THE BCC TERMINAL SYSTEM

Paul C Heckel

Butler W Lampson

Presented at:

The Seventh Hawaii International
Conference on System Sciences
8 January 1974

Also: CACM

THE BCC TERMINAL SYSTEM*

Paul C. Heckel**
Butler W. Lampson***

Abstract

This paper describes a system for communication between a timeshared computer and its full-duplex terminals. The system consists of a communications computer that is part of the timeshared system, remote satellite computers, and connecting telephone lines. It flexibly and efficiently services a large number of terminals of various types. A description is given both of the overall system design and of the algorithms used for multiplexing character output, error detection and correction, and local echoing as each is of independent interest.

1. Introduction and Overview

Several computer communication systems have been developed recently. The best known system, and the one which has received the most attention in the literature is the ARPA Computer Network which provides a 50-kilobit network interconnecting more than 40 computers.[2,4,5,7] However, terminal-computer networks are of interest because of the cost and availability of lower speed (2400 to 9600 baud) telephone lines which can support them. Tymnet[9], one of the more interesting of the terminal-computer systems, was developed about the same time, and tackles essentially the same problem, as the system described here.

This paper attempts two things: first it provides an overall description of what the BCC terminal system does, and how it does it. Second, it describes certain algorithms in enough detail that the reader should be able to implement them. Specifically: section 2, The User Interface, describes a solution to the problem of local echo resumption; section 4.1, The Error Free Communication Link, describes the error detection and correction algorithm; and section 4.2, Multiplexing, describes the output multiplexing algorithm.

The reader who is interested in the overall description might wish to skim sections 2, 4.1, and 4.2; while the reader who is interested in one of these algorithms might wish to skim the rest of the paper.

---

* This work was done while the authors were employees of Berkeley Computer Corporation.
** 1050 Crestview Drive Mountain View, California 94040
*** Xerox Palo Alto Research Center, Palo Alto, California 94304

The terminal system is somewhat more general than is described here. (For example, telephone lines other than 4800 baud can be used.) However, if we were to describe its properties precisely, without pedagogical simplifications, we would only obscure the basic ideas.

The BCC terminal system was designed to connect (presumably remote) low and medium speed devices, such as teletypes and line printers, to the BCC-500 Computer System. The basic design objectives were to make the system efficient in the use of bandwidth and resistant to telephone line errors, while keeping it flexible and manageable so that new devices could be easily interfaced.

Briefly, the BCC-500 System consists of five microprocessors connected to a 24 bit central memory. Two microprocessors are the system CPUs; a third controls memory management (swapping between core and secondary storage); and a fourth, scheduling. The fifth microprocessor, which is called the CHIO (CHaracter Input-Output), is used by the terminal system. Each of these microprocessors is fairly powerful, with 1000 to 2000 words of 90 bit microinstructions, and a cycle time of 100 nanoseconds.

The terminal system hardware consists of the CHIO microprocessor which is connected via 4800 baud lines, to local microprocessors called DCCs (Data Communication Computers). The DCC microprocessor is identical to the CHIO except that it has a 16 (rather than 24) bit memory and a slower cycle time. Thus microprograms can run on either processor without change.

The basic service provided by the terminal system is a channel from a user's program running on the CPU to a terminal connected to a DCC. The system is organized as a collection of parallel processes which communicate by sending messages to each other. In some cases the processes run in the same processor and the parallelism is provided by a scheduler or coroutine linkage; but it is convenient to ignore such details in describing the logical structure. Here is a list of the processes involved in providing a channel from the CPU to a terminal:

User's CPU program (per channel)
CPU monitor program (per channel)
CHIO buffering (per channel)
Multiplexing (brings many channels down to one)
Error free communication link (one per CHIO-DCC link)
CHIO modem (one per CHIO-DCC link)
Telephone line (one per CHIO-DCC link)
DCC modem (one per CHIO-DCC link)
Error free communication link (one per CHIO-DCC link)
Demultiplexing (separates one channel into many)
DCC buffering (per channel)
Terminal service (per channel)
DCC interface hardware and low-speed modem
Wire or local telephone line
Terminal

The channel in the other direction contains the same modules in the reverse order.

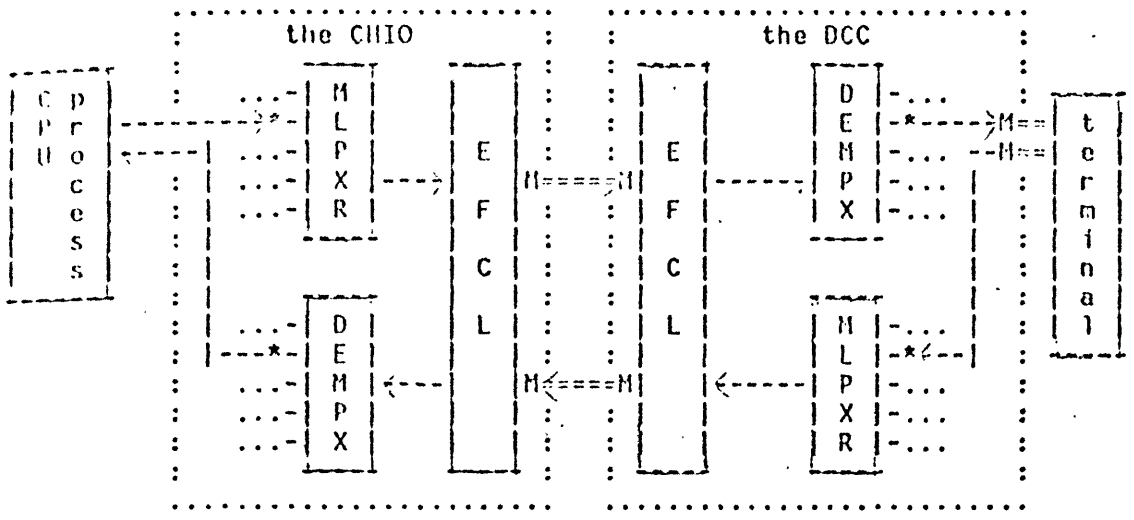

Figure 1: This shows how a single CPU process communicates with a terminal. The dashes (--) indicate the flow of characters in the system; the equal signs (==), telephone lines; the asterisks (*) buffers; and the "M"s, Modems.

The design problem was broken into three basic parts:

1) The CPU-CHIO interface, and CHIO buffer allocation;

2) The CHIO-DCC communication network: getting characters from the CHIO to the correct logical device in the correct DCC, and vice versa;

3) The DCC-terminal interface, and a simple DCC operating system.

The following philosophical decisions guided us in the design:

1) Each processor - the DCC, CHIO and obviously the system CPUs - would have its own emulated processor. When a function was not well understood, or efficiency was not important, algorithms were written in software rather than frozen in firmware (microcode), and put toward the CPU end rather then the DCC end of the network.

2) We attempted to separate the functions of the system into modules, and limit the knowledge of how each function was implemented to its module(s).

2. The User Interface

In this section we focus our attention on a single user at a

terminal attached to a DCC which sends characters to, and receives characters from, a user process running on the CPU. The terminal system is a full duplex system: the input and output channels for a device are independent except that input characters may be echoed into the corresponding output channel. In an ideal full-duplex system, all echoing of characters would be done by the user program in the CPU for three reasons:

1) This allows the program to not echo; echo a different character, or insert extra characters to make the typescript more readable.

2) It provides some valuable error checking by making it almost certain that, if the character the user wanted was the one echoed in response to a keystroke, then the program saw the same character, rather than a garbled version of it.

3) It ensures that the user's typing is properly combined with the CPU's responses - the printing on the typescript records the logical order of the interaction as seen by the user's CPU process, rather than the chronological order as seen by the user who is likely to type ahead.

Unfortunately, this ideal is impractical. If a user process were activated to echo each character, the system overhead would be large, and the user's response would be poor. Even if the echoing were done centrally in the CHIO, the users response would be poor, although the system overhead would be acceptable. However, with a little care, the system can be designed to give the effect of the CPU "echoing" each character while avoiding these problems.

The basic method is to define a rule for dealing with typed characters, called a break strategy. There are two effects of the break strategy:

Characters are automatically echoed locally (by the DCC) up to the break character. If more characters are typed, they are not echoed locally, but are echoed centrally (in the CHIO) until local echoing can be resumed.

The CPU process for the device is not activated until the break character is input (or the input buffer is almost full).

User programs in the CPU can specify and change the current break strategy (a copy of which is kept in both the CHIO and the DCC). The four break strategies are:

(A) break on no characters;
(B) break on all control characters including carriage return;
(C) break on all non-alphanumerics;
(D) break on all characters.

In addition to the break strategy, there are two flags which the user program can control:

EST (Echo Strategy) is set if characters should be echoed (it is turned off for reading passwords, for example);

EBC (Echo Break Character) is set if break characters should be echoed.

For example, a subsystem whose commands end with a carriage return would call for break strategy "B"; thus all echoing would be done locally (in the DCC) until a carriage return or control (editing) character was typed. If the user waits for the computer's response, his next input will be immediately echoed by the DCC because local echoing will have been resumed. If, however, he continues to type ahead, taking his editing for granted or typing a list of commands, these characters will not be echoed until read by the computer. Thus the output produced during a console interaction will record the interaction as seen by the CPU; no record of typing ahead will exist.

Certain aspects of this scheme are straightforward to implement. Since the break strategies are kept in both the DCC and the CHIO, the CHIO determines which characters were echoed in the DCC by executing the same logic that the DCC did. The break strategy is set by sending a message to the DCC which specifies the new strategy. The DCC responds by changing its strategy and immediately returning a message which tells the CHIO to change its strategy. Both parties know that any input characters which precede the set-strategy return message use the old strategy, and any which follow it use the new one. The setting of the strategies is synchronized in that it occurs at the same point in the input stream in each machine.

Within a given strategy, however, switching between central and local echoing is not so simple. There are three points at which echoing can occur:

In the CPU when the character is received from the CHIO and delivered to the user program (actually, the echoing is done by the CHIO when it delivers the character to the CPU);

In the CHIO when the character is received from the DCC (if the CPU has relinquished its interest in echoing, but the DCC has not yet taken it up);

In the DCC when the character is typed.

The possible transitions in the locus of responsibility for echoing are:

DCC-to-CPU
CPU-to-CHIO (and CHIO-to-CPU)
CHIO-to-DCC

We must ensure that no echos are lost, duplicated, or improperly delayed in any of these transitions. (The logic for these transitions is described in detail in Appendix A.)

The DCC-to-CPU transition is easy: the only active agent is the user typing, so that there is no possibility of conflicting decisions being made simultaneously. The CPU-to-CHIO and CHIO-to-CPU transitions are easy because they occur entirely within the CHIO, and thus can be atomic actions.

However the CHIO-to-DCC transition is tricky, because the CHIO can be telling the DCC to resume echoing at the same time that the DCC is

sending off some newly typed, unechoed, characters to the CHIO. When this happens, the DCC cannot obey the CHIO's command - it has already sent to the CHIO an unknown number of characters which must be echoed before any new characters can be echoed. The period of time, ending with the DCC's getting a message from the CHIO, and starting at the time in the DCC of the information on which the CHIO's message was based, is called an attempt-to-resume-local-echoing hiatus.

Remote echo resumption requires: first, detecting the inputting of characters during the hiatus; and second, doing something about it. Detection requires telling the DCC the last input character echoed from the CHIO, so that the DCC can tell whether any more characters have arrived since then. We do this by sequence numbering the input characters (mod 16 - because we convinced ourselves that no more than 15 characters could be in the "pipe" during the hiatus). We make sure that both machines use the same numbers. Whenever the CHIO wants the DCC to resume echoing, it sends a Request Echo Resumption (RER) command together with the character sequence number (CSEQ) of the last character it received (and echoed). When the DCC gets the RER, it determines whether the last character received from the terminal had the same CSEQ. If so, it resumes local echoing by setting the local echo mode and sending the CHIO a Resume ECho (REC) message. If not, nothing need be done because the characters which must have been input during this hiatus will eventually cause the CHIO again to attempt echo resumption. However, the DCC sends the CHIO a CSYNC message with its current CSEQ, which the CHIO uses to reset its copy of CSEQ in case the CSEQs have gotten out of sync by accident.

The efficiency of this scheme depends upon the probability that characters are input during an attempt-to-resume-local-echoing hiatus. It is a good scheme if the probability is small, but poor if it is large because (a) sending extra RERs is wasteful, and (b) the user's response is be sluggish until local echoing is resumed. (These problems do not occur if the DCC does not send a packet of input to the CHIO until a break character is typed - at which point local echoing would stop anyway.) The probability that the attempt to resume local echoing will fail is: H / I, where H is the expected duration of the attempt-to-resume-local-echoing hiatus, and I is the expected interval between input packets to the CHIO during the hiatus. For this system H was about 200 milliseconds, and I was about 4 seconds so the RERs would fail only about 5 percent of the time.

Alternatively, the DCC could remember unechoed characters for a while after sending them to the CHIO and then echo all the characters following the one specified by the CSEQ when it received the RER command. This would get rid of the acknowledgement to the CHIO and the need to retry at the cost of some buffering for each terminal in the DCC - enough to cover the maximum round-trip delay in a message sent between per-terminal processes. Our desire to minimize the amount of buffering for low-speed terminals, and the favorableness of the H/I ratio, led us to reject this method of resuming local echoing. The problem of local echo resumption has been discussed elsewhere.[9,10]

The foregoing analysis assumes that, in console interactions, neither side interrupts the other, but each waits for the other to finish. If the user "types ahead", the terminal system buffers the typing until the computer is ready to listen. While this view is valid in most cases, each will on occasion wish to interrupt the other.

The user can interrupt the computer by typing a quit character which generates a special interrupt to the user's (running) CPU process. Presumably the process will take some appropriate action, such as aborting the current computation or output. As far as the terminal system is concerned, there is nothing special about the quit sequence, except that the CHIO must be able to accept a command from the CPU to clear the output buffers for a particular terminal.

A CPU process may also want to interrupt its user. It could, of course, simply blast out a message, but this would probably result in an ugly mixture of the user's input with the characters of the message. More important, the CPU process would be unable to tell which of the input characters came before, in ignorance of, and which after, in response to, its blast. To solve this problem we introduce a control character called TAG. If the CPU outputs this control character, the DCC turns off local echoing and sends the TAG back to the CPU process. This achieves two things. First, since local echoing was turned off, the typescript will be readable. Second, the CPU process can synchronize with its user's concept of input because it knows that characters after the TAG, were typed after the TAG was processed by the DCC. In practice, the CPU process should wait a few seconds and then send a second TAG to ensure that the user had enough time to react.

3. The CHIO

The CHIO, like the other central microprocessors, communicates with the CPU via the system's memory. Each processor can also send the other an attention signal. The CPU sends messages to the CHIO by writing them in agreed-upon memory locations and then sending the attention signal. If the CPU expects immediate response from the CHIO, it waits for the response to appear in another agreed-upon location. Otherwise the CPU goes about its business. At some later time (e.g. when a break character has arrived, or the output buffer is nearly empty) the CHIO can send the system scheduler a signal requesting the wakeup of the proper CPU process.

In addition to the microcode for its normal functions, the CHIO has a microcoded emulator for an instruction set similar to the SDS 940 (chosen because support software was available). This emulated 940 runs test programs, and adjusts CHIO buffer allocation parameters to prevent exhausting buffer space.

The interface which the CHIO presents to the CPU is a collection of buffered simplex data channels. There is one input channel and one output channel for each terminal, related only in that input characters may be echoed into the corresponding output channel. In addition to its buffering, each channel has some state which can be read and set by the CPU: break strategy, speed and character structure, and the process to "wake up" when the channel needs service.

There are three basic CPU-to-CHIO commands that a user's CPU process may use. They are *Read String*, *Peek String*, and *Write String*. *Read String(L,N)* reads, and removes, characters from the CHIO's buffers for line L. It stops at the first break character or the Nth character - whichever is first - so the reading program won't get more input than it is prepared to deal with. *Peek String(L,N)* is identical to Read String except that the characters are not removed from the

buffer. *Write String(L,S)* writes string S into the CHIO's buffer for line L.

Internally, the CHIO has a character buffer for each input and output channel. Each CHIO buffer is a list of 21-character (8 word) blocks. If too many of these blocks are used for an output line, the Write String will do the write, but will return with an indication that the CPU should send no more characters. When this happens, the CPU program will normally block the process which is generating the output. When the CHIO finds that its buffer is nearly empty, it will send the CPU process a "wakeup".

This scheme, and many other features of the CPU-to-CHIO interface, require that the CPU program be friendly. User programs are not allowed to send commands directly to the CHIO, but must filter them through the system's monitor, which does the necessary error checking - in this case by blocking processes which uncooperatively refuse to stop outputting to a "full" line.

4. The Communication Link

The communication network consists of one CHIO connected to several DCCs via 4800 baud telephone lines. Characters go from the CHIO directly to the destination DCC; there is no store-and-forward capability.

The next few sections focus attention on the communication link between the per-channel processes. This link involves the CHIO, one DCC and the connecting telephone line. It is convenient to divide this link into two parts:

1) The Error Free Communication Link (EFCL) consists of (a) identical modules in the CHIO and DCC, and (b) the connecting telephone line. Its function is to provide (presumably) error free transmission of a single stream of characters between the two machines.

2) Multiplexing, which converts this single channel (the EFCL) into separate channels, one for each terminal, plus a few extra for talking to global processes in the DCC - such as the process which reports incoming calls.

The terminal system was designed to know as little as possible, about actual devices. It delivers characters unaltered from the input devices to the CPU which is responsible for converting them to the internal character set. (We considered putting the mapping to an internal character set in the DCC. However, we felt it would be best to keep the translation in one place - the CPU - until we had experience with the terminal system.) Characters in the range 0 to 37 octal are used internally as control characters by the terminal system, and may not be sent to terminals in the obvious way. Some of these control characters, like the previously mentioned RFR, have internal meaning to the terminal system and will be rejected by the CHIO if the CPU tries to send them. Others, like TAG, which can legally be sent by a user program, will result in some action by the system. Data characters in this range must be sent as two characters: the control character SHIFT1, followed by 40 plus the desired character. Thus character code 13 would be sent as SHIFT1 followed by 53. This scheme allows the system

to interface with any 8 bit device, use 8 bit data paths throughout, and still encode its control messages conveniently.

When the terminal system transmits a number from one computer to the other, as with the character count following RFR, it must encode the number into the range 40 to 377 octal (usually by adding 40 to the number).

4.1 The Error Free Communication Link

The terminal system is built out of a number of processes which interact by sending messages to each other. When the source and the destination of a message are in the same machine, it is convenient and reasonable to assume that the message can be transmitted without error. If the message must pass from one machine to another, it is still *convenient* to assume that there will be no errors, but it is no longer *reasonable* unless precautions are taken, since the raw communication path provided by modems and telephone lines is liable to errors. An important component of the terminal system, therefore, is the collection of programs and conventions which construct a virtual, error-free communication link (EFCL) from the real, error-prone one.

From the viewpoint of its users (the multiplexing and demultiplexing processes) the EFCL is a full-duplex channel which processes a character stream which is segmented into 13-byte messages. It does not interpret these messages in any way (but three characters in the range 0-37 must not appear in them: "ign", "null" and "syn"). The two halves of the channel are not entirely independent; each half needs the other to return requests for retransmission when errors are detected.

To minimize the bandwidth used for error control, only negative acknowledgements, called retransmission request (RTRs), are transmitted. A receiver sends a RTR whenever it receives anything other than a legal message. Messages are sequence-numbered within the EFCL. Message N always follows message N-1, unless the EFCL is recovering from an error. Thus the receiver always knows which message it expects next, and sends a RTR if it gets anything else. The sender saves each message on a lookback queue until it is sure that it will not have to retransmit it. This approach uses bandwidth more efficiently than a simple positive-acknowledgement scheme, but at the cost of more complex logic.

The timing information which makes the negative acknowledgement scheme work is provided in the following way. Each (full-duplex) EFCL contains a 32 "envelopes" in which messages can be sent. The envelopes are numbered 0 to 31, and they pass back and forth between the two ends of the line. If a sender puts a message into envelope N, it must keep a copy of the message for possible retransmission until it gets envelope N back. Once this happens, it knows that the message was successfully received, and its copy can be discarded. Envelopes are sent in order, envelope N+1 following envelope N (mod 32), except when a retransmission occurs. The "positive acknowledgement" of a message block is the successful reception from the other computer of a message with the same block number (a message in the same envelope). Since envelopes are not explicity identified, no bandwidth is used for the positive acknowledgement.

It is possible for all 32 envelopes to be at one end (and in fact the link is initialized in this state). When this happens, the other end will be keeping copies of 32 messages. Each end has 32 message buffers, called envelope buffers, each of which is permanently associated with a particular envelope. When envelope *N* is present, then envelope buffer *N* is free; when envelope *N* is absent, then envelope buffer *N* contains a copy of the message which was sent in that envelope. Free envelope buffers (available envelopes) are kept on a free queue; full ones waiting for transmission, on the output queue; and full ones that have been sent but whose reception has not been acknowledged (whose envelope has not yet come back), on the lookback queue.

Input messages are stored in a different set of buffers, called in-buffers. These have no permanent numbers.

The following diagrams show an idealized picture of the EFCL's structure:

SEND

user - OUT - output queue - TR - hardware

RECEIVE:

hardware - READ - read queue - RCV - input queue - IN - user

Here the capitalized words are the names of the modules which comprise the EFCL, and the dashes are coroutine linkages. Two modules separated by a queue can execute in parallel. We will proceed by describing each module; quite detailed programs for the entire system can be found in Appendix B.

OUT takes the envelope buffer from the front of the free queue (the next available envelope) and puts into it a 13-byte block which it gets from its user, and a 2-byte checksum which it calculates. It then puts this envelope buffer on the end of the output queue (from which it will be read by TR).

TR takes an envelope buffer from the front of the output queue and does two things with it. First, it outputs the buffer's contents to the hardware. (If the output queue is empty, TR sends "ign" bytes which are ignored by the receiver.) Second, TR puts the buffer on the end of the lookback queue if it is an envelope buffer (it could be an RTR or RTA). This envelope buffer will be moved from the lookback queue to the end of the free queue when its "envelope" comes back. If a retransmission request (RTR) is received before this happens however, the envelope buffer will be put back on the output queue.

IN takes an in-buffer from the input queue, delivers its 13 data bytes to the user, and returns the in-buffer to the input free queue.

There are four block types that RCV can find on the read queue: a data block (that has no errors), an RTR (retransmission request), an RTA (retransmission acknowledgment), and an error block (anything else - but most likely a block with a bad checksum). RCV can be in one of three states corresponding to its "expecting" one of the first three block

types. RCV takes the in-buffer from the front of the read queue. There are four possibilities:

1) *If RCV expects and gets a data block:* It (1) puts the buffer on the end of the input queue, and (2) moves the envelope buffer on the front of the lookback queue to the end of the output free queue - acknowledging the reception of that envelope buffer's envelope. It will still expect data blocks.

2) *If RCV gets an error block or any block other than what it expected:* It deletes any RTRs or RTAs on the output queue, and moves any envelope buffers to the end of the lookback queue. It generates an RTR for the envelope whose buffer is on the front of the lookback queue (call it L), and puts this on the of the output queue. Finally, RCV puts a synchronization block (see READ) on the output queue which will force the READ at the other computer to synchronize to the correct character position. It then expects an RTR. (If RCV was expecting a data block, and it gets an RTR, it first processes the RTR as an error block, and then as an expected RTR.)

3) *If RCV expects and gets an RTR:* Assume the RTR is for envelope N. A retransmission acknowledge (RTA) is appended to the output queue specifying that block (envelope) N is the following block. The buffers on the lookback queue, from block N to the end of the queue, are copied to the end of the output queue. RCV then expects an RTA for envelope L.

4) *If RCV expects and gets an RTA for block L (for which it asked):* RCV expects data blocks again.

READ takes characters from the input hardware, recognizes messages, puts them into in-buffers, which it appends to the read queue. Its life is complicated by the need to parse messages from the stream of garbage which may be arriving over the telephone line. The hardware helps by recognizing a string of more than 16 zero bits as part of a resynchronization sequence. The first 16 zeros are passed on as bytes (the "null" byte is the one with 8 zeros). If there are more zeros, they are absorbed by the hardware until a one bit appears. This bit is used to define byte boundaries in such a way that if a "syn" character is the first thing sent after a string of "null"s, then it will be correctly received.

READ looks for a syntactically correct, properly checksumed, block (15 non-null bytes after "ign"s are filtered out). If it sees anything else, it puts an error block on the input queue, and throws everything away until the sychronization sequence "null" "syn" appears, and then starts looking for a correct block again.

Finally, we clear up a loose end. The just described scheme works as long as no RTR's or RTA's are "lost". This case is handled as follows: whenever an error is detected, a timer is set (or reset if it is already set) to trigger in 300 milliseconds. it is already set). This timer is turned off when an RTA is received. If the timer "goes off" first, however, the EFCL puts an error block on the read queue, forcing a new attempt to resume normal communication.

It should be noted that the EFCL is based on the assumption that errors occur infrequently. If there are no errors on the line, the only inefficiency is represented by the check characters. When an error is detected, however, the EFCL stops transmitting data blocks for about 200

milliseconds. Since available telephone lines and modems guarantee no more than 1 error per 10⁵ bits, we can expect one error every 20 seconds on a 4800 baud line, with an efficiecy of 99 percent. In fact, things are really better than this because errors tend to come in bursts. If we get one error, we can expect others in the next few hundred milliseconds. There are two effects of this:

1) The mean time between error bursts, the periods for which the EFCL operates normally, is longer than 20 seconds.

2) After an error is detected, the EFCL looks for RTRs and RTAs. These messages are checksumed with a greater redundancy than the data blocks. Thus the EFCL is least likely to miss detecting an error when the probability of its occurence is highest.

4.2 Multiplexing

The choice of methods for converting the single EFCL channel into one channel for each terminal is dominated by the demands placed on two scarce resources: bandwidth on the EFCL, and buffer space in the two computers. Input multiplexing is fairly easy to handle because the volume of input is low, and the CHIO has a large amount of buffer space. Output is hard because the volume is greater, bandwidth must be shared equitably, delays in starting output to any terminal must be short, and the buffering done in the DCC should not be too great. Available telephone lines and modems provide the same amount of EFCL bandwidth in both directions; thus the efficient utilization of bandwidth is more important in the output direction than in the input direction.

A basic principle underlying the system is that characters are sent only if the receiving computer can accept them. There is no provision for transmission of control messages between the processes which handle single terminals (except for the special case of local echo resumption).

This principle causes no trouble for input multiplexing because the CHIO "always" has enough buffer space to store the demultiplexed input character streams. The input data rate is usually low, and the user doesn't type ahead very far. Furthermore, the maximum interval between break characters is short (150 characters), and since the DCC loses control of echoing at a break character, the CHIO can discard input beyond the break character, replacing it with an overflow indicator. When the CPU sees this indicator, it can respond appropriately so that the user will never be in doubt about which input was kept and which was thrown away. (This will only happen if the user program is responding very slowly and the user is typing ahead regardless.) Of course mechanical input devices (such as paper tape readers) have quite different properties, so a program which wants to input from such a device can ask the system for extra CHIO buffer space.

For output multiplexing we must be more careful, because the user program can produce characters very fast, and we don't want to have much buffering in the DCC. Furthermore, we cannot send output in large blocks because this causes excessive delay in sending to terminals whose output happens to get caught behind a few of these blocks. As a consequence, we must regulate the average rate at which the CHIO sends characters to a terminal so that it is only slightly less than (ideally equal to) the rate at which the terminal can take them. To minimize

buffering in the BCC and avoid excessive startup delays, the interval over which flow averaging is done should be as short as possible. Finally, we should take advantage of the fact that output messages tend to be quite long.

Input multiplexing is simple and straightforward. The input stream carries a sequence of messages, each of which is of consists of a burst marker, a device number, a sequence of input characters, and is terminated by the next burst marker. Input for a device is not sent to the CHIO until either (a) the input buffer is almost full, or (b) a break character has been typed. Thus several-character bursts can be sent even for low speed devices.

4.2.1 The Meta-Multiplexing Algorithm

The output multiplexing algorithm is based on the simple fact that the set of currently outputting teletypes changes slowly. Suppose, to begin with, it does not change at all. Then *we can transmit only data in the output stream if all the multiplexing information is contained in the multiplexing and demultiplexing algorithms.*

We can split the multiplexer and demultiplexer into two modules each, a Meta-Multiplexer and a Bandwidth Allocator in the transmitting computer, and a Meta-Demultiplexer and an identical Bandwidth Allocator in the receiving computer. The meta-algorithms merely require that the Bandwidth Allocator determine which channel goes with each character position in the EFCL stream. The only constraint on the code for the Bandwidth Allocator is that it only reference state which can exist in both computers - a constraint implied by having identical copies in each computer.

The "correctness" of the Multiplexer *does not depend* on the "correctness" of the Bandwidth Allocator. For example, suppose the Multiplexer's Bandwidth Allocator selected channel 3 to send the next character to, when it should have selected channel 4. The Demultiplexer's Bandwidth Allocator will send the character to channel 3 (rather than 4). The effect of such an error would be to either "cheat" a channel (4 in this case) out of its intended share, or to send characters to a channel faster than its terminal could dispose of them (3 in this case), or both. But characters would always go to the correct channel.

The set of active output channels is not invariant, but the Meta-Multiplexer can be easily extended to the more general case of a (slowly) varying set of active channels at the cost of adding three control characters to each stream of data characters for a channel. We require that the state table (in each computer) contain an "active" bit for each potentially active output channel. The set of active output channels is the set of channels with the "active" bit set.

If the Meta-Multiplexer wants to add a new active channel it does two things simultaneously: it sets the "active" bit for the channel, and it sends the Demultiplexer an Insert New Channel (INC) character followed by the number of the channel being activated. The Meta-Demultiplexer, when it gets the INC, sets its "active" bit for the newly inserted channel, but otherwise ignores the two characters.

Similarly, when the Meta-Multiplexer finds that there are no more characters for an active channel, it deactivates the channel by resetting the "active" bit, and sending a Delete Old Channel (DOC) character. The Meta-Demultiplexer, when it gets the DOC, resets the "active" bit for the selected channel, but otherwise ignores the character. In this case it is unnecessary to send the channel number, since the DOC is sent in place of a data character and the receiver therefore knows which channel is involved.

The bandwidth efficiency of a multiplexing algorithm is the percentage of the characters in the multiplex stream that are data characters. The efficiency of this algorithm is N/(N+3) where N is the average number of characters per output stream. (3 is the number of control characters added to the stream.) N is likely to be largest, and thus the algorithm most efficient, when the output EFCL is at or near saturation. If the average output stream to a channel is 22 characters, the efficiency of the multiplexer is 88 percent.

4.2.2 The Bandwidth Allocator

The Bandwidth Allocator is described more as an example of an algorithm that can be used with the Meta-Multiplexer, then for its intrensic merit. This algorithm has three constraints:

1) It must have the properties demanded by the meta-multiplexing algorithm. These properties are implied by the requirement that identical copies of the algorithm run in both machines.

2) It must not send characters to a device faster than the device can process them.

3) It must be able to multiplex devices of any speed. It is this requirement that presents most of the problem.

Time is arbitrarily divided into 100 millisecond intervals. This turns out to be 52 characters (52 data + 8 check characters times 10 intervals = 600 characters per second or 4800 baud). For each channel we keep the number of characters it can receive in the 100 millisecond interval, called IR.FR (IR is the integer part of the rate, FR the fractional part). IR.FR is 1 for a channel driving a 10 cps device; 3 for a 30 cps channel, and 1.48 for an IBM 2741 channel. We also keep the number of characters to be sent in the current interval called CI.CF, which is set to zero initially. The basic idea is to send IR and IR+1 characters in an interval, alternatelly, in such a way that the average number of characters per interval will be IR.FR. The following algorithm will give CI the values IR and IR+1 appropriately.

At the beginning of each interval, the following computation is performed for each channel:

CI.CF ← 0.CF + IR.FR

and the Bandwidth Allocator will try to send CI characters to the channel in that interval.

The Bandwidth Allocator has two passes. In the first pass it selects all (active) channels and sends each of them the fewer of 3 or

CI characters. It will finish this pass even if it has to stretch the interval beyond 52 characters (1/10 second). If fewer than 52 characters were multiplexed in the first pass, channels that can accept more than 3 characters are sent CI-3 characters until the interval is complete (52 characters are multiplexed). This allows high speed devices such as printers to take up the slop in times of plenty, while slowing output to all devices when saturation occurs.

At the end of each interval, a CHS (Check Synchronization) Control character is inserted. This checks and resets the synchronization of the Multiplexer and Demultiplexer (in theory, loss of synchronization would only occur if the LICI failed to detect an error), and provides a "do nothing" control character if there is no output to do.

5. The Data Communications Computer

The Data Communications Computer (DCC) was designed with three criteria in mind.

> It should efficiently handle input and output to a large number of low-speed (up to 300 baud) devices;
>
> it should provide flexibility, especially in interfacing with a variety of devices;
>
> it should be controllable from the CPU so that operator intervention is not required except in the case of hardware malfunction.

In the normal case, the DCC is expected to interface with up to 200 devices, mostly low-speed terminals. It includes an interpreter for a 16-bit instruction set called the Remote Processing Unit (RPU); those parts of the DCC's job which do not have to run at microcode speeds are executed by the RPU.

The DCC maintains a Device Table, indexed by the device number, which contains the partially assembled and disassembled characters for the microcoded bit scanner, descriptors for the input and output character buffers, and fields with control information such as the break strategy.

Low speed devices are bit scanned by the microcode and the assembled character is echoed (if local echo mode is set) and then stored in the input buffer for the device. However, if the input buffer is full, due to either a communication line malfunction or an unusually heavy load on the Input Multiplexer, the character is neither stored nor echoed. Thus the user does not get false feedback if his character was lost by the DCC. The Input Multiplexer removes characters from the input buffer and multiplexes them for transmission to the CHIO when requested to do so by the LICL

Output is similar to input: The Demultiplexer puts characters in the device's output buffer, from which they are later removed by the output bit scanner. Remember, it is guaranteed that the Multiplexer will not deliver characters faster than the output bit scanner can dispose of them.

Line printers, card readers, displays, and other devices whose speed is too high to be bit scanned are handled differently on the "device side" of the DCC. An RPU task inputs characters from these devices and stores them in the input buffer for the multiplexer to pick up. Similarly, for each output device, an RPU task gets output characters from the devices's output buffer (where they were put by the demultiplexer) and outputs them to its device. These tasks are activated by the input and output interrupts from the hardware interface for medium-speed devices, and by the output demultiplexer when it delivers a character.

(The RPU has 7 index registers, one of which is the program counter; 15 memory reference instructions; and 24 register to register instructions. There are four addressing modes: immediate, direct, indirect, and scratchpad; the last gives the RPU access to the working registers of the underlying microprocessor. There are two indirect word types: an indexable core pointer which is needed because the address of the instruction is a 7 bit signed field, and an indexable field descriptor which specifies a word displacement and an arbitrary contiguous bit field within the word. The memory reference instructions include load, store, call, branch, and add to memory. The register to register instructions include the standard arithmetic, logical, shift and cycle operations as well as input-output instructions. The RPU emulator has an instruction execution time of about 4 microseconds and uses 130 words of microcode.)

The DCC maintains a table which contains the state of each RPU task. If there are no normal (microcoded) DCC functions to be done, the DCC runs the highest priority RPU task until either it blocks or a normal DCC function must be done (signaled by a hardware interrupt to the microprocessor). Tasks have fixed priority: lower priority tasks will not run while higher priority tasks are active.

RPU tasks are expected to perform several functions: interfacing with devices that are not bit scanned, answering the phone and recognizing the device type, and DCC initialization and buffer allocation. Some of these functions are done in conjuction with a controlling CPU process, using one of the channels for communication.

This brings us to the third function of the DCC: initialization from the CPU. This is a problem of some difficulty because of the havoc that a misbehaving DCC process can wreak: it can clobber core and even turn off the 4800 baud line that connects it with the CHIO. A design that requires the DCC to prevent RPU tasks from doing anything illegal is infeasible without restricting the RPU's ability to do anything the microcode can do. Thus it is impossible to guarantee that the DCC can always be reloaded by the CPU. However, it is the normal mistakes of ordinary code, rather than the sophistry of the experienced knave, that we are trying protect against. We can therefore do quite well at the expense of putting a glitch in the EFCL.

There are 4 parts in the initialization procedure.

First, to handle the (rare) worst case where the communication line has been turned off, or the DCC is in an unrecoverably bad state, there is a pushbutton on the DCC which, if pushed, will initialize the DCC so that it can be loaded over the EFCL. It simply causes a branch to the microcode initialization location.

Second, whenever the EECL is about to read an input character from the hardware, it checks for the control character HHH. If it gets 3 of these in a row (two in a row could be checksum characters), it does the same initialization as the console pushbutton. The effect of this initialization is to allow the EECL and the Demultiplexer to operate, albeit in a rudimentary way: not all of the tables are initialized - just the bare minimum.

Now the CPU can load whatever parts of the DCC it wants by sending a load Remote Concentrator (LRC) control character, followed by loading information. Part of the loading information is a flag that indicates whether RPU tasks should run. Thus RPU tasks can be turned off until core has been loaded; then they can be turned on with a last LRC, and a just loaded initializaton task can run.

Finally, this initialization task can load microprocessor registers, turn on any input and output devices, and do other initialization.

6. Some Facts

The DCC contains about 500 words of microcode, the CHIO about 900. The microcode for the EECL and the Bandwidth Allocator is identical in both machines. The DCC has less microcode than the CHIO because great effort was expended in minimizing the DCC microcode so as to reduce the cost of replicating the machine. This was less important with the CHIO, where efficiency and ease of understanding took precedence. The terminal system was first simulated, and then a working system was brought up and tested. It was found to be working well, but extensive testing and use did not occur due to the unfortunate demise of BCC.

Bibliography

[1] S Carr S Crocker V Cerf
HOST-HOST Communication Protocol in the ARPA Network
Proc AFIPS SJCC 1970

[2] S Crocker J Heafner R Metcalfe J Postel
Function-Oriented Protocols for the ARPA Computer Network
Proc AFIPS SJCC 1972

[3] H Frank I T Frisch W Chou
Topological Considerations in the Design of the ARPA Computer Network
Proc AFIPS SJCC 1970

[4] F Frank R Kahn L Kleinrock
Computer Communication Network Design- Experience with Theory and Practice
Proc AFIPS SJCC 1972

[5] F E Heart R E Kahn S M Ornstein W Crowther D Walden
The Interface Message Processor for the ARPA Computer Network
Proc AFIPS SJCC 1970

[6] L Kleinrock
Analytic and Simulation Methods in Computer Network Design
Proc AFIPS SJCC 1970

[7] S Ornstein F Heart W Crowther H Rising S Russell A Michel
The Terminal IMP for the ARPA Computer Network
Proc AFIPS SJCC 1972

[8] R Thomas D Henderson
McROSS- A Multi-Computer Programming System
Proc AFIPS SJCC 1972

[9] L Tymes
TYMNET - A Terminal Oriented Communication Network
Proc AFIPS SJCC 1971

[10] J Davidson
An Echoing Strategy for Sattellite Links
Network Information Center 10599, RFC 357, July 1972

[11] R. Metcalfe
Packet Communication
MIT Project MAC Technical Report 114,
December 1973, revised PhD Thesis,
Harvard University, May 1973

[12] P Heckel
The Communications System - Phase II
BCC Working Document number CS/S-24, 1 May 1970

Appendix A

This section describes the state information and the transitions that occur in the CHIO and the DCC as responsibility for echoing is transferred.

Each machine has an echo source ES and a character sequence number CSEQ. The ES may be:

CPU - a character is echoed when it is passed to the CPU.
CHIO - a character is echoed when it is received by the CHIO.
DCC - a character is echoed when it arrives in the DCC from the terminal.

The DCC does not distinguish CPU and CHIO as echo sources.

The CHIO state transitions are:

1) CSEQ is incremented whenever a character arrives from the DCC;
2) CSEQ is set to N whenever a TAG(N) message arrives from the DCC;
3) ES := CPU whenever a break character arrives from the DCC;
4) ES := CHIO whenever the CPU does a Read String and the input buffer is empty;
5) ES := DCC whenever an REC message arrives from the DCC;
6) A RER(CSEQ) message is sent to the DCC whenever transition (4) occurs and the output buffer is empty, or the output buffer becomes empty and ES := CHIO.

The DCC state transitions are:

1) CSEQ is incremented whenever a character arrives from the terminal;
2) ES := CPU whenever a break character arrives from the terminal;
3) ES := DCC whenever a RER(N) message arrives from the CHIO and N=CSEQ. When this happens, an REC message is sent to the CHIO;
4) IF a RER message arrives and transition (3) does not apply, a TAG(CSEQ) message is sent to the CHIO;

Appendix B

This section contains a program for the EFCL Algorithm.

```
DECLARE RECORD envelope buffer
   (number : INTEGER, value : STRING(15));
DECLARE RECORD in-buffer ( value : STRING(16));
OUT(data: STRING(16)):
   DECLARE b : POINTER TO envelope buffer;
   b := first buffer on the output free queue; remove b from the
   queue; write , data and a checksum (2 bytes) into value(b);
   add b to the end of the output queue;
TR: takes no arguments;
   DECLARE b : POINTER TO envelope buffer;
   loop:
    IF the output queue is empty THEN send a "syn" to the
    hardware ELSE BEGIN
      b := first buffer on the output queue; remove b from the
      queue; send number(b) and then value(b) to the hardware,
      one character at a time;
      add b to the end of the lookback queue
    END;
    GOTO loop;

IN: RETURNS (data : STRING(13));
   DECLARE b : POINTER TO in-buffer;
   b := first buffer on the input queue (wait until there is
   one);
   data := the 13 data bytes from value(b);
   put b on the input free queue;
RCV: takes no parameters;
   DECLARE b : POINTER TO in-buffer;
   loop: IF read queue is empty THEN GOTO bad input;
    b := first buffer on read queue; remove b from the queue;
    IF value(b)[1] = number(first buffer on the lookback queue)
   and checksum is good THEN BEGIN
      put b on the end of the input queue;
      take the first buffer off the lookback queue and put it
      it on the end of the output free queue;
   END
   ELSE IF value(b) is a RTR and checksum is good THEN BEGIN
      N := the parameter of the RTR;
      remove buffers N through the end from the lookback queue
      and put them on the front of the output queue in that
      order (so that buffer N is the first).  Then put a sync
      message on the front of the output queue;
   END
   ELSE BEGIN
      bad input:
       IF time > timeout THEN BEGIN
         put a RTR(Nin) and then a sync message on the front of
         the output queue;
         timeout := time + timeoutinterval
      END
   END;
   GOTO loop;
READ: takes no arguments;
   DECLARE b : POINTER TO in-buffer, i, c: INTEGER;
```

```
loop;
 b := first buffer on input free queue; remove b from queue;
read block:
 i := 0;
read char:
 c := hardware input character;
 IF c = "syn" THEN GOTO read char;
 ELSE IF c = "null" THEN BEGIN
   read sync:
    c := hardware input character;
    IF c = "null" THEN GOTO read sync;
    ELSEIF c = "syn" THEN GOTO read block
    ELSE BEGIN
      wait for null:
       c := hardware input character;
       IF c = "null" THEN GOTO read sync
       ELSE GOTO wait for null
    END
 END
 ELSE BEGIN
   value(b)[i] := c; i := i + 1;
   IF i = 16 THEN put b on read queue ELSE GOTO read char;
END;
GOTO loop;
```