18. Murray, W. D.; Moss, C.E.; Parr, W.H. and Cox, C., A radiation and industrial hygiene survey of video display terminal operations, Human Factors, 1981, 23 (4), 413-420.

19. Quick, M. J., 1982, "Information Systems Requirements for Development Engineering Productivity," Task Force Report, Internal IBM Report, Kingston, New York. Cited in Minicucci, R. A., sub-second response time, a way to improve interactive user productivity, IBM STL, 28, 1982.

20. Rosenthal, S.G., and Ghundy, J.W., Avoiding eye problems with VDU's, Physics Technology, 1980, II 175-186.

21. Rouse, W. B., 1975, Design of man-computer interfaces for on-line interactive systems. Proceedings of the IEEE, 63 (6), 847-857.

22. Shneiderman, B., 1979, Human factors experiments in designing interactive systems, Computer, 12, 9-19.

23. Smith, M. J., Cohen, B. G., F. Stammerjohn, L.W. Jr., and Happ, A., An investigation of health complaints and job stress in video display operations. Human Factors, 1981, 23, 387-400.

24. Smith, W.A., Jr., 1967, Data Collection - Systems - Part 1: Characteristics of Errors. Journal of Industrial Engineering, 18 (12), 703-707.

25. Stewart, T.F.M., Displays and the software interface, Ergonomics, 1976, 7 (3), 137-146.

26. Thadhani, A.J., 1981, "Interactive User Productivity," IBM Systems Journal, 20 (4), 407-423.

27. Tomeski, E.A., 1975, Building human factors into computer applications: computer profession must overcome a "jackass fallacy"! Management Datamatics, 4 (4), 115-120.

28. Treu, S., 1975, Interactive command language design based on required mental work. International Journal of Man-Machine Studies, 7, 135-149.

29. Welyczkowsky, G. D., 1982, "A Study in Interactive User Productivity: The Effect of Remote Transmission Delay on TSO Productivity," Internal IBM Report, Poughkeepsie, New York. Cited in Minicucci, R. A., sub-second response time, a way to improve interactive user productivity, IBM STL, 28, 1982.

SESSION REPORT

SHARE

SHARE NO.	SESSION NO.	SESSION TITLE	ATTENDANCE
61	M973	File Transfer Protocols Used at Universities	

PROJECT	SESSION CHAIRMAN	INST. CODE
University Information Exchange Project	Sandra Ward	WAT

Dept. of Computing Services, Univ of Waterloo, Waterloo, Ontario N2L 3G1, 519-885-1211

SESSION CHAIRMAN'S COMPANY, ADDRESS, and PHONE NUMBER

1. KERMIT - The File Transfer Protocol Used at Columbia

   Daphne Tzoar

   Columbia University
   Center for Computing Activities
   612 West 115th Street
   New York, NY 10025

   Installation Code: BWY

2. YTERM/PCTRANS FILE TRANSFER PROTOCOL

   Josh Auerbach

   Computing Center
   Yale University
   New Haven, Connecticut

   Installation Code: YU

THE KERMIT FILE TRANSFER PROTOCOL

Everyone wants to get different computers talking to each other. At Columbia, we have been doing this with great success. File transfer is performed with a protocol we call "Kermit". Kermit is also the name of a family of programs that use the protocol to provide error-free file transfer between various types of computers -- word processors, microcomputers, minicomputers and large mainframes. We connect the two systems over TTY lines thus tricking each computer into thinking the other is a terminal.

Kermit was originally developed at Columbia two years ago for students with limited on-line storage allocations. Kermit enabled them to keep their work on floppies. We provided Intertec SuperBrains and the means by which they could transfer files to and from the micro and our DEC-20. Since then and the proliferation of many different micros, and due to the needs of our users in the university community, we expanded the protocol and implemented Kermit for numerous mainframes and micros, with contributions from several sites other than Columbia, most notably Stevens Institute of Technology. To date, Kermit has been shipped to over 250 sites in the US, Canada, and overseas -- it has been well tested to be sure !! It is used by many educational institutions, companies, research laboratories and several dial-up databases.

You might ask why do we need a protocol? There are two major reasons why a protocol is necessary: (1) Noise - data can become garbled in transmission. The longer the wire, and the faster the baud rate of the line, the more the data is prone to noise and error. Noise corrupts the data often in subtle ways. Therefore, we use a protocol - we intermingle control information with data to achieve data integrity. (2) Synchronization - One computer may be faster or have larger buffers than the other. If it sends data faster than the other can receive, data will be lost. Built in flow control mechanisms such as XON/XOFF cannot be guaranteed to work, because different computers may not honor the same conventions. The protocol therefore uses packetizing and checksumming techniques. By packetizing I mean that Kermit sends data in numbered chunks or packets. The packet size varies between 20 and 96 characters. We use relatively short packets, since they are more likely to arrive in one piece than are long ones. Sequence numbers are used to detect missing or extra packets. A checksum is used to allow detection of corrupted packets.

Currently, there are two versions of the protocol. In Version 1, the basic version, all commands to the remote host (like "send") are typed in by the user who must then return to the local system and issue the complementary command (like "receive"). In Version 2, the user starts a server on the remote system and returns to the local host. From that point on, all commands to the remote host are from the local Kermit. (Example - send me the file ABC DEF). Version 2 is upward compatible with version 1.

Design goals were that the protocol be simple enough to be implemented easily and general enough to run on a wide variety of micros and mainframes requiring only the most common hardware features. The protocol then is portable, not necessarily the program. It is important to realize that any Kermit can communicate with any other once a connection is established. More advanced features, like data compression, are automatically disabled unless both sides agree to use them. Therefore, new versions of Kermit can talk to older ones, perhaps without being able to take advantage of the newer functions.

In light of these goals, Kermit assumes very little about the other host. No embedded sign on procedures or special knowledge of the remote hosts file or command structure are

included so Kermit's use is uniform across any two systems. The protocol makes no assumption about speed, duplex or flow control. It does assume, however, that: (a) all printable ASCII characters are accepted as input to the host and will not be transformed, (b) a single non-printable character can be used for synchronization (generally, Control-A), and (c) if a host requires a line terminator for terminal input, it is a single ASCII character (like CR or LF). As implied, all transmission is done in ASCII. If a system uses anything else (EBCDIC, for example), it is its responsibility to convert all data. So, the CMS version does ASCII/EBCDIC and EBCDIC/ASCII translations.

The protocol is designed for character oriented transmission over serial telecommunication lines. It allows for restrictions and peculiarities of different operating systems (buffering, duplex, parity, and so on.) For example, when receiving a packet from the IBM, it knows to wait for the XON before sending the next packet.

In designing the protocol, we wanted the flexibility to talk to our half-duplex IBM system in addition to the full duplex DEC-20. And we wanted to avoid bombarding our DEC-20 front end with continuous data - it assumes that all incoming data comes from a person typing at the keyboard. It cannot allocate buffers quickly enough to accomodate streams of data coming to it and hence data can be lost. Therefore, we developed Kermit to be not truly full duplex or asynchronous. The protocol does not allow for "stacked" packets and long packets are not sent. For every packet of data that gets sent, Kermit expects a response before the next packet is sent. So while Kermit does not have transfer rates as high as truly asynchronous full-duplex protocols, we usually achieve 50-80% efficiency (that is, user bits / baud rate.)

Kermit is generally used for file transfer between a mainframe and a micro, although host to host and micro to micro file transfer are possible.

We have two types of Kermits - dumb ones and smart ones. The smart Kermits are capable of timing out when waiting to receive data from the serial port. That means, they can detect remote system crashes or protocol deadlocks where part of a packet is lost in transmission. The dumb Kermits do not have the timeout facility and will sit forever waiting for data. One smart partner is enough to prevent such deadlocks. If two dumb Kermits are talking to each other and a deadlock arises, the situation can be handled via manual intervention - typing a carriage return to "wake up" the transmission. Your micro will act as though it received a negative acknowledgement from the host and retransmit the last packet. This problem arises when talking to VM/CMS from a "dumb" micro: you cannot interrupt a read on the VM/CMS "console", so you cannot time out. You must manually intervene on the micro if you suspect a deadlock has occured by typing the carriage return.

The packet is the basic tool used in the file transfer. Much consideration was given to its design - it had to be small enough to avoid buffer overflow problems, yet large enough to be efficient. The fields with the control information had to be informative without taking up too much space in the packet. Because of these restrictions, each control field of information is a single printable byte of data. Since there are 95 printable ASCII characters, we can represent values from 0 to 94 in each field.

Many protocol definitions view the packets as layers of information which pass through a hierarchy of levels, with each adding its own information to the ends of outbound packets and stripping off relevant information from the ends of incoming packets. The remainder is then passed on to the next level. Each level must be able to identify and interpret the pertinent fields. Such is the case with the Kermit protocol.

First some preliminaries before describing the packets. Kermit uses three special functions to transform characters (numbers are in decimal, characters are in ASCII):

> (a) char (x) = x + 32 (32 == space)
> Make the integer "x" printable. "x" is assumed to be between 0 and 94. For example, 1 becomes "!".
>
> (b) unchar(x) = x - 32
> Revert the (printable) character to its original form. For example, "!" becomes 1.
>
> (c) ctl(x) = x XOR 64
> This function maps between control characters and printables, preserving the hi order bit. So, ^A --> A, ^Y --> Y. Note, ctl (ctl (x)) = x.

Frequently used keywords:

> (d) ACK -- Acknowledge receipt of a correct packet
> (e) NAK -- Negatively acknowledge. Packet was received in bad condition (bad checksum, incorrect packet number, and so on)

```
Kermit packet:
<mark> <char(count)> <char (seq)>  <type> <data>  <char (chksum)>
|                    |              |              |               |
|                    |              |_ application _|               |
|                    |                             |               |
|                    |____________ session ________|               |
|                                                                  |
|_________________________________ data link ______________________|
```

The mark character indicates the start of a packet, generally ^A, although it can be redefined. It must be a single control character that can be easily identified as the packet header - one that is not usually found in text files. It must be in the beginning of all packets. All data between packets will be ignored until Kermit finds the SOH character. If the ^A is lost, then the entire packet will be missed. Also, note that messages sent to you will be interpreted as interpacket data and thrown away.

The count is the number of characters in the packet following this field, not including any end-of-line character or padding. This field must contain a single, printable character, thus the maximum value is 94.

Next is the sequence number modulo 64. The sequence number is used to detect duplicate or missing packets.

The packet type is one of the following:

D		data
Y		acknowledge, ACK
N		negative acknowledge, NAK
S		send initiate
B		break transmission
F		file header
Z		end of file, EOF
E		error
R		receive initiate - Ask server to send specified file(s)
G		generic Kermit command - Single character in the data field specifying command. Some examples:
	I	Login
	C	Change working directory
	L	Logout
	F	Finish (shut down server, but don't logout)
	D	Directory
	T	Type
C		host command - Data field contains a string that is to be executed as a command by the host
X		text header - Response to "C" or "G" packet. Is like file transfer but the destination is the screen. As an example, if you send the generic command "directory", the "X" packets would consist of the list of all your files.

The last four packet types are used in conjuction with the server mode of the protocol. Not all have been implemented.

The next field is the data. The contents of the data field varies depending on the state of the transmission. In some cases, the data field is empty. Normally, though, the data is the contents of the file being transferred. The data field includes the quoting of non-printable characters. Control characters are converted to prevent the host or front end from acting upon them accidentally. We do not want data characters interpreted; for example, the carriage return in the data must be distinguished from the carriage return terminating the packet. For control characters and delete, we use the CTL function mentionned earlier and convert ^A to A. The other side de-controllifies the data. To distinguish the real "A" from the controllified "A" we use a quote prefix, typically #. Note, the quote character must also be quoted if used within text and quoted sequences may not be broken across packets. Other prefixes can be used if quoting is requested for eight-bit quantities and repeated quantities. Typically, the quote character for eight bit quantities is "&" and "~" for repeated characters. Some examples:

```
  A -->  A                    ^ denotes a control character
 ^A --> #A                      # is the quote character
 'A --> &A
'^A --> &#A                   ' denotes the hi order bit is on
  # --> ##                      & is the quote character
 '# --> &##
 & --> #&
'& --> &#&
```

This quoting scheme is inefficient for binary files. In fact, it leaves us with 50% overhead. But it is used only when necessary. Binary file transfer is performed without quoting as long as both sides can control parity, as is generally the case between two micros. Only if this is not the case is the prefix character used. Note, this is not a primary concern since the most common type of transfer between unlike computers is printable files.

The final field is a checksum of all data between but not including the SOH and the checksum character, modulo 64. Together with the packet length, it is used to insure data integrity. Since both sides must agree on this value we can detect corrupted, missing, or extra characters. The protocol allows for three different types of checksum. The default is the single character arithmetic sum, which is currently used and has proven adequate. We calculate it as follows: If "S" is the sum of characters, then

```
checksum = [ S + ((S AND CO)/40)] AND 3F   (constants are in HEX).

For example, if the packet consisted of:

          ^A) S~( @-#U

the checksum would be calculated as follows:

   ")" + " " + "S" +  "~" +  "("  +  " " + "@" +  "-"  +  "#"
    29 +  20 +  53 +  7E +   28   +   20 +  40 +   2D  +   23   =  1F2

    1F2 + ((1F2 AND CO) / 40) =
    1F2 + 3 = 1F5
    1F5 AND 3F = 35
    35 + 20 = 55 = "U"            (do the CHAR function)
```

Only 6 bits of the arithmetic sum are used. In order that all data bits in the packet contribute to the checksum, we add the value of bits 6 and 7 to the quantity formed by bits 0 through 5.

Two fancier types of checksum can be used only if both sides agree to do so. They are a two character arithmetic sum and a three character 16-bit CRC. So far, no one has found it necessary to resort to these techniques.

The packet may be accompanied by a line terminator if required by the host. This is necessary for systems that cannot do single character input from the terminal. Without the carriage return, VM/CMS for example, would never "see" the incoming packet. If the system is capable of reading one character at a time, however, the count field can be used to determine the number of characters to read in for the packet. The end of line character is not part of the packet nor is it included in the count or the checksum. If used, it must be a single character and is specified during initialization, by default a carriage return.

It can be seen, then, that the packets do indeed have a layered design as described by the ISO model: the outermost fields are used by the data link layer to verify data integrity (that is, the MARK, the COUNT, and the CHECKSUM), the next by the session layer to verify continuity (the SEQUENCE number), and finally the TYPE and the DATA itself at the highest level, that is the application layer.

File transfer is the goal of the protocol. Kermit is a cooperation between two programs,

one running on the micro and one on the host. It does not involve simply grabbing data as it comes in over the line. One side sends a packet and waits for an acknowledgement or a negative acknowledgement before the next packet is sent. If Kermit gets a NAK, for instance because of a bad checksum, the packet is retransmitted. It is also resent if the packet sequence number indicates a packet was lost somewhere or if Kermit times out waiting for the next packet.

The two sides begin the transfer with the sender (the side where the file exists) sending a send-init packet of the format:

```
bufsiz, timout, npad, padchar, eol, quote, 8-bit quote
chktype, repeat, reserved fields
```

Filling in the fields of the init packet is optional. If data is not supplied, defaults will be used.

The fields are interpreted as follows (where "I" refers to the Kermit sending the init packet, and "you" to the Kermit receiving it):

bufsiz	The maximum packet size I can accept. If none specified, the default value of 96 (decimal) is assumed. The other side should send packets no longer than this length. Shorter lengths can be used to circumvent buffering problems or to get packets across noisy lines with a greater chance of success (and less overhead in recovering from errors.)
timout	The number of seconds after which I want you to time out if no packets have been successfully received provided you are capable of timeouts. The normal value is in the 5-15 second range. A value of 0 means "don't time me out". This value is taken as a guideline rather than an absolute, and may be adjusted on a per-packet basis depending upon system load.
npad	The number of padding characters I need preceding each packet. Some systems may require padding; for instance, some half duplex systems may need some time to "turn the line around". If none specified, or a value of 0 is specified, no padding is done and the contents of the next field is ignored.
padchar	The character I want used for padding, normally NUL (ASCII 0). If npad is non-zero but pad is omitted, NULL will be used as the padding character.
eol	The desired line terminator I need for incoming packets. Only a single control character is permitted in this field. Hosts cannot specify printable terminators or multi-character terminator sequences. If none specified, carriage return is used.
quote	The printable ASCII character I will use when quoting control characters. If none is specified, "#" is used.

The following are newer parameters and not implemented in all versions of Kermit. Specifying values for these parameters is optional since two communicating Kermit's will use the lowest common denominator of parameters. So if a newer Kermit is talking to an older one, the newer partner will use only those features implemented in the older Kermit.

8-Bit-Quote

Specify quoting mechanism for 8-bit quantities. A quoting mechanism is necessary when sending binary files to hosts which prevent use of the 8th bit for data. Most IBM systems fall into this category. When elected, the quoting mechanism will be used by both hosts. The quoting character must be different from the control-quoting character. This field is interpreted as follows:

Y	I agree to 8-bit quoting if you request it
N	I will not do 8-bit quoting
&	(or any other character in the allowed range besides Y and N) I want to do 8-bit quoting using this character. It will be done if you put a Y in this field. The recommended 8th-bit quoting prefix character is "&".

Anything Else
8-bit quoting will not be done.

The default is not to do 8-bit quoting. The quoting is undesirable and the parity bit should be used if possible. Again, I stress that it should not be necessary to do much eight bit quoting since file transfer between unlike systems is mainly for printable data.

chktype

The type of block check. The only values presently allowed in this field are "1", "2", and "3", though future implementations may allow others. The default is one. These values specify the single and double character arithmetic checksums, or the three character CRC; however at this time only the one character checksum is implemented. The sender would request the desired type of checksum in this field; if the receiver replies with the same type in its ACK then the requested type will be used, otherwise the single-character arithmetic checksum must be used. Both sides, of course, must use the same type of checksum.

repeat

The prefix character to be used to indicate a repeated character. This can be any printable character other than blank (which denotes no repeat count prefix), but "~" is recommended. Both sides must agree on the prefix to be used, otherwise repeat counts will not be done. Groups of 4 identical characters or more may be transmitted more efficiently using a repeat count, though an individual implementation may wish to set a higher threshhold. The repeat prefix can produce a big savings in transmission for binary files (which typically contain lots of zeroes), highly indented structured programs, and so on. The default is no repeat count processing.

The Kermit receiving this packet responds with an ACK packet containing its specifications for the init parameters. From then on, the two are 'configured' to talk to each other. If all goes well, the file name, data and end-of-file are sent. This can be followed by another file header if sending more than one file, for instance when the user used a "wildcard" to specify a file group. When all files have been sent, an end-of-transmission packet is sent. Only the filename and the contents of the file are sent. No attributes (such as creation date, record length and so on) are transmitted to the other system. If at any point a NAK is received, Kermit retries sending the current packet up to a certain threshold, like 5 (more times if it is the initialization packet). If the current packet is not

sent successfully within the retry limit, transmission is aborted.

Note that all initializing must be done in one exchange of packets. Parameter values are negotiated, but it is done only once per transfer and in a single step. The send-init packet, however, may not get through if incorrect assumptions are made about the other system. (For example, it may require a different end-of-line character than the default in order to read incoming packets.) In such a situation, the user must issue the SET command to change the default end-of-line character.

Essentially then, the Kermit protocol consists of a set of states and rules indicating how one gets from one state to another.

Many interesting situations can crop up as packets fly back and forth. A few heuristics are necessary to cope with them. Some examples:

- A NAK for the current packet is equivalent to an ACK for the previous packet (except when the previous packet was the send init with transmission parameters). This covers 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.

- 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, because it's already there!

- 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 acting as a server, in which case it has been sending out periodic NAKs. If you don't do this, you may find that there are sufficient awaiting 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 a number of times.

- Similarly, after reading a packet (successfully or not), you should clear the input buffer. There should be nothing there for you anyway, since the other side must normally wait for you to send your packet in response. Failure to clear the buffer could result in propogating the repeated sending of a packet caused by stacked-up NAKs.

- Finally, it should be noted that while a local Kermit is in charge of the screen, the remote Kermit is not. If an error occurs and the remote Kermit tries to print a message to the terminal, it will go to the local Kermit as interpacket data and will be ignored. Therefore, the host should send the text of the message in the form of an error packet to the local Kermit before aborting. This allows the local Kermit to print the error message on the screen and terminate gracefully rather than wait forever for the packet that will never come. An example of such a situation would be the host trying to write to a full or read-only disk.

To give you a better idea of what the packets really look like, let us assume we have a file called SHARE.TXT that contains the following lines:

```
Hi there.
How are you?
I am OK.

Here is what the packets would look like for sending this file
(the semi-colons followed by text are comments):

^A) S~( @-#U                                ; Send-init
^A) Y~( @-#[                                ; ACK with init parameters
^A,!FSHARE.TXTT                              ; File name
^A#!Y?                                       ; ACK
^AO"DHi there.#M#J,                          ; Data line one
^A#"Y@
^A3#DHow are you?#M#J[                       ; Data line two
^A##YA
^A/$DI am OK.#M#J5                           ; Data line three
^A#$YB
^A#%ZD                                       ; End-of-file packet
^A#%YC
^A#&B-                                       ; Break transmission
^A#&YD                                       ; OK
```

Take data line two for example. First is the Control-A synchronization character. Next is the packet size after the CHAR function has been performed. To interpret this field, we must perform the UNCHAR funtion. So, the actual size is ASCII 3 which is (in decimal) 51 − 32 or 19, which you will notice is the exact number of characters following this field. The packet sequence number, after UNCHAR, is ASCII "#" (35) minus 32 = 3. And, note this is the fourth packet in the transfer. Next is the packet type, D for data. Then the text, including the transformation of the control characters carriage return and line feed to #M and #J respectively. Lastly, is the checksum, after the CHAR function. Of course, had any of the ACK's above been a NAK, the previous packet would have been resent.

The file in the example contained actual carriage return-line feeds. If it existed on a system using the concept of records, however, such as CMS, accomodations would have to be made so that the file would be readable on another system. Because of this, Kermit-CMS deletes trailing blanks and adds CRLF to the end of all outgoing records and strips the CRLF from all incoming lines to create records.

I should mention at this point that a separate terminal emulator program is not necessary in order to start the file transfer. Although it is not within the bounds of the protocol, the Kermit program for all micros and some mainframes does terminal emulation which allows you to log in to another host and initiate the transfer.

In conclusion, the protocol has proven to be very successful. We are able to transfer files between unlike computers with ease. In addition, implementing new versions while not trivial, is not an extended project. The protocol, moreover, can be used in areas other than file transfer. Consider the case where workstations have dedicated connections to a host and the host has a dedicated Kermit server on each line. The best system for this would be UNIX since it allows I/O redirection. The user could perform host functions, including accessing his files on the mainframe, from the workstation. He need never be directly involved with the host. This connection could be initiated by the user or done behind his back as part of the micro operating system. As another example, terminals and plotters could implement the protocol in firmware. Then, a user would not have to worry about a noisy line sending random characters to his graph or listing. He would no longer

be concerned with spurious characters being added arbitrarily to data files. Another application is creating a Kermit PROM for a microcomputer. Then, instead of using a disk or tape, you could transfer a file from a foreign host directly to memory and execute your program from the specified memory location. Many more applications may arise in the future.

Finally, I leave you with a list of major versions of Kermit:

```
DEC-10   DEC-20   IBM VM/CMS   UNIX   VAX/VMS   RT-11

CP/M     PC DOS   Apple DOS
```

YTERM/PCTRANS FILE TRANSFER PROTOCOL

This document describes the protocol used by YTERM and PCTRANS to exchange (primarily) data files over asynchronous communication lines. Where appropriate, planned extensions of the protocol are mentioned. The protocol is based on a set of "clean" constructs which, however, are modified in some places to accommodate to "dirty" reality. In particular, the protocol must be capable of dealing with the peculiar environment presented by a Series/1 minicomputer running the Yale ASCII Terminal Communication System.

NOTE: this document may be printed on an EBCDIC device which does not correctly represent the ASCII "backslash" character. If the character in parentheses here (¢) looks like something other than backslash (cent-sign, probably) bear that in mind in reading the document.

Protocol Layers

The protocol has two distinct layers. Each is designed to be complete at its own level and it should be possible to use one without the other. In terms of the popular OSI Reference Model (ISO/DIS 7498), the lower layer deals with issues at the "transport layer" and below. Its goal is the efficient and "apparently error free" transmission of arbitrarily long strings of ASCII characters with the ability to encode all 128 7-bit characters. The higher layer deals with issues above the "transport layer" such as the representation of binary data, the representation of file names and other requests, and the conversational details of file transfer.

THE LINK-TRANSPORT PROTOCOL

Character set

Although the character set can ENCODE all 128 7-bit ASCII characters it does not use them all. The following are the characters it uses.

1. Graphics plus blank (ASCII codes 20-7E hex, 040-176 octal). These characters can be used as data.

2. ASCII Escape (code 1B hex, 33 octal). This character can be used in conjunction with the first set for control purposes.

3. XON (11 hex, 21 octal) and XOFF (13 hex, 23 octal). These characters are used, optionally, where possible, to provide pacing control. All implementations of the protocol should attempt (if possible) to monitor received data during transmission. If an XOFF is received, transmission should be suspended until an XON is received. While receiving

data, an implementation may choose to send an XOFF if its buffer is nearly full. In this case, an XON is used to signal the ability to accept more data.

   In the current YTERM/PCTRANS implementation, YTERM sends XOFF/XON in an attempt to preserve orderly data flow. PCTRANS, when running under Yale ASCII, respects these. However, PCTRANS does not send, nor does YTERM respect, these pacing characters.

4. Carriage return (0D hex, 15 octal) is used in certain circumstances.

   a. YTERM ends every buffer of data with a CR when sending to PCTRANS.

   b. YTERM ends every acknowledgment with a CR while receiving data from PCTRANS if (but only if) it is running in "B" or "H" mode. In "F" mode, YTERM does not chase acknowledgements with CR.

5. All other control characters are irrelevant to the protocol. YTERM does not send "stray" controls, but PCTRANS does and YTERM discards them according to the following rules:

   a. Control characters other than ESC and RS (1E hex, 36 octal) are simply ignored.

   b. If RS is received, it and the next two characters are ignored.

   c. If ESC is received, YTERM checks to see if it is followed by a character which forms a sequence SIGNIFICANT in terms of the protocol. If so, it take action accordingly. If not, it removes the ESC and the following character and ignores them.

   d. If the character following an ESC is a left square bracket ([), YTERM will also ignore subsequent characters up to and including the next control sequence final character (see ANSI standard X3.64; final characters are in the range 40-7E hex, 100-176 octal).

Framing

All transmissions which are part of the protocol should be distinguishable from things which are not. In a terminal emulation environment this is essential. In general, things which fall cleanly outside the "frames" established by the protocol should be reflected to the emulated terminal screen. Things which fall within framed areas but are not legal in the character set repertoire of the protocol should be discarded. Because the terminal emulation environment of YTERM conforms to ANSI standard X3.64 and we expect that standard to carry increasing weight in the future, we will use X3.64 recommendations wherever possible.

1. YTERM enters file transfer mode upon receipt of the three character sequence <ESC>Py (that is: escape, capital P, lower case y). The <ESC>P is a device control string (DCS) introducer, the y qualifies the device control string to be one obeying this protocol. Currently, if

YTERM sees <ESC>P but does not see lower case y, it assumes it is being sent an unimplemented device control string. It ignores subsequent characters until it receives <ESC>¢ (string terminator). During this process, it indicates rejection via a message on the 25th line.

2. All subsequent graphic characters received by YTERM are considered part of the protocol until <ESC>¢ (normal termination) or <ESC>0 (abort and discard) is received. Logically, this "string" can be indefinitely long. Actually, the string is packetized and periodic error checks are performed.

3. Eventually, the protocol will be made symmetrical. Currently, as an implementation convenience, it is not. PCTRANS always sends a string to YTERM first. If the higher protocol layer calls for YTERM to respond, he does so. When YTERM sends a "string" to PCTRANS, the opening "frame" (the <ESC>Py) is elided. YTERM simply begins sending data, knowing that PCTRANS expects it. The "string" is terminated exactly as described above, however.

While one of the partners is sending data, the other is periodically acknowledging (at specific points, to be defined presently). The sequences to be used for acknowledgment are:

1. <ESC>1 -- This means that data has been correctly received and further transmission is encouraged.

2. <ESC>2 -- This means that a correctable error has occurred and the sender should retransmit the preceding unit of data.

3. <ESC>3 -- This means that an uncorrectable error has occurred or that a retry counter has been exhausted (it is the receiver's responsibility to maintain a retry count). Transmission of the string should be considered to be at an end. In practice, PCTRANS (but not YTERM) will echo the <ESC>3 sequence with one of its own. This is done to cleanly terminate file transfer in case the <ESC>3 was typed by a human user to break a deadlock.

4. <ESC>5 -- Means that the string was rejected by the higher protocol layer. Both implementations of the protocol begin to pass data to the higher level as soon as a packet has been error checked. Allowing the higher layer to reflect a rejection through the protocol avoids lots of unnecessary transmission if the higher layer determines early on that it doesn't want the string.

PCTRANS always just sends its acknowledgments without a following carriage return. So does YTERM when running in "F" mode. In the other modes, YTERM follows its acknowledgments with a carriage return.

Basic encoding

Between <ESC>Py and <ESC>¢ the only protocol-significant characters are from the set 20 hex through 7E hex. With the exception of number sign (#,

23 hex, 43 octal), each of these characters represents itself.

The # character is used as a "knockdown indicator" to aid in representing the remaining 33 characters in the 128 character set. The "knockdown" of a character is the code for the character plus 40 hex modulo 80 hex (we could also say "plus 100 octal modulo 200 octal" or "plus 64 decimal modulo 128 decimal"). For example, the knockdown of CNTRL-A is A. A character outside the protocol-set is represented by # followed by the knockdown of the character. This same encoding must also be used to represent # itself. # is represented as #c (number sign followed by lower case c).

A double number sign (##) is also special to the protocol. The result of applying the knockdown resolution rule to this combination yields a lower case c, which could have been sent as a single character. This condition is used to signal that the next four characters contain error control information and (usually) that an acknowledgement is desired.

Special numeric encoding

In several places it becomes convenient to represent binary numbers. Whenever there is this need within the YCC "link/transport" protocol, we will use an encoding based on the characters 30-6F hex. These characters represent the numbers 0-63, respectively (six-bit significance). Where it is necessary to represent more than six bits of significance, more than one character is used. The number of characters (and hence the maximum significance) is determined by context.

Error checking and confirmation

Error checking in this protocol is based on a 16 bit CRC. The algorithm for the CRC is taken from the IBM Personal Computer Basic I/O System Cassette I/O routine, which, in turn, borrowed the algorithm from the well-researched CRC used by the SDLC protocol. YTERM/PCTRANS use an initial value of 0, rather than 0FFFFH, for the CRC. Here is a detailed statement on how to implement the CRC.

1. The <ESC>Py and <ESC>¢ which are used for framing are NOT part of the set subject to CRC computation. CRC computation begins with the first character following the <ESC>Py.

2. The characters which are input to the algorithm are the characters BEFORE knockdown encoding for the sender and AFTER knockdown decoding for the receiver. That is, where two characters are used to represent one, only the ONE is input to the CRC.

3. The CRC algorithm is a bit-wise algorithm. Each ASCII character is seven bits. The least significant bit is input to the algorithm FIRST and the most significant bit is input to the algorithm last. Any eighth bit is assumed to be for parity checking at a lower protocol

layer. It is assumed to be arbitrary and irrelevant. It is NOT input to the CRC algorithm.

4. The CRC starts out (at start of transmission or after being cleared by an error check confirmation) as sixteen bits of zero. As each bit is input to the algorithm, the CRC advances according to the following procedure:

   a. The input bit is compared to the most significant bit of the current CRC.

   b. If the bits are unequal:

      1) The current CRC is "exclusive or'd" with the bit pattern 0000100000010000 (0810 hex).

      2) The "increment" (explained momentarily) is set to one.

   c. On the other hand, if the input bit is equal to the MSB of the CRC, the CRC is not changed at this point and the "increment" is set to zero.

   d. Now, the CRC is "shifted left" by one bit (multiplied by 2, the result taken module 2**16) and the "increment" determined in the preceding step is added.

   e. The process is repeated for each of the 7 bits of each of the n characters until the time comes for error checking and confirmation.

   Both the sender and the receiver must keep track of the CRC as characters are sent and received.

   Error checking is always requested by the sender. The request takes the form of two consecutive number signs (##) followed by four bytes of ASCII, using the "special numeric encoding" described above. Currently, PCTRANS includes an error check request every 512 bytes; YTERM includes one every 128 bytes.

   a. First, the sender computes the sequence number for this check. The sequence number is a number between 0 and 15. The first check in each string has a sequence number of zero. Subsequent checks have sequence numbers which increment by one each time. For long strings, the sequence number will wrap to zero after the number 15 is used.

   b. In order assure correct delivery of the sequence number, the sender represents the sequence number as a seven bit quantity and inputs it to the CRC process as if it were a transmitted character.

   c. The sender now forms a 24 bit number by concatenating bits as follows:

      1) Four bits of zero (reserved for possible future use).

      2) The four bit sequence number.

      3) The sixteen bits of the CRC.

   d. The sender now splits this 24 bit number into four six-bit numbers. He uses the special numeric encoding to represent these six-bit numbers as four ASCII characters in the range 30-6F (see above).

   e. The sender sends two number signs followed by the four bytes just computed. Note that the sequence number has been input to the CRC but the two number signs and the four characters resulting from the encoding have not. After sending the check request, the sender WAITS for confirmation. Actually, YTERM always follows the check request with a carriage return. PCTRANS does not.

5. After unpacking the error check request, the receiver should input the sequence number to his CRC process and then compare the CRC to the CRC he has been compiling and the sequence number to the expected sequence number (NOTE: the six raw bytes of the error check request are NOT input to the CRC algorithm). If everything agrees, he now has good data, as far as it goes. The data is passed to the higher protocol layer.

   a. If the data is acceptable to the higher layer, he sends <ESC>1 and prepares to receive the next segment.

   b. If the data is unacceptable to the higher layer, he sends <ESC>5. He then regards the received string as terminated.

   In any case, we are assuming good data transmission. Eventually, the sender sends the next bunch of data. Both CRCs have been reset to zero. The next error check uses the next sequence number. Eventually, the end of the string is signaled by <ESC>¢ or <ESC>0.

All of the above assumes good, clean transmission. Let's consider what can go wrong.

1. Let's assume that the ## sequence is correctly sent but some part of the data is garbled. In this case, the error check occurs, the CRC doesn't agree. ACTION: receiver sends <ESC>2, sender retransmits the sequence in error. Both PCTRANS and YTERM will request retransmission of a packet up to 5 times.

2. If the 5th retransmission of a packet is not error free or if a more profound sort of environmental error occurs, the receipient sends <ESC>3. This ends the transmission of the string.

3. Let's assume that the ## characters are garbled. In this case, the sender is waiting for acknowledgment, the receiver is waiting for more data (doesn't know the sender wants an error check). Ideally, timeouts should handle this case. In the YTERM/PCTRANS implementation, this deadlock must be broken by a human being. If YTERM is receiving, an <ESC>2 will free things up. If YTERM is sending, a carriage return will do the same.

4. Occasionally, a positive acknowledgment may be mistakenly felt to be negative. Or, the acknowledgment got lost and the human being sent a negative one, causing the sender to retransmit.

   The receiver, seeing by the sequence number that the last sequence has been retransmitted, should send <ESC>1 and discard the data. This will allow the protocol to proceed.

5. Less likely, but still conceivable, is that the sender THINKS he has gotten <ESC>1 when, in fact, the intent was something else. In that case, the sender would go on to the next sequence while the receiver would not be expecting that. The receiver then sees a sequence number which is too high, or <ESC>¢, meaning that the sender is happy even though he shouldn't be. This is currently a nonrecoverable error.

Special pacing issues when YTERM sends.

The basic operation of the protocol has now been described. There are some additional issues which arrise when YTERM is sending.

1. When YTERM is sending in "line mode" ("H" or "B") he waits for on XON character to tell him that the host is ready to receive. In "F" mode we waits for a lower case y character to do the same thing. This character is searched for AFTER the acknowedgement of the preceding write. It is also waited for before sending an acknowledgment sequence if YTERM is the nominal receiver.

2. When YTERM is sending DATA (as opposed to acknowledgements) in the "F" mode, it waits for every graphic character to be echoed before sending another. Furthermore, it will retransmit a character if it receives a BEL (07 hex or octal) prior to the character being echoed. Before retransmitting the character, it sends the DC2 character (12 hex, 22 octal) which is the default "error reset" character for Yale ASCII.

   This echoing process takes place at the lowest level and is a source of both inefficiency and potential "hangs" in the process. We hope to eliminate or moderate the need for this by eventual changes to Yale ASCII.

3. Echo waits do not occur in the "H" or "B" modes. Characters are not echoed by the remote host (generally) in "H" mode; in "B" mode echoed characters are simply discarded when received and no synchronization is done on them.

THE YALE "FILE TRANSFER" PROTOCOL

This protocol provides a syntax and semantics for strings delivered by the link/transport protocol. This is a separate protocol built on top of the first.

1. Strings begin (following the <ESC>Py) with a three character request code. So far we have identified the following codes as being useful:

   a. WSA -- I am writing a file using "standard ASCII"; prepare to receive.

   b. RSA -- I want to read a file using "standard ASCII"; send please.

   c. W43 -- I am writing a file using "4 for 3 binary"; prepare to receive.

   d. R43 -- I want to read a file using "4 for 3 binary"; send please.

   e. SCK -- Here is the time and date where I am.

   f. ERS -- Erase file (! see below).

   g. REN -- Rename file (! see below).

   h. DIR -- I want a list of files matching this pattern.

   i. DRL -- Here is a list of files matching your pattern.

2. The request code is followed by optional operands which are specific to the code as follows:

   a. For all writes: file-name;file-data

   b. For all reads and ERS: file-name;

   c. For SCK: YYMMDDHHMMSS (using printable ASCII numerics)

   d. For REN: old-file-name;new-file-name;

   e. For DIR: file-pattern;

   f. For DRL: file-pattern;list-of-files

   Most of these are self-explanatory. The list-of-files in DRL takes the form of a pseudo-file whose format is described in the YTERM User's Guide.

It is assumed that a READ request, if not aborted, will be responded to with a symmetric write and that a DIR (if not aborted) will be responded to with a DRL. YTERM and PCTRANS always use the DOS (not host) file name and the response filename must equal the request filename exactly.

Finally, it is assumed that ERS and REN are for master-to-slave communications in a terminal-emulation environment. Most other environments would refuse to execute these requests.

The "standard ASCII" encoding described above means character-for-character 7-bit ASCII at the application level. Of course, knockdown coding is used by the link/transport layer.

The "4 for 3" encoding described above means that four 6 bit numbers are used to send three 8 bit bytes of binary data. The three 8 bit bytes are concatenated to form a 24 bit number which is then resubdivided into four 6 bit numbers. The "special numeric encoding" mentioned previously is then used here (strictly speaking, this is unnecessary at this layer since all 128 7-bit characters are usable; however, we know that use of zero-origin for numbers will be inefficient due to knockdown encoding while use of 30 origin is more efficient).

Only the "file data" part of a string is subject to 4 for 3 encoding.

When file data is not a multiple of 3 bytes in length, there will be one or two bytes left over in a 4 for 3 encoding. This residual should be padded on the right with zeros to make 24 bits before the final application of the 4 for 3 transformation. The resulting four character substring should then be truncated on the right according to the length of the source string: three characters are sent to represent two and two to represent one.