowing if both sides can do it. If either side cannot, Kermit will work as it does now. Adding an extension such as this was provided for in the original Kermit definition. See section 3 of the windowing definition for details.

**Q.** On the receive side, in section 4.2, why does the definition say that writing to disk is done when the Receive-Table becomes full rather than as soon as you get a good packet?

**A.** The definition was phrased this way because it makes the logic of the receive side clearer and simpler to implement.

Actually, you could also write a packet to disk when it is a good packet and it is the earliest entry in the receive table. This approach has the disadvantage that you don't know at this point that the sender has received your ACK, so you have to be prepared to handle the same packet later on if the sender never gets the ACK, times out, and sends the same packet again. Thus you have to be prepared to deal with packets previous to the current window; you will have to ACK such a packet if it has been received properly before.

By writing packets to disk only when the receive table becomes full, (the oldest packet) you know that the sender has received your ACK (otherwise the sender could not have rotated the window to the n+1 position to send the current packet, where n is the window size). This makes it very easy to stay in synch with the sender. The disadvantage of this approach is that when you receive the End-of-File packet, you have to take the time to write all the remaining packets in the Receive-Table to disk.

**Q.** Could you briefly explain what happens if a single packet gets corrupted?

**A.** In essence, the receiver will ignore the bad packet. When it gets the next good packet, it will realize (because packets are numbered) that one or more packets were lost, and NAK those packets. The receiver continues to accept good data.

As long as the sender's window does not become "blocked", the only loss of throughput will be the time it takes to transmit the NAK'd packets.

**Q.** There are currently two proposals for Kermit extensions: the Windowing extension and a proposal for extended packet lengths. What are the relative advantages and disadvantages of sliding windows and extended packet lengths?

**A.** What is best depends on the exact conditions and systems involved in a particular file transfer. There are some general rules however.

Windowing helps more and more as the communications path delays get longer.

Windowing is also more and more helpful as the baud rate goes up.

Increased packet length is most helpful on circuits with low error rates. If the error rate is high, it is difficult for a long packet to get through uncorrupted. Also, it then takes longer to re-transmit the corrupted packet.

On some machines, the CPU time to process a packet is relatively constant no matter what the packet length, so longer packets can reduce CPU time.

**Q.** Are extended packet lengths and sliding windows mutually exlusive?

**A.** No, there is no real reason that they would have to be. As a practical matter, it is slightly easier to implement windowing if you know the maximum packet size ahead of time, since you can then just use an array to store your data. In standard Kermit, you know automactically that your maximum packet length is 94, so you can just go ahead and dimension an array at 94 by Window-size.

If you are going to use both extended packet length and windowing, you need to select the maximum packet length and window-size so that the combination does not exceed the available memory for each side of the transfer.

In addition, it is possible to see the desired relationship between packet size and windowing for various baud rates and communications delays. For the common case of an error corrected by one retransmission of the corrupted packet, the minimum window size needed for continuous throughput (the window never gets "blocked") can be calculated by:

```
                   4 x delay x baud rate
WS  >   1 +  -----------------------
                   packet-size x 10
```

(this is the # of bits)

Windowing always helps (the minimal continuous throughput window size is always greater than 1).

In the above equation, the "4" derives from the fact that a corrupted packet has 4 transit times involved:

- Original (bad checksum) packet
- NAK for the packet
- Retransmission of packet
- ACK for retransmission.

All of this must happen before the window becomes blocked.

The "delay" is the effective maximum one-way communications path delay, which includes any CPU delays.

Strictly speaking, the "packet-size" should have the length of the ACK packets added to it.

As an example, if you assume a 2-second (one-way) delay, at 1200 baud, with a packet size of 94, the minimum window size for continuous throughput would be:

```
           4 x 2 x 1200
WS  >  ------------     =   10.2
            94 x 10
```

Under these circumstances, a window size of at least 11 should be chosen, if possible.

### 7.2.3. More Q-and-A About Windows

While reading the following questions and answers, keep in mind that the Kermit windowing definiton was developed to handle a common situation of long circuit delays with possible moderate error rates. Kermit does not need this type of extension for clean lines with insignificant delays - Kermit could be left alone, or use Extended Packet Lengths, in such environments.

Long delays with significant error rates will occur under two obvious and common conditions:

1. Local phone line (of uncertain quality) to Public Data Networks (such as Telenet).

2. Satellite phone links. These often occur with the lower-priced phone services, which often also have noisier lines. In addition, satellite links will increase as more people need to transfer data overseas.

The above conditions will become more common, as well increased baud rates, which make the delays more significant.

As an aside, note that the benefit of Extended Packet Lengths over the Public Data Networks is limited by the number of outstanding bytes the PDN allows. (Internally, the PDNs require end-to-end acknowledgement. They use their own windowing system within the network.) I don't currently know the exact impact of this.

Now on to the questions...

**Q.** Can sliding windows be done on half-duplex channels? Are any modifications to the proposal required?

**A.** An underlying assumption in the development of windowing was that there was a full-duplex channel.

The intent of windowing is to try to keep the sender continuously sending data. Obviously, this is not possible on a half-duplex channel. A better solution for half-duplex channels would be to use an extended packet length.

An attempt to use windowing on half-duplex really is just a way of doing extended packet lengths. The sender would send out a group of packets, then wait and get a group of ACKS. It would be better to simply send out a large packet, which would have less overhead.

**Q.** Is the cost in complexity for sliding windows worth the increase in performance?

**A.** Under the conditions described above (long delays and possibly significant error rates) windowing can increase performance by a factor of 2, 3, or more, especially at higher baud rates. This increase is necessary to make Kermit viable under some conditions. With classic Kermit over the Public Data Networks, I have had througput as low as 250 baud over a 1200 baud circuit (with a negligible error rate). Windowing should allow throughput close to the maximum baud rate.

Windowing is most helpful when the delay is significant in relation to data sending time. Any delay becomes more significant as users move to higher baud rates (2400 baud and beyond).

The complexity of implementing windowing has yet to be fully evaluated. The first implementation (for the IBM PC using C-Kermit) proved to be fairly manageable. It appears that the windowing logic can be implemented so that Kermit Classic uses the same code, but with a window size of 1, which should avoid having to keep separate sections of code.

The windowing definiton was developed with the idea of keeping changes to Kermit to a minimum. No new packet types were developed, ACKs and NAKS were kept the same, and windowing is in effect only during actual data transfer (D packets). We tried to define the protocol so that a window size of 1 was the same as the current classic Kermit.

These factors should help reduce the complexity of implementing windowing. We currently have a working implementation of Kermit for the IBM PC going through testing.

It's fun to see the modem "Send" light stay on constantly!

**Q.** Why doesn't the Windowing proposal use a "bulk ACK"?

**A.** There are a couple of possibilities for ways to use some sort of "bulk" or combined ACK. We looked at them when developing the Windowing definition. We did not see any advantages that outweighed the disadvantages.

Here are two possible ways of changing how ACKs would work:

1. An ACK for any packet would also ACK all previous packets. The concept that an ACK would also ACK all previous packets seems attractive at first, since it would appear to reduce overhead. However, it has a major drawback in that you then must re-synch when you get errors. This is because, once you have an error, you have to send a NAK, then stop and wait for a re-transmission of the NAK'd packet, before you send out any more ACKs. (If you sent out an ACK for a later packet, it would imply that you had received the NAK'd packet. Not until you safely get the re-transmission can you go ahead.) This would negate one of the nicest parts of windowing as it is defined now, which is that the sender can transmit continuously, including during error recovery, as long as the window does not become blocked. It does not appear to us that the reduction in the number of ACKs sent is worth this penalty. In addition, this is a departure from the way ACKs in Kermit work now. It seemed best to make as few changes to Kermit as possible. If this facility turns out to be useful, it would be better to introduce a new packet type (or other means of distinguishing regular ACKs from "Bulk ACKS").

2. A new "Bulk ACK" packet type could be developed. This did not seem to us to be a good idea, since it required defining a new packet type. We were trying to fit windowing in with as few changes to Kermit as possible. A "Bulk ACK", in which one packet could contain a whole string of ACKs and NAKs, also seems like a good idea at first. The penalty here is a little more subtle. First, if you lose a "Bulk ACK" packet, you lose more information and it takes longer to get things flowing smoothly again. Second, and probably more importantly, efficient windowing depends on the window never becoming "blocked" (i.e., the sender can always keep sending). A "Bulk ACK" interferes with this to some extent, because if you have a long delay, the "Bulk ACK" with its multiple individual ACKs may not get back to the sender in time to prevent the window from becoming blocked. With the current definition of windowing, returning an ACK for each packet gets the ACKs (or NAKs) to the sender as soon as possible. This provides the best chance for keeping the window open so that the sender can transmit continually. Once again, remember the conditions under which windowing is most useful: long delays with significant error rates. Under these conditions, individual ACKs have advantages. If these conditions don't apply, it may not be necessary to use windowing, or it may be better to use extended packet lengths.

# 8. Kermit Commands

The following list of Kermit commands and terms is suggested. It is not intended to recommend a particular style of command parsing, only to promote a consistent vocabulary, both in documentation and in choosing the names for commands.

## 8.1. Basic Commands

SEND
: This verb tells a Kermit program to send one or more files from its own file structure.

RECEIVE
: This verb should tell a Kermit program to expect one or more files to arrive.

GET
: This verb should tell a user Kermit to send one or more files. Some Kermit implementations have separate RECEIVE and GET commands; others use RECEIVE for both purposes, which creates confusion.

Since it can be useful, even necessary, to specify different names for source and destination files, these commands should take operands as follows (optional operands in [brackets]):

SEND local-source-filespec [remote-destination-filespec]
: If the destination file specification is included, this will go in the file header packet, instead of the file's local name.

RECEIVE [local-destination-filespec]
: If the destination filespec is given, the incoming file will be stored under that name, rather than the one in the file header pakcet.

GET remote-source-filespec [local-destination-filespec]
: If the destination filespec is given, the incoming file will be stored under that name, rather than the one in the file header packet.

If a file group is being sent or received, alternate names should *not* be used. It may be necessary to adopt a multi-line syntax for these commands when filespecs may contain characters that are also valid command field delimiters.

## 8.2. Program Management Commands

EXIT
: Leave the Kermit program, doing whatever cleaning up must be done -- deassigning of devices, closing of files, etc.

QUIT
: Leave the Kermit program without cleaning up, in such a manner as to allow further manipulation of the files and devices.

PUSH
: Preserve the current Kermit environment and enter the system command processor.

TAKE
: Read and execute Kermit program commands from a local file.

LOG
: Specify a log for file transfer transactions, or for terminal session logging.

## 8.3. Terminal Emulation Commands

CONNECT
: This verb, valid only for a local Kermit, means to go into terminal emulation mode; present the illusion of being directly connected as a terminal to the remote system. Provide an "escape character" to allow the user to "get back" to the local system. The escape character, when typed, should take a single-character argument; the following are suggested:

    0 (zero) Transmit a NUL
    B Transmit a BREAK
    C Close the connection, return to local Kermit command level

P Push to system command processor
Q Quit logging (if logging is being done)
R Resume logging
S Show status of connection
? Show the available arguments to the escape character
*(a second copy of the escape character)*: Transmit the escape character itself

Lower case equivalents should be accepted. If any invalid argument is typed, issue a beep.

Also see the SET command.

## 8.4. Special User-Mode Commands

These commands are used only by Users of Servers.

BYE This command sends a message to the remote server to log itself out, and upon successful completion, terminate the local Kermit program.

FINISH This command causes the remote server to shut itself down gracefully without logging out its job, leaving the local Kermit at Kermit command level, allowing the user to re-CONNECT to the remote job.

## 8.5. Commands Whose Object Should Be Specified

Some Kermit implementations include various local file management services and commands to invoke them. For instance, an implementation might have commands to let you get directory listings, delete files, switch disks, and inquire about free disk space without having to exit and restart the program. In addition, remote servers may also provide such services. A user Kermit must be able to distinguish between commands aimed at its own system and those aimed at the remote one. When any confusion is possible, such a command may be prefixed by one of the following "object prefixes":

REMOTE
Ask the remote Kermit server to provide this service.

LOCAL Perform the service locally.

If the "object prefix" is omitted, the command should be executed locally. The services include:

LOGIN This should be used in its timesharing sense, to create an identity ("job", "session", "access", "account") on the system.

LOGOUT
To terminate a session that was initiated by LOGIN.

COPY Make a new copy of the specified file with the specified name.

CWD Change Working Directory. This is ugly, but more natural verbs like CONNECT and ATTACH are too imprecise. CWD is the ARPAnet file transfer standard command to invoke this function.

DIRECTORY
Provide a list of the names, and possibly other attributes, of the files in the current working directory (or the specified directory).

DELETE Delete the specified files.

ERASE This could be a synomym for DELETE, since its meaning is clear.

(It doesn’t seem wise to include UNDELETE or UNERASE in the standard list; most systems don’t support such a function, and users’ expectations should not be toyed with...)

KERMIT Send a command to the remote Kermit server in its own interactive command syntax.

RENAME
Change the name of the specified file.

TYPE Display the contents of the specified file(s) at the terminal.

SPACE Tell how much space is used and available for storing files in the current working directory (or the specified directory).

SUBMIT Submit the specified file(s) for background (batch) processing.

PRINT Print the specified file(s) on a printer.

MOUNT Request a mount of the specified tape, disk, or other removable storage medium.

WHO Show who is logged in (e.g. to a timesharing system), or give information about a specified user or network host.

MAIL Send electronic mail to the specified user(s).

MESSAGE
Send a terminal message (on a network or timesharing system).

HELP Give brief information about how to use Kermit.

SET Set various parameters relating to debugging, transmission, file mode, and so forth.

SHOW Display settings of SET parameters, capabilities in force, etc.

STATISTICS
Give information about the performance of the most recent file transfer -- elapsed time, effective baud rate, various counts, etc.

HOST Pass the given command string to the specified (i.e. remote or local) host for execution in its own command language.

LOGGING
Open or close a transaction or debugging log.

## 8.6. The SET Command

A SET command should be provided to allow the user to tailor a connection to the peculiarities of the communication path, the local or remote file system, etc. Here are some parameters that should be SET-able:

BLOCK-CHECK
Specify the type of block check to be used: single character checksum, two-character checksum, 3-character CRC.

DEBUGGING
Display or log the packet traffic, packet numbers, and/or program states. Useful for debugging new versions of Kermit, novel combinations of Kermit programs, etc.

DELAY How many seconds a remote (non-server) Kermit should wait before sending the Send-Init packet, to give the user time to escape back to the local Kermit and type a RECEIVE command.

DISPLAY
Style of file transfer display (NONE, SERIAL, SCREEN, etc).

DUPLEX For terminal emulation, specify FULL or HALF duplex echoing.

END-OF-LINE
Specify any line terminator that must be used after a packet.

ESCAPE Specify the escape character for terminal emulation.

FILE attributes
Almost any of the attributes listed above in the Attributes section (6.5). The most common need is to tell the Kermit program whether an incoming or outbound file is text or binary.

FLOW-CONTROL
Specify the flow control mechanism for the line, such as XON/XOFF, ENQ/ACK, DTR/CTS, etc. Allow flow control to be turned off (NONE) as well as on. Flow control is done only on full-duplex connections.

HANDSHAKE

Specify any line-access negotiation that must be used or simulated during file transfer. For instance, a half duplex system will often need to "turn the line around" after sending a packet, in order to give you permission to reply. A common handshake is XON (^Q); the current user of the line transmits an XON when done transmitting data.

LINE Specify the line or device designator for the connection. This is for use in a Kermit program that can run in either remote or local mode; the default line is the controlling terminal (for remote operation). If an external device is used, local operation is presumed.

LOG Specify a local file in which to keep a log of the transaction. There may be logs for debugging purposes (packet traffic, state transitions, etc) and for auditing purposes (to record the name and disposition of each file transferred).

MARKER
Change the start-of-packet marker from the default of SOH (CTRL-A) to some other control character, in case one or both systems has problems using CTRL-A for this purpose.

PACKET-LENGTH
The maximum length for a packet. This should normally be no less than 30 or 40, and can be greater than 94 only if the long-packet protocol extension is available, in which case it can be a much larger number, up to the maximum size allowed for the particular Kermit program (but never greater than 9024). Short packets can be an advantage on noisy lines; they reduce the probabily of a particular packet being corrupted, as well as the retransmission overhead when corruption does occur. Long packets boost performance on clean lines.

PADDING
The number of padding characters that should be sent before each packet, and what the padding character should be. Rarely necessary.

PARITY Specify the parity (ODD, EVEN, MARK, SPACE, NONE) of the physical connection. If other than none, the "8th bit" cannot be used to transmit data and must not be used by either side in block check computation.

PAUSE How many seconds to pause after receiving a packet before sending the next packet. Normally 0, but when a system communication processor or front end has trouble keeping up with the traffic, a short pause between packets may allow it to recover its wits; hopefully, something under a second will suffice.

PREFIX Change the default prefix for control characters, 8-bit characters, or repeated quantities.

PROMPT
Change the program’s prompt. This is useful when running Kermit between two systems whose prompt is the same, to eliminate confusion about which Kermit you are talking to.

REPEAT-COUNT-PROCESSING
Change the default for repeat count processing. Normally, it will be done if both Kermit programs agree to do it.

RETRY The maximum number of times to attempt to send or receive a packet before giving up. The normal number is about 5, but the user should be able to adjust it according to the condition of the line, the load on the systems, etc.

TIMEOUT
Specify the length of the timer to set when waiting for a packet to arrive.

WINDOW-SIZE
Maximum number of unacknowledged packets outstanding, when the sliding window option is available, usually between 4 and 31.

## 8.7. Macros, the DEFINE Command

In addition to the individual set commands, a "macro" facility is recommended to allow users to combine the characteristics of specific systems into a single SET option. For example:

```
DEFINE IBM = PARITY ODD, DUPLEX HALF, HANDSHAKE XON
DEFINE UNIX = PARITY NONE, DUPLEX FULL
DEFINE TELENET = PARITY MARK
```

This could be done by providing a fancy runtime parser for commands like this (which could be automatically TAKEn from the user's Kermit initialization file upon program startup), or simply hardwired into the SET command table.

With these definitions in place, the user would simply type "SET IBM", "SET UNIX", and so forth, to set up the program to communication to the remote system.

# 9. Kermit Programs

## 9.1. Terminal emulation

The local system must be able to act as a terminal so that the user can connect to the remote system, log in, and start up the remote Kermit.

Terminal emulation should be provided by any Kermit program that runs locally, so that the user need not exit and restart the local Kermit program in order to switch between terminal and protocol operation. On smaller systems, this is particularly important for various reasons -- restarting the program and typing in all the necessary SET commands is too inconvenient and time-consuming; in some micros, switching in and out of terminal emulation may cause carrier to drop, etc.

Only bare-bones terminal emulation need be supplied by Kermit; there is no need to emulate any particular kind of "smart" terminal. Simple "dumb" terminal emulation is sufficient to do the job. Emulation of fancier terminals is nice to have, however, to take advantage of the remote system’s editing and display capabilities. In some cases, microcomputer firmware will take care of this. To build emulation for a particular type of terminal into the program, you must interpret and act upon escape sequences as they arrive at the port.

No error checking is done during terminal emulation. It is "outside the protocol"; characters go back and forth "bare". In this sense, terminal emulation through Kermit is no better than actually using a real terminal.

Some Kermit implementations may allow logging of the terminal emulation session to a local file. Such a facility allows "capture" of remote typescripts and files, again with no error checking or correction. When this facility is provided, it is also desirable to have a convenient way of "toggling" the logging on and off.

If the local system does not provide system- or firmware-level flow control, like XON/XOFF, the terminal emulation program should attempt to simulate it, especially if logging is being done.

The terminal emulation facility should be able to handle either remote or local echoing (full or half duplex), any required handshake, and it should be able to transmit any parity required by the remote side or the communication medium.

A terminal emulator works by continuously sampling both console input from the local terminal and input from the communication line. Simple input and output functions will not suffice, however, since if you ask for input from a certain device and there is none available, you will generally block until input *does* become available, during which time you will be missing input from the other device. Thus you must have a way to bounce back and forth regardless of whether input is available. Several mechanisms are commonly used:

- Continuously jump back and forth between the port status register and the console status register, checking the status bits for input available. This is only practical on single-user, single-process systems, where the CPU has nothing else to do.
- Issue an ordinary blocking input request for the port, but enable interrupts on console input, or vice versa.
- Handle port input in one process and console input in another, parallel process. The UNIX Kermit program listed in this manual uses this method.

Any input at the port should be displayed immediately on the screen. Any input from the console should be output immediately to the port. In addition, if the connection is half duplex, console input should also be sent immediately to the screen.

The terminal emulation code must examine each console character to determine whether it is the "escape character". If so, it should take the next character as a special command, which it executes. These commands are described above, in section 8.3.

The terminal emulator should be able to send every ASCII character, NUL through DEL, and it should also be able to transmit a BREAK signal (BREAK is not a character, but an "escape" from ASCII transmission in which a 0 is put on the line for about a quarter of a second, regardless of the baud rate, with no framing bits). BREAK is important when communicating with various systems, such as IBM mainframes.

Finally, it is sometimes necessary to perform certain transformations on the CR character that is normally typed to end a line of input. Some systems use LF, EOT, or other characters for this function. To complicate matters, intervening communications equipment (particularly the public packet-switched networks) may have their own independent requirements. Thus if using Kermit to communicate over, say, TRANSPAC with a system that uses LF for end-of-line, it may be necessary to transform CR into LFCR (linefeed first -- the CR tells the network to send the packet, which will contain the LF, and the host uses the LF for termination). The user should be provided with a mechanism for specifying this transformation, a command like "SET CR *sequence*".

## 9.2. Writing a Kermit Program

Before writing a new implementation of Kermit or modifying an old one, first be sure to contact the Kermit Distribution center at Columbia University to make sure that you're not duplicating someone else's effort, and that you have all the latest material to work from. If you do write or significantly modify (or document) a Kermit program, please send it back to Columbia so that it can be included in the standard Kermit distribution and others can benifit from it. It is only through this kind of sharing that Kermit has grown from its modest beginnings to its present scale.

The following sections provide some hints on Kermit programming.

### 9.2.1. Program Organization

A basic Kermit implementation can usually be written as a relatively small program, self-contained in a single source file. However, it is often the case that a program written to run on one system will be adapted to run on other systems as well. In that case, it is best to avoid having totally divergent sources, because when new features are added to (or bugs fixed in) the system-independent parts of the program -- i.e. to the protocol itself -- only one implementation will reap the benefits initially, and the other will require painful, error-prone "retrofitting" to bring it up to the same level.

Thus, if there is any chance that a Kermit program will run on more than one machine, or under more than one operating system, or support more than one kind of port or modem, etc, it is desirable to isolate the system-dependent parts in a way that makes the common parts usable by the various implementations. There are several approaches:

1. Runtime support. If possible, the program can inspect the hardware or inquire of the system about relevant parameters, and configure itself dynamically at startup time. This is hardly ever possible.

2. Conditional compilation (or assembly). If the number of systems or options to be supported is small, the system dependent code can be enclosed in conditional compilation brackets (like `IF IBMPC .... ENDIF`). However, as the number of system dependencies to be supported grows, this method becomes unwieldy and error-prone -- installing support for system X tends to break the pre-existing support for system Y.

3. Modular composition. When there is a potentially large number of options a program should support, it should be broken up into separate modules (source files), with clearly specified, simple calling conventions. This allows people with new options to provide their own support for them in an easy way, without endangering any existing support. Suggested modules for a Kermit program are:

4.

- System-Indendent protocol handling: state switching, packet formation, encoding (prefixing) and decoding, etc.
- User Interface: the command parser. Putting this in a separate module allows plug-in of command parsers to suit the user's taste, to mimic the style of the host system command parser or some popular application, etc.
- Screen i/o: This module would contain the screen control codes, cursor positioning routines, etc.
- Port i/o: Allows support of various port hardware. This module can define the port status register location, the status bits, and so forth, and can implement the functions to read and write characters at the port.
- Modem control: This module would support any kind of "intelligent" modem, which is not simply a transparent extension of the communications port. Such modems may accept special commands to perform functions like dialing out, redialing a recent number, hanging up, etc., and may need special initialization (for instance, setting modem signals like DTR).
- Console input: This module would supply the function to get characters from the console; it would know about the status register locations and bits, interrupt structure, key-to-character mappings, etc., and could also implement key redefinitions, keystroke macros, programmable function keys, expanded control and meta functions, etc.
- Terminal Emulation: This module would interpret escape sequences in the incoming character stream (obtained from the port i/o module) for the particular type of terminal being emulated and interpret them by making appropriate calls the the screen i/o module, and it would send user typein (obtained from the console input module) out the serial port (again using the port i/o module). Ideally, this module could be replacable by other modules to emulate different kinds of terminals (e.g. ANSI, VT52, ADM3A, etc).
- File i/o: This module contains all the knowledge about the host system's file structure; how to open and close files, perform "get next file" operations, read and write files, determine and set their attributes, detect the end of a file, and so forth, and provides the functions, including buffering, to get a character from a file and put a character to a file. This module may also provide file management services for local files -- directory listings, deleting, renaming, copying, and so forth.
- Definitions and Data: Separate modules might also be kept for compile-time parameter definitions and for global runtime data.

### 9.2.2. Programming Language

The language to be used in writing a Kermit program is more than a matter of taste. The primary consideration is that the language provide the necessary functionality and speed. For instance, a microcomputer implementation of BASIC may not allow the kind of low-level access to device registers needed to do terminal emulation, or to detect console input during file transfer, or even if it can do these things, it might not be able to run fast enough do drive the communication line at the desired baud rate.

The second consideration in choosing a language is portability. This is used in two senses: (1) whether the language is in the public domain (or, equivalently, provided "free" as part of the basic system), and (2) whether it is well standardized and in wide use on a variety of systems. A language that is portable in both senses is to be preferred.

Whatever programming language is selected, it is important that all lines in the program source be kept to 80 characters or less (after expansion of tabs). This is because Kermit material must often be shipped over RJE and other card-format communication links.

In addition, it is important that the names of all files used in creating and supporting a particular Kermit implementation be (possibly a subset) of the form `NAME.TYPE`, where NAME is limited to six characters, and

TYPE is limited to three, and where the NAME of each file begin with a common 2 or 3 character prefix. This is so that all related files will be grouped together in an alphabetic directory listing, and so when all of the hundreds of Kermit related files are placed together on a tape, all names will be both legal and unique, especially on systems (like PDP-11 operating systems) with restrictive file naming conventions.

### 9.2.3. Documentation

A new Kermit program should be thoroughly documented; one of the hallmarks of Kermit is its documentation. The documentation should be at both the user level (how to use the program, what the commands are, etc, similar to the documentation presently found in the *Kermit Users Guide*), and the implementation level (describe system dependencies, give pointers for adapting to new systems, and so forth). In addition, programs themselves should contain copious commentary. Like program source, documentation should be kept within 80-character lines.

If possible, a section for the implementation should be written for the Kermit User Guide using the UNILOGIC Scribe formatting language (subsets of which are also to be found in some microcomputer text processing software such as Perfect Writer or Final Word), using the same general conventions as the existing Scribe-format implementation sections.

Kermit programs should also contain a revision history, in which each change is briefly explained, assigned an "edit number", and the programmer and site are identified. The lines or sections comprising the edit should be marked with the corresponding edit number, and the Kermit program, upon startup, should announce its version and edit numbers, so that when users complain of problems we will know what version of the program is in question.

The version number changes when the functionality has been changed sufficiently to require major revisions of user documentation. The edit number should increase (monotonically, irrespective of version number) whenever a change is made to the program. The edit numbers are very important for program management; after shipping out a version of, say, CP/M Kermit-80, we often receive many copies of it, each containing its own set of changes, which we must reconcile in some manner. Edit numbers help a great deal here.

### 9.2.4. Bootstrapping

Finally, a bootstrap procedure should be provided. Kermit is generally distributed on magnetic tape to large central sites; the users at those sites need ways of "downloading" the various implementations to their micros and other local systems. A simple bootstrap procedure would consist of precise instructions on how to accomplish an "unguarded" capture of the program. Perhaps a simple, short program can be written for each each end that will do the job; listings and instructions can be provided for the user to type in and run these programs.

# Appendix I
# Packet Format and Types

### Basic Kermit Packet Layout

```
       |<------Included in CHECK------>|
       |                               |
+------+-----+-----+------+------ - - -+-------+
| MARK | LEN | SEQ | TYPE | DATA       | CHECK |<terminator>
+------+-----+-----+------+------ - - -+-------+
             |                                 |
             |<--------LEN-32 characters------>|
```

| | |
|---|---|
| MARK | A real control character, usually CTRL-A. |
| LEN | One character, length of remainder of packet + 32, max 95 |
| SEQ | One character, packet sequence number + 32, modulo 64 |
| TYPE | One character, an uppercase letter |
| CHECK | One, two, or three characters, as negotiated. |

<terminator> Any control character required for reading the packet.

### Kermit Extended Packet Layout

```
       |<-------------------------Included in CHECK------------->|
       |                                                         |
       |<-------Included in HCHECK------->|                      |
       |                                  |                      |
+------+-----+-----+------+-------+-------+--------+----- - - - -+-------+
| MARK |     | SEQ | TYPE | LENX1 | LENX2 | HCHECK | DATA        | CHECK |
+------+-----+-----+------+-------+-------+--------+----- - - - -+-------+
        blank                                      |                     |
                                                   |<------------------->|
                    LX1=LENX1-32, LX2=LX2-32         95 x LX1 + LX2 chars
```

```
HCHECK is a single-character type 1 checksum
```

### Initialization String

```
 1       2       3       4       5       6       7       8       9       10
+-------+-------+-------+-------+-------+-------+-------+-------+-------+- -
| MAXL  | TIME  | NPAD  | PADC  | EOL   | QCTL  | QBIN  | CHKT  | REPT  |
+-------+-------+-------+-------+-------+-------+-------+-------+-------+- -

     10              CAPAS+1  CAPAS+2  CAPAS+3
- --+-------+-      -+--------+--------+--------+- -
    | CAPAS    ... 0| WINDO  | MAXLX1 | MAXLX1 |
- --+-------+-      -+--------+--------+--------+- -
```

| | |
|---|---|
| MAXL | Maximum length (0-94) +32 |
| TIME | Timeout, seconds (0-94) +32 |
| NPAD | Number of pad characters (0-94) +32 |
| EOL | Packet terminator (0-63) +32 |
| QCTL | Control prefix, literal |
| QBIN | 8th bit prefix, literal |
| CHKT | Block check type {1,2,3}, literal |
| REPT | Repeat count prefix, literal |
| CAPAS | Extendable capabilities mask, ends when value-32 is even |
| WINDO | Window size (0-31) +32 |
| MAXLX1 | High part of extended packet maximum length (int(max/95)+32) |
| MAXLX2 | Low part of extended packet maximum length (mod(max,95)+32) |

### Packet Types

Y Acknowledgment (ACK). Data according to what kind of packet is being acknowledged.
N Negative Acknowledgment (NAK). Data field always empty.
S Send Initiation. Data field contains unencoded initialization string. Tells receiver to expect files. ACK to this packet also contains unencoded initialization string.
I Initialize. Data field contains unencoded initialization string. Sent to server to set parameters prior to a command. ACK to this packet also contains unencoded initialization string.
F File Header. Indicates file data about to arrive for named file. Data field contains encoded file name. ACK to this packet may contain encoded name receiver will store file under.
X Text Header. Indicates screen data about to arrive. Data field contains encoded heading for display.
A File Attributes. Data field contains unencoded attributes. ACK may contain unencoded corresponding agreement or refusal, per attribute.
D Data Packet. Data field contains encoded file or screen data. ACK may contain X to interrupt sending this file, Z to interrupt entire transaction.
Z End of file. Data field may contain D for Discard.
B Break transmission.
E Error. Data field contains encoded error message.
R Receive Initiate. Data field contains encoded file name.
C Host Command. Data field contains encoded command for host's command processor.
K Kermit Command. Data field contains encoded command for Kermit command processor.
T Timeout psuedopacket, for internal use.
Q Block check error psuedopacket, for internal use.
G Generic Kermit Command. Data field contains a single character subcommand, followed by zero or more length-encoded operands, encoded after formation:

    I Login [<%user[%password[%account]]>]
    C CWD, Change Working Directory [<%directory[%password]>]
    L Logout, Bye
    F Finish (Shut down the server, but don't logout).
    D Directory [<%filespec>]
    U Disk Usage Query [<%area>]
    E Erase (delete) <%filespec>
    T Type <%filespec>
    R Rename <%oldname%newname>
    K Copy <%source%destination>
    W Who's logged in? [<%user ID or network host[%options]>]
    M Send a short Message <%destination%text>
    H Help [<%topic>]
    Q Server Status Query
    P Program <%[program-filespec][%program-commands]>
    J Journal <%command[%argument]>
    V Variable <%command[%argument[%argument]]>

# Appendix II
# List of Features

There's no true linear scale along which to rate Kermit implementations. A basic, minimal implementation provides file transfer in both directions, and, for microcomputers (PC's, workstations, other single user systems), terminal emulation. Even within this framework, there can be variations. For instance, can the program send a *file group* in a single command, or must a command be issued for each file? Can it time out? Here is a list of features that may be present; for any Kermit implementation, the documentation should show whether these features exist, and how to invoke them.

- File groups. Can it send a group of files with a single command, using "wildcard", pattern, or list notation? Can it successfully send or receive a group of files of mixed types? Can it recover from an error on a particular file and go on to the next one? Can it keep a log of the files involved showing the disposition of each?

- Filenames. Can it take action to avoid overwriting a local file when a new file of the same name arrives? Can it convert filenames to and from legal or "normal form"?

- File types. Can binary as well as text files be transmitted?

- 8th-Bit prefixing. Can it send and receive 8-bit data through a 7-bit channel using the prefixing mechanism?

- Repeat-Count processing. Can it send and receive data with repeated characters replaced by a prefix sequence?

- Terminal Emulation. Does it have a terminal emulation facility? Does it emulate a particular terminal? To what extent? Does it provide various communication options, such as duplex, parity, and handshake selection? Can it transmit all ASCII characters? Can it transmit BREAK? Can it log the remote session locally?

- Communications Options. Can duplex, parity, handshake, and line terminator be specified for file transfer?

- Block Check Options. In addition to the basic single-character checksum, can the two-character checksum and the three-character CRC be selected?

- Basic Server. Can it run in server mode, accepting commands to send or receive files, and to shut itself down?

- Advanced Server. Can it accept server commands to delete files, provide directory listings, send messages, and forth?

- Issue Commands to Server. Can it send commands to a server, and handle all possible responses?

- Host Commands. Can it parse and send remote "host commands"? If it is a server, can it pass these commands to the host system command processor and return the results to the local user Kermit?

- Interrupt File Transfers. Can it interrupt sending or receiving a file? Can it respond to interrupt requests from the other side?

- Local File Management Services. Are there commands to get local directory listings, delete local files, and so forth?

- File Attributes. Can it send file attribute information about local files, and can deal with incoming file attribute information? Can alternate dispositions be specified. Can files be archived?

- Long Packets. Is the long packet protocol extension implemented?

- Sliding Windows. Is the sliding window protocol extension implemented?

- Debugging Capability. Can packet traffic be logged, examined, single-stepped?

- Frills. Does it have login scripts? Raw download/upload? A DIAL command and modem control? Phone directories?

# Appendix III
# The ASCII Character Set

ASCII Code (ANSI X3.4-1968)

There are 128 characters in the ASCII (American national Standard Code for Information Interchange) "alphabet". The characters are listed in order of ASCII value; the columns are labeled as follows:

| | |
|---|---|
| Bit | Even parity bit for ASCII character. |
| ASCII Dec | Decimal (base 10) representation. |
| ASCII Oct | Octal (base 8) representation. |
| ASCII Hex | Hexadecimal (base 16) representation. |
| EBCDIC Hex | EBCDIC hexadecimal equivalent for Kermit translate tables. |
| Char | Name or graphical representation of character. |
| Remark | Description of character. |

The first group consists of nonprintable ’control’ characters:

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 0 | 000 | 000 | 00 | 00 | NUL | ^@, Null, Idle |
| 1 | 001 | 001 | 01 | 01 | SOH | ^A, Start of heading |
| 1 | 002 | 002 | 02 | 02 | STX | ^B, Start of text |
| 0 | 003 | 003 | 03 | 03 | ETX | ^C, End of text |
| 1 | 004 | 004 | 04 | 37 | EOT | ^D, End of transmission |
| 0 | 005 | 005 | 05 | 2D | ENQ | ^E, Enquiry |
| 0 | 006 | 006 | 06 | 2E | ACK | ^F, Acknowledge |
| 1 | 007 | 007 | 07 | 2F | BEL | ^G, Bell, beep, or fleep |
| 1 | 008 | 010 | 08 | 16 | BS | ^H, Backspace |
| 0 | 009 | 011 | 09 | 05 | HT | ^I, Horizontal tab |
| 0 | 010 | 012 | 0A | 25 | LF | ^J, Line feed |
| 1 | 011 | 013 | 0B | 0B | VT | ^K, Vertical tab |
| 0 | 012 | 014 | 0C | 0C | FF | ^L, Form feed (top of page) |
| 1 | 013 | 015 | 0D | 0D | CR | ^M, Carriage return |
| 1 | 014 | 016 | 0E | 0E | SO | ^N, Shift out |
| 0 | 015 | 017 | 0F | 0F | SI | ^O, Shift in |
| 1 | 016 | 020 | 10 | 10 | DLE | ^P, Data link escape |
| 0 | 017 | 021 | 11 | 11 | DC1 | ^Q, Device control 1, XON |
| 0 | 018 | 022 | 12 | 12 | DC2 | ^R, Device control 2 |
| 1 | 019 | 023 | 13 | 13 | DC3 | ^S, Device control 3, XOFF |
| 0 | 020 | 024 | 14 | 3C | DC4 | ^T, Device control 4 |
| 1 | 021 | 025 | 15 | 3D | NAK | ^U, Negative acknowledge |
| 1 | 022 | 026 | 16 | 32 | SYN | ^V, Synchronous idle |
| 0 | 023 | 027 | 17 | 26 | ETB | ^W, End of transmission block |
| 0 | 024 | 030 | 18 | 18 | CAN | ^X, Cancel |
| 1 | 025 | 031 | 19 | 19 | EM | ^Y, End of medium |
| 1 | 026 | 032 | 1A | 3F | SUB | ^Z, Substitute |
| 0 | 027 | 033 | 1B | 27 | ESC | ^[, Escape, prefix, altmode |
| 1 | 028 | 034 | 1C | 1C | FS | ^\, File separator |
| 0 | 029 | 035 | 1D | 1D | GS | ^], Group separator |
| 0 | 030 | 036 | 1E | 1E | RS | ^^, Record separator |
| 1 | 031 | 037 | 1F | 1F | US | ^_, Unit separator |

The last four are usually associated with the control version of backslash, right square bracket, uparrow (or circumflex), and underscore, respectively, but some terminals do not transmit these control characters.

The following characters are printable:

First, some punctuation characters.

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 1 | 032 | 040 | 20 | 40 | SP | Space, blank |
| 0 | 033 | 041 | 21 | 5A | ! | Exclamation mark |
| 0 | 034 | 042 | 22 | 7F | " | Doublequote |
| 1 | 035 | 043 | 23 | 7B | # | Number sign, pound sign |
| 0 | 036 | 044 | 24 | 5B | $ | Dollar sign |
| 1 | 037 | 045 | 25 | 6C | % | Percent sign |
| 1 | 038 | 046 | 26 | 50 | & | Ampersand |
| 0 | 039 | 047 | 27 | 7D | ’ | Apostrophe, accent acute |
| 0 | 040 | 050 | 28 | 4D | ( | Left parenthesis |
| 1 | 041 | 051 | 29 | 5D | ) | Right parenthesis |
| 1 | 042 | 052 | 2A | 5C | * | Asterisk, star |
| 0 | 043 | 053 | 2B | 4E | + | Plus sign |
| 1 | 044 | 054 | 2C | 6B | , | Comma |
| 0 | 045 | 055 | 2D | 60 | - | Dash, hyphen, minus sign |
| 0 | 046 | 056 | 2E | 4B | . | Period, dot |
| 1 | 047 | 057 | 2F | 61 | / | Slash |

Numeric characters:

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 0 | 048 | 060 | 30 | F0 | 0 | Zero |
| 1 | 049 | 061 | 31 | F1 | 1 | One |
| 1 | 050 | 062 | 32 | F2 | 2 | Two |
| 0 | 051 | 063 | 33 | F3 | 3 | Three |
| 1 | 052 | 064 | 34 | F4 | 4 | Four |
| 0 | 053 | 065 | 35 | F5 | 5 | Five |
| 0 | 054 | 066 | 36 | F6 | 6 | Six |
| 1 | 055 | 067 | 37 | F7 | 7 | Seven |
| 1 | 056 | 070 | 38 | F8 | 8 | Eight |
| 0 | 057 | 071 | 39 | F9 | 9 | Nine |

More punctuation characters:

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 0 | 058 | 072 | 3A | 7A | : | Colon |
| 1 | 059 | 073 | 3B | 5E | ; | Semicolon |
| 0 | 060 | 074 | 3C | 4C | < | Left angle bracket |
| 1 | 061 | 075 | 3D | 7E | = | Equal sign |
| 1 | 062 | 076 | 3E | 6E | > | Right angle bracket |
| 0 | 063 | 077 | 3F | 6F | ? | Question mark |
| 1 | 064 | 100 | 40 | 7C | @ | "At" sign |

Upper-case alphabetic characters (letters):

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 0 | 065 | 101 | 41 | C1 | A | |
| 0 | 066 | 102 | 42 | C2 | B | |
| 1 | 067 | 103 | 43 | C3 | C | |
| 0 | 068 | 104 | 44 | C4 | D | |
| 1 | 069 | 105 | 45 | C5 | E | |
| 1 | 070 | 106 | 46 | C6 | F | |
| 0 | 071 | 107 | 47 | C7 | G | |
| 0 | 072 | 110 | 48 | C8 | H | |
| 1 | 073 | 111 | 49 | C9 | I | |
| 1 | 074 | 112 | 4A | D1 | J | |
| 0 | 075 | 113 | 4B | D2 | K | |
| 1 | 076 | 114 | 4C | D3 | L | |
| 0 | 077 | 115 | 4D | D4 | M | |
| 0 | 078 | 116 | 4E | D5 | N | |
| 1 | 079 | 117 | 4F | D6 | O | |
| 0 | 080 | 120 | 50 | D7 | P | |
| 1 | 081 | 121 | 51 | D8 | Q | |
| 1 | 082 | 122 | 52 | D9 | R | |
| 0 | 083 | 123 | 53 | E2 | S | |
| 1 | 084 | 124 | 54 | E3 | T | |
| 0 | 085 | 125 | 55 | E4 | U | |
| 0 | 086 | 126 | 56 | E5 | V | |
| 1 | 087 | 127 | 57 | E6 | W | |
| 1 | 088 | 130 | 58 | E7 | X | |
| 0 | 089 | 131 | 59 | E8 | Y | |
| 0 | 090 | 132 | 5A | E9 | Z | |

More punctuation characters:

| | .....ASCII.... | | | EBCDIC | | |
|---|---|---|---|---|---|---|
| Bit | Dec | Oct | Hex | Hex | Char | Remarks |
| 1 | 091 | 133 | 5B | AD | [ | Left square bracket |
| 0 | 092 | 134 | 5C | E0 | \ | Backslash |
| 1 | 093 | 135 | 5D | BD | ] | Right square bracket |
| 1 | 094 | 136 | 5E | 5F | ^ | Circumflex, up arrow |
| 0 | 095 | 137 | 5F | 6D | _ | Underscore, left arrow |
| 0 | 096 | 140 | 60 | 79 | ‘ | Accent grave |

Lower-case alphabetic characters (letters):

| Bit | Dec (ASCII) | Oct (ASCII) | Hex (ASCII) | EBCDIC Hex | Char | Remarks |
|---|---|---|---|---|---|---|
| 1 | 097 | 141 | 61 | 81 | a | |
| 1 | 098 | 142 | 62 | 82 | b | |
| 0 | 099 | 143 | 63 | 83 | c | |
| 1 | 100 | 144 | 64 | 84 | d | |
| 0 | 101 | 145 | 65 | 85 | e | |
| 0 | 102 | 146 | 66 | 86 | f | |
| 1 | 103 | 147 | 67 | 87 | g | |
| 1 | 104 | 150 | 68 | 88 | h | |
| 0 | 105 | 151 | 69 | 89 | i | |
| 0 | 106 | 152 | 6A | 91 | j | |
| 1 | 107 | 153 | 6B | 92 | k | |
| 0 | 108 | 154 | 6C | 93 | l | |
| 1 | 109 | 155 | 6D | 94 | m | |
| 1 | 110 | 156 | 6E | 95 | n | |
| 0 | 111 | 157 | 6F | 96 | o | |
| 1 | 112 | 160 | 70 | 97 | p | |
| 0 | 113 | 161 | 71 | 98 | q | |
| 0 | 114 | 162 | 72 | 99 | r | |
| 1 | 115 | 163 | 73 | A2 | s | |
| 0 | 116 | 164 | 74 | A3 | t | |
| 1 | 117 | 165 | 75 | A4 | u | |
| 1 | 118 | 166 | 76 | A5 | v | |
| 0 | 119 | 167 | 77 | A6 | w | |
| 0 | 120 | 170 | 78 | A7 | x | |
| 1 | 121 | 171 | 79 | A8 | y | |
| 1 | 122 | 172 | 7A | A9 | z | |

More punctuation characters:

| Bit | Dec (ASCII) | Oct (ASCII) | Hex (ASCII) | EBCDIC Hex | Char | Remarks |
|---|---|---|---|---|---|---|
| 0 | 123 | 173 | 7B | C0 | { | Left brace (curly bracket) |
| 1 | 124 | 174 | 7C | 4F | \| | Vertical bar |
| 0 | 125 | 175 | 7D | D0 | } | Right brace (curly bracket) |
| 0 | 126 | 176 | 7E | A1 | ~ | Tilde |

Finally, one more nonprintable character:

| | | | | | | |
|---|---|---|---|---|---|---|
| 0 | 127 | 177 | 7F | 07 | DEL | Delete, rubout |

# Index

# Table of Contents