KERMIT

A Simple File Transfer Protocol for Microcomputers and Mainframes

Frank da Cruz, Bill Catchings

Columbia University Center for Computing Activities
New York, N.Y. 10027
May 1983[1]

During recent years, the technical press has focused a lot of attention on developments in computer networking—the IEEE 802 committee, TCP/IP, SNA, the latest VLSI Ethernet interface, fibre optics, satellite communications, broadband versus baseband. But little attention has been given to the single mechanism that may be the most widely used in the real world for direct interprocessor communication: the so-called ‘‘asynchronous protocol’’, which is to be found in some form at almost every institution where there is a need to transfer files between microcomputers and central computers.

Columbia University is such an institution. Large timesharing computers at a central site are complemented by many smaller systems scattered in the laboratories and departments. The past few years have witnessed the inexorable progress of diverse microcomputers, word processors, and professional workstations into offices and laboratories throughout the University, and into the homes or dormitory rooms of faculty, students, and staff. As soon as these small machines began to appear, their users asked for ways to exchange files with the central and departmental systems.

At the same time, student use of our central systems was growing at an astonishing rate. We could no longer afford to provide students with perpetual online disk storage; we began to issue IDs per course, per term. With the decreased longevity of the IDs came the need for students to economically archive their files. Given a reliable way to from the central mainframes and back, microcomputers with floppy disks could provide inexpensive removable media ideal for this purpose.

The situation called for a file transfer mechanism that could work among all our computers, large and small. We knew of none that could handle the required diversity. Some were intended for use between microcomputers, others between large computers, but none specifically addressed the need for communication among the widest possible range of computers, particularly between micros and our IBM and DEC mainframes. Most commercial packages served a limited set of systems, and their cost would have been prohibitive when multiplied by the large number of machines we needed to support.

So we embarked on our own project. We were not well-versed in these matters at the outset; we learned as we proceeded, and we’re still learning. This article discusses some of the issues and tradeoffs that came up in the design, and illustrates them in terms of our result, the KERMIT protocol for point-to-point file transfer over telecommunication lines. Because commercial local area networking products are expensive, not yet widely available, and unsuitable for one-shot or long-haul applications, humble asynchronous protocols such as KERMIT are likely to be with us for a long time to come.

---

[1]This is the original manuscript of the article published in BYTE Magazine as "Kermit: A File Transfer Protocol for Universities", June and July 1984. Some minor editorial differences exist between this manuscript and the article as published. The Kermit file transfer protocol is named after Kermit the Frog, star of the television series THE MUPPET SHOW, used by permission of Henson Associates, Inc.

It is assumed the reader is familiar with common computing and telecommunications terminology, and with the ASCII alphabet, which is listed at the end of this article for reference.

• The Communication Medium

The only communication medium common to all computers, large and small, is the asynchronous serial telecommunication line, used for connecting terminals to computers. Standards for this medium are almost universally followed—connectors, voltages, and signals (EIA RS-232-C [1]), character encoding (ASCII, ANSI X3.4-1977 [2]), and bit transmission sequence (ANSI X3.15-1976 [3, 4]). Serial connections can be made in many ways: dedicated local lines (‘‘null modem’’ cables), leased telephone circuits, dialup connections. Dialup connections can be initiated manually from the home or office using an inexpensive acoustic coupler, or automatically from one computer to another using a programmable dialout mechanism. The asynchronous serial line offers the ordinary user a high degree of convenience and control in establishing intersystem connections, at relatively low cost.

Once two computers are connected with a serial line, information can be transferred from one machine to the other, provided one side can be instructed to send the information and the other to receive it. But right away, several important factors come into play:

1. *Noise*—It is rarely safe to assume that there will be no electrical interference on a line; any long or switched data communication line will have occasional interference, or noise, which typically results in garbled or extra characters. Noise corrupts data, perhaps in subtle ways that might not be noticed until it’s too late.
2. *Synchronization*—Data must not come in faster than the receiving machine can handle it. Although line speeds at the two ends of the connection may match, the receiving machine might not be able to process a steady stream of input at that speed. Its central processor may be too slow or too heavily loaded, or its buffers too full or too small. The typical symptom of a synchronization problem is lost data; most operating systems will simply discard incoming data they are not prepared to receive.
3. *Line Outages*—A line may stop working for short periods because of a faulty connector, loss of power, or similar reason. On dialup or switched connections, such intermittent failures will cause carrier to drop and the connection to be closed, but for any connection in which the carrier signal is not used, the symptom will be lost data.

Other communication media, such as the parallel data bus, have safeguards built in to prevent or minimize these effects. For instance, distances may be strictly limited, the environment controlled; special signals may be available for synchronization, and so forth. The serial telecommunication line provides no such safeguards, and we must therefore regard it as an intrinsically unreliable medium.

• Getting Reliable Communication over an Unreliable Medium

To determine whether data has been transmitted between two machines correctly and completely, the two machines can compare the data before and after transmission. A scheme that is commonly used for file transfer employs cooperating programs running simultaneously on each machine, communicating in a well-defined, concise language. The sending program divides outbound data into discrete pieces, adding to each piece special information describing the data for the receiving program. The result is called a ‘‘packet’’. The receiver separates the description from the data and determines whether they still match. If so, the packet is acknowledged and the transfer proceeds. If not, the packet is ‘‘negatively acknowledged’’ and the sender retransmits it; this procedure repeats for each packet until it is received correctly.

The process is called a communication *protocol*—a set of rules for forming and transmitting packets, carried out by programs that embody those rules. Protocols vary in complexity; our preference was for a simple approach that could be realized in almost any language on almost any computer by a programmer of moderate skill, allowing the protocol to be adapted easily to new systems.

• Accommodating Diverse Systems

Most systems agree how to communicate at the lowest levels—the EIA RS-232-C asynchronous communication line and the ASCII character set—but there is rarely agreement beyond that. To avoid a design that might lock out some kinds of systems, we must consider certain important ways in which systems can differ.

– *Mainframes vs Micros*

A distinction must first be made between *micros* and *mainframes*. These terms are not used perjoratively; a ‘‘micro’’ could be a powerful workstation, and a ‘‘mainframe’’ could be a small minicomputer. For our purposes, a micro is any single-user system in which the serial communication port is strictly an external device. A mainframe is any system which is ‘‘host’’ to multiple simultaneous terminal users, who log in to ‘‘jobs’’, and where a user’s terminal is the job’s ‘‘controlling terminal’’. Some mainframe systems allow users to ‘‘assign’’ another terminal line on the same machine as an external input/output device.

Mainframe operating system terminal drivers usually treat a job’s controlling terminal specially. Full duplex systems echo incoming characters on the controlling terminal, but not on an assigned line. System command interpreters or user processes might take special action on certain characters on the controlling line, but not on an assigned line (for instance, control-C under CP/M or most DEC operating systems). Messages sent to a job’s controlling terminal from other jobs could interfere with transmission of data. The ability of a system to test for the availability of input on a serial line might depend on whether the line is the job’s controlling terminal or an assigned device; CP/M and IBM VM/370 are examples of such systems. CP/M can test for data *only* at the console, VM can test *anywhere but* the console.

Output to a job’s controlling terminal may be reformatted by the operating system: control characters may be translated to printable equivalents, lower case letters specially flagged or translated to upper case (or vice versa), tabs expanded to spaces. In addition, based on the terminal’s declared ‘‘width’’ and ‘‘length’’, long lines might be ‘‘wrapped around’’ or truncated, formfeeds translated to a series of linefeeds, and the system may want to pause at the end of each screenful of output. Input from a job’s controlling terminal may also be handled specially: lower case letters may be converted to upper case, linefeed may be supplied when carriage return is typed, control characters may invoke special functions like line editing or program interruption. The DECSYSTEM-20 is an example of a computer where any of these might happen.

The moral here is that care must be taken to disable special handling of a mainframe job’s controlling terminal when it is to be a vehicle for interprocessor communication. But some systems simply do not allow certain of these features to be disabled, so file transfer protocols must be designed around them.

– *Line Access*

Line access can be either *full* or *half duplex*. If full duplex, transmission can occur in both directions at once. If half duplex, the two sides must take turns sending, each signaling the other when the line is free; data sent out of turn is discarded, or it can cause a break in synchronization. On mainframes, the host echoes characters typed at the terminal in full duplex, but not in half duplex. Naturally, echoing is undesirable during file transfer. Full duplex systems can usually accommodate half duplex communication, but not vice versa. IBM mainframes are the most prevalent half duplex systems.

– Buffering and Flow Control

Some systems cannot handle sustained bursts of input on a telecommunications line; the input buffer can fill up faster than it can be emptied, especially at high line speeds. Some systems attempt to buffer ‘‘typeahead’’ (unrequested input), while others discard it. Those that buffer typeahead may or may not provide a mechanism to test or clear the buffer.

Systems may try to regulate how fast characters come in using a *flow control* mechanism, either in the data stream (XON/XOFF) or in parallel to it (modem control signals) [5], but no two systems can be assumed to honor the same conventions for flow control, or to do it at all. Even when flow control is being done, the control signals themselves are subject to noise corruption.

Our experiments with several host computers revealed that a burst of more than about a line’s worth of characters (60-100 characters) into a terminal port at moderate speed could result in loss of data—or worse—on some hosts. For instance, the communications front end of the DECSYSTEM-2060 is designed on the statistical assumption that all terminal input comes from human fingers, and it cannot allocate buffers fast enough when this assumption is violated by sending continuous data simultaneously from several microcomputers attached to terminal ports.

– Character Interpretation

Systems can differ in how they interpret characters that arrive at the terminal port. A host can accept some characters as sent, ignore others, translate others, take special action on others. Communications front ends or multiplexers might swallow certain characters (typically DC1, DC3) for flow control, padding (NUL or DEL), or for transfer of control (‘‘escape’’). The characters that typically trigger special behavior are the ASCII control characters, 0-31 and 127. For instance, of these 33 control characters, 17 invoke special functions of our DECSYSTEM-20 command processor. However, all hosts and communication processors we’ve encountered allow any ‘‘printable’’ character (ASCII 32-126) to reach an application program, even though the character may be translated to a different encoding, like EBCDIC [6], for internal use.

Some operating systems allow an application to input a character at a time, others delay passing the characters to the program until a ‘‘logical record’’ has been detected, usually a sequence of characters terminated by carriage return or linefeed. Some record oriented systems like IBM VM/370 discard the terminator, others keep it. And there are different ways of keeping it—UNIX translates carriage return into linefeed; most DEC operating systems keep the carriage return but also add a linefeed.

– Timing Out

Hosts may or may not have the ability to ‘‘time out’’. When exchanging messages with another computer, it is desirable to be able to issue an input request without waiting forever should the incoming data be lost. A lost message could result in a protocol ‘‘deadlock’’ in which one system is waiting forever for the message while the other waits for a response. Some systems can set timer interrupts to allow escape from potentially blocking operations; others, including many microcomputers, can not do so. When timeouts are not possible, they may be simulated by sleep-and-test or loop-and-test operations, or deadlocked systems may be awakened by manual intervention.

– File Organization

Some computers store all files in a uniform way, such as the linear stream of bytes that is a UNIX file. Other computers may have more complicated or diverse file organizations and access methods: record-oriented storage with its many variations, exemplified in IBM OS/360 or DEC RMS. Even simple microcomputers can present complications when files are treated as uniform data to be transferred; for instance under CP/M, the ends of binary and text files are determined differently. A major question in any operating system is whether a file is specified sufficiently by its contents and its name, or if additional external information is required to make the file valid. A simple generalized file transfer facility can be expected to transmit a file's name and contents, but not every conceivable attribute a file might possess.

Designers of expensive networks have gone to great lengths to pass file attributes along when transferring files between unlike systems. For instance, the DECnet Data Access Protocol [7] supports 42 different ‘‘generic system capabilities’’ (like whether files can be preallocated, appended to, accessed randomly, etc), 8 data types (ASCII, EBCDIC, executable, etc), 4 organizations (sequential, relative, indexed, hashed), 5 record formats (fixed, variable, etc), 8 record attributes (for format control), 14 file allocation attributes (byte size, record size, block size, etc), 28 access options (supersede, update, append, rewind, etc), 26 device characteristics (terminal, directory structured, shared, spooled, etc), various access options (new, old, rename, password, etc), in addition to the better known file attributes like name, creation date, protection code, and so on. All this was deemed necessary even when the designers had only a small number of machines to worry about, all from a single vendor.

The ARPA network, which attempts to provide services for many more machines from many vendors, makes some simplifying assumptions and sets some restrictions in its File Transfer Protocol (FTP) [8]. All files are forced into certain categories with respect to encoding (ASCII, EBCDIC, image), record format control, byte size, file structure (record or stream), and it is generally left to the host FTP implementation to do the necessary transformations. No particular provision is made, or can be made, to ensure that such transformations are invertible.

DECnet is able to provide invertibility for operating systems like VMS or RSX, which can store the necessary file attributes along with the file. But simpler file systems, like those of TOPS-10 or TOPS-20, can lose vital information about incoming files. For instance, if VMS requires some type of file to have a specific blocksize, while TOPS-20 has no concept of block size, then the blocksize will be lost upon transfer from VMS to TOPS-20 and cannot be restored automatically when the file is sent back, leaving the result potentially unusable.

Invertibility is a major problem, with no simple solution. Fortunately, most file transfer between unlike systems involves only textual information—data, documents, program source—which is sequential in organization, and for which any required transformations (e.g. blocked to stream, EBCDIC to ASCII) are simple and not dependent on any special file attributes.

In fact, invertability *can* be achieved if that is the primary goal of a file transfer protocol. All the external attributes of a file can be encoded and included with the contents of the file to be stored on the remote system. For unlike systems, this can render the file less than useful on the target system, but allows it to be restored correctly upon return. However, it is more commonly desired that textual files remain intelligible when transferred to a foreign system, even if transformations must be made. To allow the necessary transformations to take place on textual files between unlike systems, there must be a standard way of representing these files during transmission.

– Binary Files versus Parity

Each ASCII character is represented by a string of 7 bits. Printable ASCII files can be transmitted in a straightforward fashion, because ASCII transmission is designed for them: a serial stream of 8-bit characters, 7 bits for data and 1 for parity, framed by start and stop bits for the benefit of the hardware [3]. The parity bit is added as a check on the integrity of a character; some systems always transmit parity, others insist on parity for incoming characters, still others ignore the parity bit for communication purposes and pass it along to the software, while still others discard it altogether. In addition, communication front ends or common carriers might usurp the parity bit, regardless of what the system itself may do.

Computer file systems generally store an ASCII file as a sequence of either 7-bit or 8-bit bytes. 8-bit bytes are more common, in which the 8th bit of each byte is superfluous. Besides files composed of ASCII characters, however, computers also have ‘‘binary’’ files, in which every bit is meaningful; examples include executable ‘‘core images’’ of programs, numbers stored in ‘‘internal format’’, databases with imbedded pointers. Such binary data must be mapped to ASCII characters for transmission over serial lines. When two systems allow the user-level software to control the parity bit, the ANSI standards [2, 3] may be stretched to permit the transmission of 8 data bits per character, which corresponds to the byte size of most machines. But since not all computers allow this flexibility, the ability to transfer binary data in this fashion cannot be assumed.

– *Software*

Finally, systems differ in the application software they have. In particular, no system can be assumed to have a particular programming language. Even widespread languages like FORTRAN and BASIC may be lacking from some computers, either because they have not been implemented, or because they are proprietary and have not been purchased. Even when two different systems support the same language, it is unrealistic to expect the two implementations of the language to be totally compatible. A general purpose file transfer protocol should not be geared towards the features any particular language.

• KERMIT

Our protocol, which we call KERMIT, addresses the problems outlined above by setting certain minimal standards for transmission, and providing a mapping between disk storage organization, machine word and byte size, and the transmission medium.

KERMIT has the following characteristics:

- Communication takes place over ordinary terminal connections.
- Communication is half duplex. This allows both full and half duplex systems to participate, and it eliminates the echoing that would otherwise occur for characters arriving at a host job's controlling terminal.
- The packet length is variable, but the maximum is 96 characters so that most hosts can take packets in without buffering problems.
- Packets are sent in alternate directions; a reply is required for each packet. This is to allow half duplex systems to participate, and to prevent buffer overruns that would occur on some systems if packets were sent back to back.
- A timeout facility, when available, allows transmission to resume after lost packets.
- All transmission is in ASCII. Any non-ASCII hosts are responsible for conversion. ASCII control characters are prefixed and then converted to printable characters during transmission to ensure that they arrive as sent. A single ASCII control character (normally SOH) is used to mark the beginning of a packet.
- Binary files can be transmitted by a similar prefix scheme, or by use of the parity bit when both sides have control of it.
- Logical records (lines) in textual files are terminated during transmission with quoted carriage-return/linefeed sequences, which are transparent to the protocol and may appear anywhere in a packet. Systems that delimit records in other ways are responsible for conversion, if they desire the distinction between records to be preserved across unlike systems.
- Only a file's name and contents are transmitted—no attributes. It is the user's responsibility to see that the file is stored correctly on the target system. Within this framework, invertible transfer of

text files can be assured, but invertible transfer of non-text files depends on the capabilities of the particular implementations of KERMIT and the host operating systems.

- KERMIT has no special knowledge of the host on the other side. No attempt is made to ‘‘integrate’’ the two sides. Rather, KERMIT is designed to work more or less uniformly on all systems.
- KERMIT need not be written in any particular language. It is not a portable program, but a portable protocol.

Thus KERMIT accommodates itself to many systems by conforming to a common subset of their features. But the resulting simplicity and generality allow KERMIT on any machine to communicate with KERMIT on any other machine, micro-to-mainframe, micro-to-micro, mainframe-to-mainframe. The back-and-forth exchange of packets keeps the two sides synchronized; the protocol can be called ‘‘asynchronous’’ only because the communication hardware itself operates asynchronously.

As far as the user is concerned, KERMIT is a do-it-yourself operation. For instance, to transfer files between your micro and a mainframe, you would run KERMIT on your micro, put KERMIT into terminal emulation mode, which ‘‘connects’’ you to the mainframe, log in and run KERMIT on the mainframe, then ‘‘escape’’ back to the micro and issue commands to the micro’s KERMIT to send or fetch the desired files. Any inconvenience implicit in this procedure is a consequence of the power it gives the ordinary user to establish reliable connections between computers that could not otherwise be connected.

• Packets

KERMIT packets need to contain the data that is being transferred, plus minimum information to assure (with high probability) that the expected data arrives completely and correctly. Several issues come up when designing the packet layout: how to represent data, how to delimit fields within the packet, how to delimit the packet itself, how to arrange the fields within the packet. Since the transmission medium itself is character-oriented, it is not feasible to transmit bit strings of arbitrary length, as do the bit-oriented protocols like HDLC and SDLC [5]. Therefore the smallest unit of information in a packet must be the ASCII character. As we will see, this precludes some techniques that are used with other communication media.

– Control Fields

Most popular protocol definitions view the packet as layers of information, which pass through a hierarchy of protocol levels, each level adding its own information at the ends of an outbound packet or stripping its information from the ends of an incoming packet, and then passing the result along to the next level in the hierarchy. The fields for each layer must be arranged so that they can be found, identified, and interpreted correctly at the appropriate level.

Since KERMIT packets are short, it is important to minimize the amount of control information per packet. It would be convenient to limit the control fields to one character each. Since we have 95 printable characters to work with (128 ASCII characters, less DEL and the 32 control characters), we can represent values from 0 to 94 with a single character.

- The *packet sequence number* is used to detect missing or duplicate packets. It is unlikely that a large number of packets could be lost, especially since packet *n* is acknowledged before packet *n+1* is sent. So the sequence number can be a small quantity, which ‘‘wraps around’’ to its minimum value when it exceeds a specified maximum value.
- To prevent long packets, a small maximum length can be enforced by specifying the *packet length*

with a single character; since there are 95 printable ASCII characters, this would be the maximum length, depending on how we count the control fields.

- The *checksum* can be of fixed length. The actual length depends on the desired balance between and efficiency and error detection.

The packet length and checksum act together to detect corrupted, missing, or extra characters. These are the essential fields for promoting error-free transmission. But so far, we've only considered packets that carry actual file data; we will also require special packets composed only of control information, for instance to tell the remote host the name of the file that is about to come, or to tell it that the transmission is complete. This can be accomplished with a *packet type* field. The number of functions we need to specify in this field is small, so a single character can suffice here too.

– Packet Framing

We choose to mark the beginning of a packet with a distinguished start character, SOH (Start Of Header, ASCII 1, Control-A). This character cannot appear anywhere else within the packet. SOH was chosen because, unlike most other control characters, it is generally accepted upon input at a job's controlling terminal as a data character, rather than an interrupt or break character on most mainframes. This is probably no accident, since it was originally intended for this use by the designers of the ASCII alphabet [9]. Should a system be incapable of sending or receiving SOH, it is possible to redefine the start-of-packet character to be any other control character; the two sides need not use the same one.

There are three principal options for recognizing the end of a packet: a fixed length, a distinguished packet-end character, and a length field. There are arguments for and against each involving what happens when characters, particularly a length or terminator, is lost or garbled, which will be mentioned later. KERMIT uses a length field.

To take in a packet, a KERMIT program gets characters from the line until it encounters the SOH. The next character is the length; KERMIT reads and decodes the length and then reads that many subsequent characters to complete the packet. If another SOH is encountered before the count is exhausted, the current packet is forgotten and a new one is started. This stratagy allows arbitrary amounts of noise to be generated spontaneously between packets without interfering with the protocol.

– Encoding

When transmitting textual data, KERMIT terminates logical records with carriage-return linefeed combinations (CRLFs). On record oriented systems, trailing blanks or length fields are removed and a CRLF appended to outbound records, with the inverse operation performed on incoming records. On stream oriented systems, incoming CRLFs may be translated to some other terminator. Files, of course, need not have logical records, in which case record processing can be skipped altogether, and the file can be treated as a long string of bytes. This is known as ''image'' transfer, and can also be used between like systems where no transformations are necessary.

In order to make each character in the packet printable, KERMIT ''quotes'' any unprintable character by transforming it to a printable one and precedes it with a special prefix character. The prefix is normally ''#''; the transformation is done by complementing bit 6 (adding or subtracting 64_{10}, modulo 64). Thus control-A becomes ''#A'', control-Z becomes ''#Z'', US (control-underscore on most terminals) becomes ''#_''. The prefix character is also used to quote itself: ''##''. Upon input, the reverse transformation is performed. Printable characters are not tranformed. The assumption is that most files to be transferred are printable, and printable files contain relatively few control characters; when this is true, the character stream is not significantly lengthened by quoting. For binary files, the average quoting overhead will be 26.6% if all bit patterns are equally likely, since the characters that must be quoted (the control characters, plus DEL, and ''#'' itself) comprise 26.6% of the ASCII alphabet.

KERMIT also provides a scheme for indicating the status of the 8th bit when transferring binary files between systems that must use the 8th bit for parity. A byte whose 8th bit is set is preceded by another special quoting character, ‘‘&’’. If the low-order 7 bits coincide with an ASCII control character, then control-character quoting is also done. For instance, the byte 10000001_2 would be transmitted as ‘‘&#A’’. The ‘‘&’’ character itself can be included as data by quoting it (#&), and the control-quote character may have its 8th bit set (&##). 8th-bit quoting is only done when necessary; if both sides can control the parity bit, then its value is preserved during transmission. If the 8th bit is set randomly on binary files, then 8th-bit quoting will add 50% character overhead. For some kinds of binary data, it could be less; for instance, positive binary numbers in 2’s complement notation do not to have their high-order bits set, in which case at least one byte per word will not be quoted.

A third kind of ‘‘quoting’’ implements rudimentary data compression. At low speeds, the bottleneck in file transmission is likely to be the line itself, so any measure that can cut down on use of the line would be welcome. The special prefix character ‘‘~’’ indicates that the next character is a repeat count (a single character, encoded printably) and that the character after that (which may also have control or 8th-bit prefixes) is repeated so many times. For instance ‘‘~}A’’ indicates a series of 93 letter A’s; ‘‘~H&#B’’ indicates a series of 40 control-B’s with the parity bit set. The repeat count prefix itself can be included as text by quoting it with ‘‘#’’.

To keep the protocol simple, no other transformations are done. At this point, however, it might be worth mentioning some things we did *not* do to the data:

- *Fancy Data compression*. If the data is known to be (or resemble) English text, a Huffman encoding [10, 11] based on the frequency of characters in English text could be used. A Huffman code resembles Morse code, which has variable length characters whose boundaries can always be distinguished. The more frequent the character, the shorter the bit string to represent it. Of course, this scheme can backfire if the character distribution of the data is very different from the one assumed. In any case, variable length characters and ASCII transmission don’t mix well.
- *Error Correcting Codes*. Techniques, such as Hamming codes [12], exist for detecting and correcting errors on a per-character basis. These are expensive in resources and complex to program. KERMIT uses per-packet block check techniques, which are explained below.
- *Nibble Encoding*. To circumvent problems with control and 8-bit characters, it would have been possible to divide every character into two 4-bit ‘‘nibbles’’, sending each nibble as a printable character (e.g. a hexadecimal digit). The character overhead caused by this scheme would would always be 100%. But it would be an easy way to transfer binary files.

– *Error Detection*

Character parity and Hamming codes are forms of ‘‘vertical redundancy checks’’ (VRCs), formed by combining all the bits of a character in one way or another. The other kind of check that can be used is the ‘‘longitudinal redundancy check’’ (LRC), which produces a ‘‘block check character’’ formed by some combination of each character within a sequence. The sending side computes the LRC and sends it with the packet; the receiving side recomputes it for comparison. There are various forms of LRCs. One form produces a ‘‘column parity’’ character, or ‘‘logical sum’’, whose bits are the exclusive-ORs of the corresponding bits of the data characters. Another is the ‘‘checksum’’ which is the arithmetic sum of all the characters in the sequence, interpreted numerically. Another is the ‘‘Cyclic Redundancy Check’’ (CRC) [13, 14], which passes the characters through what amounts to a shift register with imbedded feedback loops, producing a block check in which each bit is effected in many ways by the preceding characters.

All of these techniques will catch single-bit errors. They vary in their ability to detect other kinds of errors. For instance, a double-bit column error will always go undetected with column parity, since the result of XORing any two bits together is the same as XORing their complements, whereas half the possible double bit errors can be caught by addition because of the carry into the next bit position. CRC does even better by rippling the effect of a data bit multiply through the block check character, but the method is complex, and a software implementation of CRC can be inscrutable.

Standard, base-level KERMIT employs a single-character arithmetic checksum, which is simple to program, is low in overhead, and has proven quite adequate in practice. The sum is formed by adding together the ASCII values of each character in the packet except the SOH and the checksum itself, and including any quoting characters. Even non-ASCII hosts must do this calculation in ASCII. The result can approach $12{,}000_{10}$ in the worst case. The binary representation of this number is 10111011100000, which is 14 bits long. This is much more than one character's worth of bits, but we can make the observation that every character included in the sum has contributed to the low order 7 bits, so we can discard some high order bits and still have a viable validity check.

The KERMIT protocol also allows other block check options, including a two-character checksum and a three-character 16-bit CRC. The two-character checksum is simply the low order 12 bits of the arithmetic sum, broken into two printable characters. The CRC sequence is formed from the 16-bit quantity generated by the CCITT-recommended polynomial $X^{16}+X^{12}+X^{5}+1$ which is also used in some form with other popular transmission techniques, like ISO HDLC and IBM SDLC [5]. The high order 4 bits of the CRC go into the first character, the middle 6 into the second, and the low order 6 into the third.

Some care must be taken in the formation of the single-character block check. Since it must be expressed as a single printable character, values of the high order data bits may be lost, which could result in undetected errors, especially when transferring binary files. Therefore, we extract the 7th and 8th bits of the sum and add them back to the low order bits; if the arithmetic sum of all the characters is *S*, then the value of the single-character KERMIT checksum is given by

```
(S + ((S AND 300)/100)) AND 77
```

(numbers are in octal notation). This ensures that the checksum, terse though it is, reflects every bit from every character in the packet.

The probability that an error will not be caught by a correctly transmitted arithmetic checksum is the ratio of the number of possible errors that cancel each other out to the total number of possible errors, which works out to be something like $1/2^n$, where n is the number of bits in the checksum, assuming all errors are equally likely. This is 1/64 for the single character checksum, and 1/4096 for the 2-character checksum. But the probability that errors will go undetected by this method *under real conditions* cannot be easily derived, because all kinds of errors are not equally likely. A 16-bit CRC will detect all single and double bit errors, all messages with an odd number of bits in error, all error bursts shorter than 16 bits, and better than 99.99% of longer bursts [13]. These probabilities all assume, of course, that the block check has been identified correctly, i.e. that the length field points to it, and that no intervening characters have been lost or spuriously added.

A final note on parity—a parity bit on each character combined with a logical sum of all the characters (VRC *and* LRC) would allow detection *and* correction of single-bit errors without retransmission by pinpointing the ‘‘row’’ and ‘‘column’’ of the bad bit. But control of the parity bit cannot be achieved on every system, so we use the parity bit for binary data when we can, or surrender it to the communication hardware if we must. If we have use of the 8th bit for data, then it is figured into the block check; if we do not, then it must be omitted from the block check in case it has been changed by agents beyond the knowledge or control of the KERMIT program.

– Packet Layout

KERMIT packets have the format:

```
+------+----------+----------+------+-----------+-------+
| MARK | char(LEN) | char(SEQ) | TYPE |    DATA    | CHECK |
+------+----------+----------+------+-----------+-------+
|                 |          |                  |       |
|                 |          +-- (application) --+       |
|                 |                              |       |
|                 +-------------- (session) ------+       |
|                                                        |
+-------------------------------- (data link) ------------+
```

where all fields consist of ASCII characters, and the *char* function converts a number in the range 0-94_{10} to a printable ASCII character by adding 32_{10}.

In terms of the ISO network reference model [15], 8-bit bytes are presented to the KERMIT program by the hardware and operating system software comprising the physical link layer. Correct transmission is ensured by the packet-level routines that implement the data link layer using the outer ‘‘skin’’ of the packet—the MARK, LEN, and CHECK fields. The network and transport layers are moot, since KERMIT is a point-to-point affair, in which the user personally makes all the required connections. The session layer is responsible for requesting retransmission of missing packets or ignoring redundant ones, based on the SEQ field; the presentation layer is responsible for any data conversions (EBCDIC/ASCII, insertion or stripping of CRLFs, etc). Finally, the remainder of the packet, the TYPE and DATA fields, are the province of the application layer; our application, of course, is file transfer. In any particular implementation, however, the organization of the program may not strictly follow this model. For instance, since transmission is always in stream ASCII, IBM implementations must convert from EBCDIC and insert CRLFs *before* checksum computation.

The fields of a KERMIT packet are as follows:

<u>MARK</u> Start-of-packet character, normally SOH (ASCII 1).

<u>LEN</u> The number of ASCII characters, including quoting characters and the checksum, within the packet that follow this field, in other words the packet length minus two. Since this number is expressed as a single character via the *char* function, packet character counts of 0 to 94_{10} are permitted, and 96_{10} is the maximum total packet length.

<u>SEQ</u> The packet sequence number, modulo 100_8. The sequence numbers ‘‘wraps around’’ to 0 after each group of 64_{10} packets.

<u>TYPE</u> The packet type, a single printable ASCII character, one of the following:

- D Data
- Y Acknowledge (ACK)
- N Negative Acknowledge (NAK)
- S Send Initiate (Send-Init)
- R Receive Initiate
- B Break Transmission (EOT)
- F File Header
- Z End of file (EOF)
- E Error

- G Generic Command. A single character in the data field, possibly followed by operands, requests host-independent remote execution the specified command:
  - L Logout, Bye.
  - F Finish, but don’t logout.

D Directory query (followed by optional file specification).
U Disk usage query.
E Erase (followed by file specification).
T Type (followed by file specification).
Q Query server status.
and others.

C Host Command. The data field contains a string to be executed as a system dependent (literal) command by the host.

X Text display header, to indicate the arrival of text to be displayed on the screen, for instance as the result of a generic or host command executed at the other end. Operation is exactly like a file transfer.

DATA The ‘‘contents’’ of the packet, if any contents are required in the given type of packet, interpreted according to the packet type. Nonprintable ASCII characters are prefixed with quote characters and then ‘‘uncontrollified’’. Characters with the 8th bit set may also be prefixed, and a repeated character can be prefixed by a count. A prefixed sequence may not be broken across packets.

CHECK The block check sequence, based on all the characters in the packet between, but not including, the mark and the check itself, one, two, or three characters in length as described above, each character transformed by *char*. Normally, the single-character checksum is used.

The packet may be followed by any line terminator required by the host, carriage return (ASCII 15) by default. Line terminators are not 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 for that host.

Here are some sample KERMIT data packets:

```
^AE"D   No celestial body has required J
^AE#Das much labor for the study of its#
^AE$D#M#Jmotion as the moon.  Since ClaA
^AE%Dirault (1747), who indicated a way7
^AE&D of#M#Jconstructing a theory conta5
```

The ‘‘`^A`’’ represents the SOH (or Control-A) character. In the final packet shown, ‘‘`E`’’ is the length. The ASCII value of the ‘‘`E`’’ character is 69_{10}, less 32 (the *unchar* transformation, which is the opposite of *char*) gives a length of 37. The next character tells the packet sequence number, in this case 6 (‘‘&’’ is ASCII 38). The next is the packet type ‘‘`D`’’ for Data. The next characters, ‘‘` of#M#Jconstructing a theory conta`’’, form the data; note the prefixed carriage return and line feed. The final character, ‘‘`5`’’ is the checksum, which represents the number 21 (all numbers in this paragraph are in decimal).

– Effects of Packet Corruption

What are the consequences of transmission errors in the various fields? If the SOH is garbled, the packet will be treated as interpacket garbage, and lost. If any other character within the packet is garbled into SOH, the current packet will be discarded, and a new (spurious) packet detected. If the length is garbled into a smaller number, then a character from the data field will be misinterpreted as the checksum; if larger, then the program will probably become stuck trying to input characters that will not be sent until one side or the other times out and retransmits. If the sequence number, type, any of the data characters, or the checksum itself is garbled, the checksum should be wrong. If characters are lost, there will most likely be a timeout. If noise characters are spontaneously generated, they will be ignored if they are between packets, or will cause the wrong character to be interpreted as the checksum if they come during packet transmission.

Most kinds of errors are caught by the checksum comparison, and are handled by immediate retransmission. Timeouts are more costly because the line sits idle for the timeout period. The packet design minimizes the necessity for timeouts due to packet corruption: the only fields that can be corrupted to cause a timeout are the SOH and the packet length, and the latter only half the time. Lost characters, however, can produce the same effect (as they would with a fixed-length block protocol). Had a distinguished end-of-packet character been used rather than a length field, then there would be a timeout every time it was corrupted. It is always better to retransmit immediately than to time out.

• Protocol

The KERMIT protocol can be described as a set of *states* and a set of *transitions* that define, for a given event, what action to take and what new state to change to. The inherent simplicity of the design, particularly the requirement that each packet must be acknowledged, reduces the number of states and actions, and the amount of state information that must be maintained, to a minimum.

Here is a simplified version of a state diagram for KERMIT receiving a file:

```
                          +--------------+
                          | Receive Init |
                          +--------------+
                                  |
                                  | (Get Send Init Packet)
                                  |
         (Get EOT Packet)         V
                          +--------------+
(Done) <------------------| Receive File |<---------------+
                          +--------------+                |
                                  |                       |
                                  | (Get File             | (Get EOF Packet)
                                  |  Header Packet)       |
                                  V                       |
                          +--------------+                |
                  +------>| Receive Data |----------------+
                  |       +--------------+
                  |               |
                  |               | (Get Data Packet)
                  |               |
                  +---------------+
```

For simplicity, some transitions are not shown in the diagram:

- If in any state a bad packet is received or a timeout occurs, a null transition back to the same state occurs, with a NAK for the expected packet.
- In any state an error may occur that can cause the transfer to terminate. For instance, the target disk might fill up. The side that encountered the error sends an error packet, containing an informative error message, and quits. Upon receipt of the error packet, the other side displays the message on the screen (if it is in control of the screen) and also quits.
- Actions that are taken on each transition, such as opening a file when a File Header packet is received, are not shown; in particular each packet successfully received is ACK'd.

The receiver starts out in Receive Init state and waits for the other side to send a Send-Init packet. If any other kind of packet is received, or the Send-Init does not arrive within the timeout interval, a NAK is sent. Timeouts or NAKs can occur up to a threshold which, when exceeded for a particular packet, causes the protocol to assume that the connection has become unusable, and to give up. After the Send-Init arrives, the state becomes Receive File; KERMIT waits for a File Header packet containing the name of the file which is to come. When the file header arrives, KERMIT opens a new file using the name provided (perhaps transformed to suit local naming conventions, or to avoid a name collision), and switches to Receive Data state. KERMIT then receives the contents of the file, until an EOF (End Of File) packet arrives. At that point KERMIT switches back to Receive File state. If another file is to be sent, another File Header packet will follow, otherwise an EOT (End Of Transmission) packet will terminate the transfer. The distinction between EOF and EOT, plus the File Header itself, allows files to be sent in groups. EOF marks the end of a file, EOT marks the end of a group. This distinction also allows the two sides to disconnect cleanly: the EOF must be ACK'd before the sender will believe the file has been transmitted correctly; the EOT will follow, but if the ACK which is sent in response is lost, no harm will be done since both sides are terminating anyway.

The state transitions for a sending KERMIT are similar. In each state, instead of waiting for particular packet types, KERMIT sends the appropriate packet and waits for an ACK. If the ACK does not arrive within the allotted time, or a NAK appears instead of an ACK, the same packet is retransmitted. A send operation begins with a Send-Init packet, includes one or more files, each starting with a File Header, followed by one or more data packets, followed by EOF. When all the specified files have been sent, an EOT packet closes the connection and terminates the operation.

```
                            +-----------+
                            | Send Init |
                            +-----------+
                                  |
                                  | (Get Good ACK)
                                  |
  (No more files to send)         V
 +----------+               +-----------+
 | Send EOT |<--------------| Send File |<---------------+
 +----------+               +-----------+                |
                                  |                      |
                                  | (Good ACK)           | (EOF)
                                  |                      |
                                  V                      |
                            +-----------+                |
                   +------->| Send Data |----------------+
                   |        +-----------+
                   |              |
                   |              | (Good ACK)
                   |              |
                   +--------------+
```

Base-level KERMIT provides that during any particular transaction, the sender is the ‘‘master’’ and the receiver is the ‘‘slave’’. These roles may be reversed in the next transaction; any KERMIT implementation is capable of acting as either master or slave. In addition, mainframe implementations may also be put in a kind of permanent slave, or ‘‘server’’, mode in which all commands come in command packets from the master, or ‘‘user’’ KERMIT.

– Initial Connection

To allow a diverse group of computers to communicate with one another, an exchange takes places during initial connection in which the two sides ‘‘configure’’ each other. The sending KERMIT includes setup parameters in its Send-Init packet and the receiving KERMIT responds with an ACK packet containing the corresponding parameters as they apply to itself. The Data field of the Send-Init packet looks like this:

```
  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 *char* function (ASCII value + 32_{10}) unless indicated otherwise.

MAXL The maximum length packet I want to receive, a number up to 94_{10}.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.

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.

NPAD The number of padding characters I want to precede each incoming packet; you respond in kind. Padding may be necessary for a half duplex system that requires some time to change the direction of transmission.

PADC The control character I need for padding, if any, XOR'd with 100 octal to make it printable. You respond in kind. Normally NUL (ASCII 0), some systems use DEL (ASCII 177). This field is ignored if the value NPAD is zero.

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. (Can you see the Catch-22 here?)

QCTL The printable ASCII character I will use to quote control characters and prefix characters, normally ''#''. You respond with the one you will use.

QBIN The printable ASCII character I want to use to quote characters which have the 8th bit set, for transmitting binary files when one or both systems cannot use the parity bit for data. Since this kind of quoting increases both processor and transmission overhead, it is normally to be avoided.

CHKT Check Type, the method for detecting errors. ''1'' for single-character checksum (the normal method), ''2'' for two-character checksum, ''3'' for three-character CRC-CCITT. If your response agrees, the designated method will be used; otherwise the single-character checksum will be used. Other check types may also be added.

REPT The prefix character I will use to indicate a repeated character. This can be any printable character other than blank (which denotes no repeat count prefix), but ''~'' is recommended. If you don't respond identically, repeat counts will not be done. Groups of 4 or more identical characters may be transmitted more efficiently using a repeat count, though an individual implementation may wish to set a higher threshhold.

CAPAS An extendable bit mask encoded printably, to indicate whether certain advanced capabilities, such as file-attribute packets, are supported.

Reserved Fields
The next four fields are reserved for future use. Sites wishing to add their own parameters to the initial connection exchange should start at the fifth field after the capability mask in order to remain compatible with other KERMIT programs.

Naturally, the three prefix characters must be distinct and should be chosen to be uncommonly used printable characters, to minimize further overhead from having to prefix them when they are found in the data.

Trailing fields within the DATA field may be omitted, in which case they will assume appropriate defaults. Defaults for intermediate fields can be elected by setting those fields to blank. Every parameter has an appropriate default, and in fact the entire data field of the Send Init packet or its ACK may be left empty to accept all defaults. The more exotic paramaters are at the end, and reflect more recent developments in the KERMIT protocol; earlier implementations can still communicate with newer ones, since there will not be agreement to use these options. The Send-Init mechanism preserves compatibility from the very earliest KERMIT to the very newest.

There is no protracted negotiation; everything must be settled in a single exchange. Some parameters, however, are outside the scope of this exchange and must be set even before the very first packet is sent. For instance, if the receiving computer can only read characters with odd parity but the sending computer sends them with even parity, the Send-Init packet will never arrive successfully. In cases like this, the user may have to issue some preliminary commands to inform one or both KERMITs about the vagaries of the other system. Another example is the packet terminator (EOL) mentioned above—if the receiving KERMIT requires one that the sending KERMIT doesn't know about, the Send-Init will never get through.

For these reasons, most implementations of KERMIT provide SET commands for all the parameters listed above, and some others as well.

– Rules and Heuristics

During a file transfer, one KERMIT sends information packets—file headers, data, and so forth, and the other KERMIT sends only ACKs or NAKs in response. The most important rule in the KERMIT protocol is

1. *''Wait for a response before sending the next packet.''*

This prevents buffer overruns, and allows participation by half duplex systems. Of course, KERMIT should not wait forever; a timeout should occur after a few seconds if the expected packet has not arrived. Upon timeout, a sending KERMIT retransmits the current packet; a receiving KERMIT re-ACKs the current packet or NAKs the expected one.

Some interesting heuristics are used in the KERMIT protocol to boost efficiency and improve error recovery. A number of important rules take care of the cases when packets are lost in transmission. The first can be stated as

2. *''A NAK for the next packet implies an ACK for the current packet.''*

A NAK with packet number $n+1$ means the receiving KERMIT got packet n, sent an ACK that was never received, and not knowing that the ACK didn't get through (since we don't ACK an ACK), is now waiting for packet $n+1$, which was never sent. In this case, we simply send packet $n+1$. An important exception is when the missing ACK is for a Send-Init packet; do you see why? The next rule,

3. *''ACK and discard redundant packets.''*

handles the situation where the same packet arrives again. The sending KERMIT timed out waiting for an ACK which was sent, but lost, and retransmitted the packet. The receiver must discard the redundant data and ACK the packet again; to do otherwise could have undesirable effects, like adding the same data to a file twice or opening the same file multiple times. Note that the situation resulting from a lost ACK depends upon which side times out first.

KERMIT must handle another situation arising from possible lost packets:

4. *''NAK the expected command.''*

The potential problem occurs in either the Receive Init state or when a KERMIT server is waiting for a command. In either case KERMIT won't know whether communication has begun if the other side's initial packet was lost. KERMIT can't assume the other side will time out and retransmit, so it must check periodically by sending a NAK for packet zero. If KERMIT gets no response it assumes nothing has happened yet and goes back to sleep for a while.

But sending periodic NAKs opens the door to the *buffering problem*. Some systems buffer input for a device; when a program isn't looking for input some or all the input is saved by the computer for future requests. This can cause problems when talking to a KERMIT server or sending a file to a KERMIT in receive wait. If some time has elapsed since activating the remote KERMIT and escaping back and starting up the local KERMIT, a number of NAKs may have accumulated in the local KERMIT's input buffer, so:

5. *''Clear the input buffer at the beginning of a transfer.''*

If the input buffer is not cleared, the local KERMIT will think the remote side is having trouble receiving the first packet. In an effort to get the packet through, it will be sent again; this repeats for every NAK waiting in the input buffer. By the time the first ACK is finally encountered in the buffer, a number of duplicates of the first packet will have been sent out. If this number exceeds the NAK threshhold, the connection will be broken. If not, however, the second packet will be retransmitted once for each of the extra ACKs the remote KERMIT correctly sent for the duplicate first packets. This can continue throughout the file transfer, causing each packet to be sent many times. So, in addition,

6. *''Clear the input buffer after reading each packet.''*

Any computer that buffers its input should clear its input buffer before the transfer and after each packet that arrives successfully. But since not all systems provide a clear-buffer function, we may add another rule:

7. *''Discard redundant ACKs.''*

In the situation just described, the first packet would be sent out multiple times, once for each buffered NAK. Upon receipt of the first ACK, the second packet would go out, but the next response would be another ACK for the first packet; by rule **7**, KERMIT would simply take in the redundant ACK, ignore it, and look for the next ACK, until it got the desired one, in violation of the spirit of Rule **1**.

If we allowed ourselves to violate Rule **1**, we could add a final rule, *''An ACK for packet n also ACKs all previous packets,''* as is done in network protocols like DDCMP [16], allowing data packets to be sent in a continuous stream. KERMIT cannot use this rule for many reasons: sequence number wraparound, buffer overflows, locking out half duplex systems (how can they NAK a bad packet if they can't get control of the line?). Thus if we violate rule **1**, it must be only in a very minor way.

– Example

Here is a sequence of packets from a real file transfer. Each packet starts with Control-A, shown as `^A`.

```
^A) SH( @-#^                                    Send Init
^A) YH( @-#%                                    ACK for Send Init
^A+!FMOON.DOC2                                  File Header
^A#!Y?                                          ACK for File Header
^AE"D   No celestial body has required J        First Data packet
^A#"Y@                                          ACK for first Data packet
^AE#Das m%%%uch labor for the study of its#     Second Data packet, Bad
^A##N8                                          NAK for second Data packet
^AE#Das much labor for the study of its#        Second Data packet again
^A##YA                                          ACK for second Data packet
^AE$D#M#Jmotion as the moon.  Since ClaA        etc...
^A#$YB

(many packets omitted here)

^AD"Dout 300 terms are sufficient.#M#JU         Last Data packet
^A#"Y@                                          ACK for last Data
^A##ZB                                          EOF
^A##YA                                          ACK for EOF
^A#$B+                                          EOT
^A#$YB                                          ACK for EOT
```

In the first packet, we see following the control-A the packet length ‘‘`)`’’ (41_{10}, less 32, or 9), followed by the packet type, S (Send-Init), followed by the appropriate parameter declarations: maximum packet length is H (72-32=40), timeout is ‘‘`(`’’ (40-32=8), number of padding characters is 0 (space=32-32=0), the padding character is 0, end-of-line is ‘‘`-`’’ (45-32=13, the ASCII value of carriage return), the control-quote character is ‘‘#’’, and the remaining fields are omitted, defaulting to appropriate values. The final character ‘‘`^`’’ is the single-character checksum, computed as follows (all numbers and computations in octal, and ‘‘`sp`’’ represents a space):

```
)    sp   S     H     (     sp   @     -    #
51 + 40 + 123 + 110 + 50 + 40 + 100 + 55 + 43 = 674

674 + (674/300) = 676

676 AND 77 = 76;

char(76) = 76+40 = 136 = "^"
```

The receiver ACKs with its own parameters, which are the same. Then comes the file header, the file, EOF, and EOT. One data packet was corrupted by a burst of ‘‘`%`’’ characters, NAK’d, and retransmitted.

• Performance

For text files (documents or program source), assuming an average line length of 40 with lines separated by a carriage-return/linefeed pair, the only control characters normally found in the text file, we see about 5% overhead for prefixing of control characters. Assuming no line terminators for packets (although one or both sides may require them), no retransmissions or timeouts, and no time wasted for the line to turn around between packet and response, then for average packet length *p*, using a single-character checksum, the KERMIT protocol overhead consists of:

5 control field characters in the data packet
5 characters in the acknowledgement packet

+ $0.05p$ for control character quoting

This gives $10/p + 0.05$ overhead. E.g. if the packet length is 40, there is 30% overhead. If p is 96 (the maximum), there is about 15%. These figures will vary with the average line length and the frequency of other control characters (like tabs and formfeeds) in the file, and will go up with immediate retransmissions, and *way* up with delayed retransmissions. For binary files, the quoting overhead will be higher. But transmission overhead can also go down dramatically if prefix encoding is used for repeated characters, depending on the nature of the data (binary data containing many zeroes, highly indented or columnar data or program text will tend to benefit). Each file transfer also gets a fixed overhead for the preliminary (Send Init, File Header) and terminating (EOF, EOT) packets.

If the mainframe end of a connection is heavily loaded, it may take considerable time to digest and process incoming characters before replying. On half duplex mainframes, there may be a pause between sending and receiving, even if the load is light; this might be used to advantage by preparing the next packet in advance while waiting for the current ACK. Another problem may occur on heavily loaded mainframes—undesirable timeouts. Timeouts are intended to detect lost packets. A heavily loaded system may take longer than the timeout interval to send a packet. For this reason, mainframe KERMITs should take the requested timeout interval only as a minimum, and should adjust it for each packet based on the current system load, up to a reasonable maximum.

On a noisy line, there is a greater likelihood of corrupted packets and therefore of retransmission overhead. Performance on noisy lines can be improved by reducing the packet length, and thus the probability that any particular packet will be corrupted, and the amount of time required to retransmit a corrupted packet. A KERMIT program can unilaterally adjust the packet length according to the number of retransmissions that are occurring. Short packets cut down on retransmission overhead, long packets cut down on character overhead.

• **‘‘User Interface’’**

KERMIT was designed from a mainframe perspective. Like many mainframe programs, KERMIT issues a prompt, the user types a command, KERMIT executes the command and issues another prompt, and so on until the user exits from the program. Much care is devoted to the command parser, even on microcomputer versions. The goal is to provide English-like commands composed of sequences of keywords or operands, with abbreviations possible for any keyword in any field down to the minimum unique length, and with ‘‘?’’ help available at any point in a command. Not all implementations need follow this model, but most do.

The basic commands are SEND and RECEIVE. These allow most KERMITs to exchange files. Operands can be the name of a single file, or a file group designator (e.g. with ‘‘wildcards’’) to transmit multiple files in a single operation. Although some systems may not provide wildcard file processing, the KERMIT protocol allows it.

The CONNECT command provides the mechanism for logging in and typing commands at the remote host, which is necessary in order to start the KERMIT on that side. The CONNECT facility provides character-at-a-time transmission, parity selection, remote or local eching, and the ability to send any character, including the ‘‘escape character’’ that must be used to get back to the local KERMIT. However, there is no error detection or correction during CONNECT, just as there normally is none between an ordinary terminal and a host.

When two systems are dissimilar, a SET command is provided to allow them to accommodate each other’s peculiarities, for instance SET PARITY ODD to add odd parity to all outbound characters, or SET LOCAL-ECHO to do local echoing when connected as a terminal to a half duplex system. The SET command must sometimes be used to supply information to the target system on how to store an incoming file with respect to block size, byte size, record format, record length.

Most KERMIT implementations take special care to reassure the user during file transfer. The names of the files being transferred are shown, and a dynamic display is made of the packet traffic, showing successful transmission of packets as well as timeouts and retransmissions. Messages are issued when the user connects to the remote system or escapes back from it, and KERMIT prompts identify the implementation. Helpful error messages are displayed when necessary; these may emanate from either the local or the remote system. The final disposition of the transfer is clearly stated, complete or failed.

The actions required of the KERMIT user depend upon the degree to which the KERMIT programs involved have implemented the specification. Minimal implementations require that the user connect to the remote host, start KERMIT there, issue a SEND (or RECEIVE) command, escape back to the local machine, and issue the complementary RECEIVE (or SEND) command. All this must be done for each transfer. More advanced implementations allow the remote side to run as a ‘‘server’’ and to take all its instructions in special command packets from the local KERMIT; all that is required of the user on the remote end is to connect initially in order to start the server. The server will even log itself out upon command from the local KERMIT. A minimal server can process commands to send files, receive files, and shut itself down.

Here is an example of a session in which the user of an IBM PC gets files from a DECSYSTEM-20. The actions shown are required for minimal KERMIT implementations. The parts the user types are underlined, comments begin with ‘‘!’’ or appear in italics. Everything else is system typeout.

```
A>kermit                ! Run Kermit on the PC.
Kermit V1.20

Kermit-86>              ! This is the Kermit prompt for the PC.
Kermit-86>connect       ! Connect to the DEC-20.
[Connecting to host. Type CTRL-]C to return to PC.]

                        ! You are now connected to the DEC-20.
Columbia University CU20B                      ! The system prints its herald.
@terminal vt52          ! Set your terminal type (optional).
@login my-id password   ! Login using normal login method.

(The DEC-20 prints various messages.)

@kermit                 ! Run Kermit on the DEC-20.
Kermit-20>              ! This is Kermit-20’s prompt.
Kermit-20>send *.for    ! Send all FORTRAN files.
^]c                     ! Type the escape sequence to return to the PC.
[Back at PC.]
Kermit-86>receive       ! Tell the PC files are coming.

(The progress of the transfer is shown continuously on the screen.)

Transfer Complete.
Kermit-86>connect       ! Get back to the DEC-20.
[Connecting to host. Type CTRL-]C to return to PC.]
Kermit-20>exit          ! Get out of Kermit-20.
@logout                 ! Logout from the DEC-20.

Logged out Job 55, User MY-ID, Account MY-ACCOUNT, TTY 146,
  at 24-Apr-83 15:18:56,  Used 0:00:17 in 0:21:55

^]c                     ! Now "escape" back to the PC,
[Back at PC.]
Kermit-86>exit          ! and exit from the PC’s Kermit.
```

The session is somewhat simpler when the remote KERMIT is being run as a server. The user must still CONNECT, log in, and start KERMIT on the remote end, but need never again CONNECT to issue subsequent SEND, RECEIVE, EXIT, or LOGOUT commands, even though many transactions may take place. All actions can be initiated from the PC.

• Advanced Features

An optional feature of the KERMIT protocol is a special packet designed to express the attributes of a file in a compact and generic manner. The receiver may either attempt to use the attributes to the incoming file, or archive them for later use. Attributes include not only file characteristics but also the intended disposition—store, print, submit for batch processing, send as mail, etc. Other optional features include mechanisms for gracefully interrupting, delaying, or suspending operations in progress; alternate forms of packet data encoding; filename conversion, local file management, raw data transmission and capture; command macro definition, etc.

Although KERMIT was never intended to fulfill the role of a general purpose network server, its design has made it simple to add new functions. A KERMIT server has the ability to accept commands in packets from a remote KERMIT. The basic commands are for sending or fetching files and for shutting down the server. Other commands may require display of text at the user's terminal, which is controlled by the local KERMIT. For example, a directory listing could be requested; the resulting text is sent back to the local KERMIT exactly as if a file were being transferred, except the destination is the user's screen, rather than a disk file. (Or it could be disk file too.) With this ability in place, it is possible to implement all sorts of commands, for instance to delete a file, show who's logged in, inquire about disk space, verify access to a directory, submit batch jobs, send messages, and so forth.

The ability of the KERMIT server to perform host functions can be added very simply under certain operating systems. For instance, under UNIX [17], KERMIT can ‘‘fork’’ a *shell* with commands to perform any function possible under UNIX, redirecting the standard output through a process (KERMIT itself) that encapsulates it into KERMIT packets and sends it along.

A server with these capabilities could provide convenient access to a timesharing system by users at personal workstations, without requiring the users to be directly involved with the host. If, for instance, workstations had dedicated connections to a host, and the host had dedicated KERMIT servers for each such line, users could get access to and manage their host files completely by commands typed at the workstation. Taking this idea one step further, the workstation system software could be modified to make the whole process transparent by incorporating a KERMIT-like protocol in its file access logic—fetching and updating host files as necessary behind the user's back. Since the demands placed on a host by KERMIT are relatively modest, many more simultaneous users could probably be serviced in this way. This approach could be a relatively painless entree into the distributed, networked environment of tomorrow. When local area network protocols become mature and the hardware economical and widespread, KERMIT can be replaced by ‘‘the real thing’’. But for the ordinary computer user for whom dedicated connections are impractical, ‘‘do-it-yourself’’ KERMIT, or some facility like it, will be a valuable tool for years to come.

• Conclusion

The need for a cheap, convenient file transfer capability among diverse systems is pressing, and there are certainly many efforts similar to ours under way at many places. We hope that this article may contribute to those efforts; we don't claim to have the last word on any of the issues raised here, and expect that this article may flush some other approaches out of the woodwork. We have billed KERMIT as a ‘‘simple’’ protocol; anyone who has read this far will begin to appreciate what must go into the more complicated protocols used in real networks, or when ‘‘integration’’ of micro and mainframe is a major goal—demand paging of remote files, remote database queries, distributed editing and computation.

Meanwhile, the KERMIT protocol has been proven successful, and continues to grow in popularity. As of this writing, implementations exist for more than 50 computer systems. Some of the major ones include:

Machine	Operating System	Language
DECsystem-10	TOPS-10	MACRO-10
DECSYSTEM-20	TOPS-20	MACRO-20
IBM 370 Series	VM/CMS	IBM Assembler

VAX-11	VMS	Bliss-32
VAX, SUN, PDP-11, etc	UNIX	C
PDP-11	RT-11, RSX, RSTS	MACRO-11
8080, 8085, or Z80	CP/M	8080 ASM
8086, 8088	PC DOS, MS DOS	IBM PC Assembler
Apple II 6502	Apple DOS	DEC-10/20 CROSS

Some of these have been contributed or enhanced by the institutions listed in the acknowledgements, below. No single implementation necessarily includes all the features mentioned in this article, but all are able to communicate at least at base level. Additional implementations are in preparation, and present ones are being enhanced.

Columbia University is willing to provide all KERMIT programs, sources, manuals, and other documentation to computing centers, academic or corporate, in return for a modest fee to cover costs for media, printing, postage, labor, and computing resources. Only magnetic tape and listings can be shipped. We cannot produce floppy disks; instructions are included for bootstrapping the microcomputer implementations from the mainframe computers. Details will be provided on request; write to:

KERMIT Distribution
Columbia University Center for Computing Activities
7th Floor, Watson Laboratory
612 West 115th Street
New York, NY 10025

The protocol specification supplemented by examples of existing KERMIT implementations allows new implementations to be created with relative ease. In the past, KERMIT implementors have shared their work with other KERMIT users by contributing it to the Columbia KERMIT library. We hope that this practice will continue until KERMIT has spread throughout the known world.

• Acknowledgements

In designing the initial KERMIT protocol, we studied several models, primarily the ANSI recommendation [9]. Others include the Stanford University DIALNET project, the University of Utah ‘‘Small FTP’’ project, and the Stanford University Medical Center TTYFTP project. And we examined some real networks, like ARPANET and DECnet.

Acknowledgements also to the many sites that have contributed new KERMIT implementations or enhanced old ones: Stevens Institute of Technology, Digital Equipment Corporation, the National Institutes of Health, Cornell University, the University of Toronto, the University of Tennessee, the University of Toledo, Cerritos College, and many others. Thanks to Dr. Howard Eskin for help with this article.

• The ASCII Alphabet

Dec	Oct	Character	Dec	Oct	
000	000	NUL (^@)	064	100	@
001	001	SOH (^A)	065	101	A
002	002	STX (^B)	066	102	B
003	003	ETX (^C)	067	103	C
004	004	EOT (^D)	068	104	D
005	005	ENQ (^E)	069	105	E
006	006	ACK (^F)	070	106	F
007	007	BEL (^G)	071	107	G
008	010	BS (^H)	072	110	H
009	011	HT (^I)	073	111	I
010	012	LF (^J)	074	112	J
011	013	VT (^K)	075	113	K
012	014	FF (^L)	076	114	L
013	015	CR (^M)	077	115	M
014	016	SO (^N)	078	116	N
015	017	SI (^O)	079	117	O
016	020	DLE (^P)	080	120	P
017	021	DC1 (^Q)	081	121	Q
018	022	DC2 (^R)	082	122	R
019	023	DC3 (^S)	083	123	S
020	024	DC4 (^T)	084	124	T
021	025	NAK (^U)	085	125	U
022	026	SYN (^V)	086	126	V
023	027	ETB (^W)	087	127	W
024	030	CAN (^X)	088	130	X
025	031	EM (^Y)	089	131	Y
026	032	SUB (^Z)	090	132	Z
027	033	ESC (^[)	091	133	[
028	034	FS (^\)	092	134	\
029	035	GS (^])	093	135	]
030	036	RS (^^)	094	136	^
031	037	US (^_)	095	137	_
032	040	(SP)	096	140	‘
033	041	!	097	141	a
034	042	"	098	142	b
035	043	#	099	143	c
036	044	$	100	144	d
037	045	%	101	145	e
038	046	&	102	146	f
039	047	’	103	147	g
040	050	(	104	150	h
041	051	)	105	151	i
042	052	*	106	152	j
043	053	+	107	153	k
044	054	,	108	154	l
045	055	-	109	155	m
046	056	.	110	156	n
047	057	/	111	157	o
048	060	0	112	160	p
049	061	1	113	161	q
050	062	2	114	162	r
051	063	3	115	163	s
052	064	4	116	164	t
053	065	5	117	165	u
054	066	6	118	166	v
055	067	7	119	167	w

```
056 070 8           120 170 x
057 071 9           121 171 y
058 072 :           122 172 z
059 073 ;           123 173 {
060 074 <           124 174 |
061 075 =           125 175 }
062 076 >           126 176 ~
063 077 ?           127 177 DEL
```

• References

The following publications provided useful guidance or diversion in the development of KERMIT.

[1] *EIA Standard RS-232-C*
Electronic Industries Association, 2001 Eye Street N.W., Washington DC 20006, 1969, 1981.

[2] *ANSI X3.4-1977, Code for Information Interchange*
American National Standards Institute, 1430 Broadway, NYC 10018, 1977.

[3] *ANSI X3.15-1976,*
Bit Sequencing of ASCII in Serial-By-Bit Data Transmission
1976.

[4] *ANSI 3.16-1976, Character Structure and Character Parity Sense for Serial-By-Bit Data Communication in ASCII*
1976.

[5] McNamara, J.E.
Technical Aspects of Data Communication.
Digital Press, 1982.

[6] Mackenzie, C.E.
Coded-Character Sets: History and Development.
Addison-Wesley, 1980.

[7] *DNA Data Access Protocol (DAP) Functional Specification*
Digital Equipment Corporation, 1980.
AA-K177A-TK.

[8] Neigus, N.J.
File Transfer Protocol for the ARPA Network
Bolt Beranek and Newman, Inc., 1973.
RFC 542, NIC 17759. Available in the ARPANET Protocol Handbook, NTIS AD/A-052 594.

[9] *ANSI X3.28-1976, Procedures for the Use of Control Characters of ASCII in Specified Data Communications Links*
1976.

[10] Pierce, J.R., and Posner, E.C.
Applications of Communications Theory: *Introduction to Communication Science and Systems.*
Plenum Press, 1980.

[11] Knuth, D.E.
The Art of Computer Programming. Volume I: *Fundamental Algorithms.*
Addison-Wesley, 1973.

[12] Hamming, R.W.
Error Detecting and Error Correcting Codes.
Bell System Technical Journal 29:147-160, April, 1950.

[13] Martin, James.
Teleprocessing Network Organization.
Prentice-Hall, 1970.

[14] Perez, Aram.
Byte-wise CRC Calculations.
IEEE MICRO 3(3):40-50, June, 1983.

[15] *ISO Reference Model for Open Systems Interconnection (OSI)*
International Organization for Standardization (ISO), 1982.
Draft Proposal 7498.

[16] *DNA Digital Data Communications Message Protocol (DDCMP) Functional Specification*
Digital Equipment Corporation, 1980.
AA-K175A-TK.

[17] Thomas, R. and Yates, J.
A User Guide to the UNIX System.
OSBORNE/McGraw-Hill, 1982.