Computer Systems

G. Bell, D. Siewiorek, and S. H. Fuller, Editors

A Terminal-Oriented Communication System

Paul G. Heckel
Interactive Systems Consultants

Butler W. Lampson
Xerox Palo Alto Research Center

This paper describes a system for full-duplex communication between a time-shared computer and its terminals. The system consists of a communications computer directly connected to the time-shared system, a number of small remote computers to which the terminals are attached, and connecting medium speed telephone lines. It can service a large number of terminals of various types. The overall system design is presented along with the algorithms used to solve three specific problems: local echoing, error detection and correction on the telephone lines, and multiplexing of character output.

Key Words and Phrases: terminal system, error correction, multiplexing, local echoing, communication system, network

CR Categories: 3.81, 4.31

Authors' present addresses: P.C. Heckel, Interactive Systems Consultants, P.O. Box 2345, Palo Alto, CA 94304; B.W. Lampson, Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, CA 94304. The work described in this paper was done while the authors were employed by the Berkeley Computer Corporation, Berkeley, Calif.

1. Introduction and Overview

A number of computer communication systems have been developed in the last few years. The best known such system is the Arpanet, which provides a 50-kilobit network interconnecting more than 40 computers [1-3]. By contrast, the system described in this paper connects computers with terminals, rather than with each other. Such terminal networks are of interest because there are many requirements for connecting a number of geographically distributed terminals with a centrally located computer, and because terminal networks can use medium speed (2400-9600 baud) telephone lines which have reasonable cost and wide availability. Another system tackling essentially the same problem is Tymnet [5], which was developed at about the same time as the system described here. Of course, a general-purpose network like the Arpanet can (and does) carry terminal traffic [4].

Our system was designed to connect (presumably remote) low and medium speed devices, such as teletypes and line printers, to the Berkeley Computer Corporation's BCC-500 computer system. The basic service provided is a full-duplex channel between a user's terminal and his program running on the BCC-500 CPU. The design objectives were to make the system efficient in the use of bandwidth and resistant to telephone line errors, while keeping it flexible so that a wide variety of devices could be handled.

The paper provides an overall description of what the BCC terminal system does and how it does it. In addition, it presents in detail the solutions to three specific problems: local echoing (see Section 2); error detection and correction on the multiplexed telephone line (see Section 4.1); and output multiplexing (see Section 4.2).

The structure of the system is shown in Figure 1. The hardware components, named along the heavy black line, are

- A CPU on which user programs execute;
- A central dedicated processor called the CHIO which handles all character-oriented input-output to the CPU;
- A number of small remote computers called *concentrators* to which terminals are connected, either directly or via standard low-speed modems and telephone lines; and
- Leased voice-grade telephone lines with medium-speed (e.g. 4800 baud) modems which connect the concentrators to the CHIO.

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. Figure 1 shows the major processes involved in providing a channel from a user program to a terminal and back, and indicates how they are interconnected.

We describe the components of the system in turn, starting with the user's program and working out toward the terminal.

2. The User Interface

In this section we focus our attention on a single user at a terminal. The terminal is attached to a concentrator which sends characters to, and receives characters from, a user program running on the CPU. The terminal system is full-duplex: the input and output channels for a terminal 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) The program can omit an echo, echo a different character, or insert extra characters which make the typescript more readable.

(2) The user can type ahead of the program's responses and be sure that his typing is properly combined with the responses, so the printing on the typescript records the logical order of the interaction as seen by the program, rather than the chronological order as seen by the user.

(3) There is some valuable error checking since it is almost certain that, if the character the user thinks he typed is the one he sees echoed, then the program saw the same character and not some garbled version of it.

Unfortunately this ideal is impractical. If a user program were activated to echo each character, the system overhead would be large and the response time would be long. Even if the echoing were done centrally in the CHIO, the response time would still be long, although the overhead would then be acceptable. With a little care, however, the system can be designed to simulate the ideal while avoiding these problems.

The basic method is to specify a set of *break characters* which, like a pause in conversation, indicate points at which the user *might* expect a response. Such a set is called a *break set.* The terminal system then knows that, if a break character has not yet been typed, the input cannot elicit a response from the computer. This fact has four useful consequences:

(1) Characters can be echoed locally (by the concentrator) up to a break character. If more characters are typed, they are not echoed locally, but are echoed centrally (by the program or the CHIO) until local echoing can be resumed.

(2) Input characters need not be sent from the concentrator to the CHIO until a break character is typed.

(3) The user process waiting for input need not be activated until a break character arrives (or the input buffer is almost full).

(4) The break character provides a natural boundary for blocks of characters delivered from the CHIO to the CPU.

User programs can specify and change the current break set, a copy of which is kept in both the CHIO and the concentrator. We defined four break sets: (a) no characters; (b) all control characters, including carriage return; (c) all nonalphanumerics; (d) all characters. In addition to the break set, there are two flags which the user program can control: *DontEcho* prevents characters from being echoed; it is set when a password is being typed, for example. *DontEchoBreak* prevents break characters from being echoed.

For example, a subsystem whose commands end with a carriage return can call for break set (b), and all echoing will be done in the concentrator until a carriage return or control (editing) character is typed. If the user waits for the computer's response, his next input will be immediately echoed by the concentrator 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 they are read by the computer. Thus the output produced during a console interaction will record the interaction as seen by the program; no record of typing ahead will exist.

Certain aspects of this scheme are straightforward to implement. Since the break sets are kept in both the concentrator and the CHIO, the CHIO determines which characters were echoed in the concentrator by using the same algorithm that the concentrator used. The break set is changed by sending a message to the concentrator, which specifies the new set. The concentrator responds by changing its set and immediately returning a message which tells the CHIO to change its set. Both parties know that any input characters which precede the return message use the old set, and any

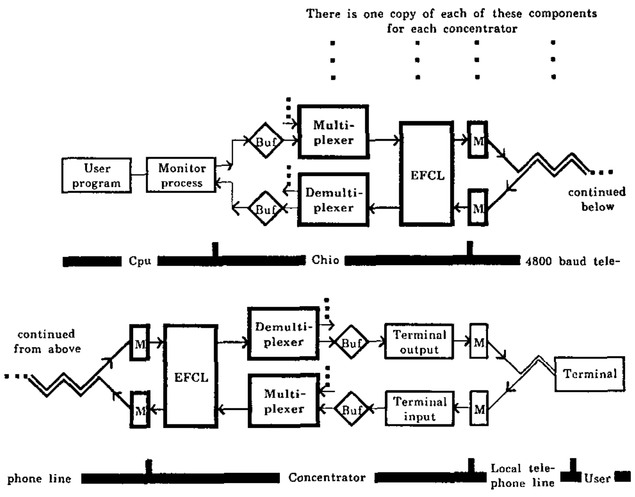


Fig. 1. Structure of the terminal system.

Components which serve many channels are bold; those which serve only one channel are light. M stands for modem.

which follow it use the new one. The changing of the break sets is synchronized; i.e. it occurs at the same point in the input stream for each machine.

Switching between central and local echoing is not so simple. There are three points at which echoing can occur:

- When the character is delivered to the user program (actually the echoing is done by the CHIO when it delivers the character);
- In the CHIO when the character is received from the concentrator if the program has relinquished its interest in echoing, but the concentrator has not yet taken it up;
- In the concentrator when the character is typed.

The possible transitions in the locus of responsibility for echoing are: concentrator-to-program, program-to-CHIO and CHIO-to-program, and CHIO-to-concentrator. We must ensure that no echos are lost, duplicated, or improperly delayed in any of these transitions.

The concentrator-to-program transition is easy: The only active agent is the user typing; so there is no possibility of conflicting decisions being made simultaneously. The transition occurs whenever the concentrator is echoing and a break character appears in the input stream. The CHIO-to-program transition (whenever the CHIO is echoing and a break character appears) and the program-to-CHIO transition (whenever the program tries to read and there is no input waiting) are also easy because the program is waiting for the CHIO when they occur, and thus they can be atomic actions.

The CHIO-to-concentrator transition, on the other hand, is tricky, because the CHIO can be telling the concentrator to resume echoing at the same time that the concentrator is sending off some newly typed, unechoed, characters to the CHIO. When this happens, the concentrator cannot obey the CHIO's command—it has already sent the CHIO an unknown number of characters which must be echoed before any new characters can be echoed.

When the CHIO sends a command to resume local echoing at some time t_1, it is acting on information about the state of the user's input which is derived from the input characters it has received from the concentrator. If the last character which has been received in the CHIO by time t_1 was sent by the concentrator at time t_0, the CHIO is acting in ignorance of anything which happened after t_0. The interval between t_0 and the time t_2 at which the concentrator receives the command to resume echoing will be called the *hiatus*. If any characters from the terminal are passed from the concentrator to the CHIO during the hiatus, it is not possible to resume local echoing in a straightforward way.

Local echo resumption thus requires first detecting when characters are input during the hiatus and second doing something about it. Detection requires telling the concentrator the last input character echoed from the CHIO so that the concentrator 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, i.e. the bold portions of Figure 1, during the hiatus). The numbers increment independently for each channel, and we make sure that both machines attach the same number to each character, as described below.

Now whenever the CHIO wants the concentrator to resume echoing, it sends a Request Echo Resumption **(rer)** command, together with the character sequence number **(cseq)** of the last character it received (which must have been echoed already). When the concentrator gets the **rer,** it determines whether the last character from the terminal which it has sent to the CHIO had the same **cseq.** If so, it resumes local echoing and sends the CHIO a Resume Echo **(rec)** message. If not, it continues not to echo; the characters which must have been input during the hiatus will eventually cause the CHIO to again attempt echo resumption. However, the concentrator does send the CHIO a Synchronize Character Numbers **(csync)** message with its current **cseq,** which the CHIO uses to reset its copy of **cseq** in case the **cseq**s have gotten out of sync.

The efficiency of this scheme depends upon the probability that characters are input during a hiatus. It is a good scheme if the probability is small, but poor if it is large because (a) sending extra **rer**s is wasteful, and (b) the user's response is sluggish until local echoing is resumed. The probability that the attempt to resume local echoing will fail is h/i, where h is the expected duration of the hiatus and i is the expected interval between the arrival of input characters at the CHIO during the hiatus. For our system h was about 200 milliseconds and i was at least 4 seconds, so the **rer**'s would fail less than 5 percent of the time.

It is interesting to note that i can be increased by buffering more input characters in the concentrator before sending them on to the CHIO, although it cannot be made greater than the interval between break characters. To take advantage of this increase, the concentrator must distinguish between buffered characters which have been echoed (the normal case) and those which have not. When it receives an **rer,** it must immediately echo all the characters which have been buffered but not yet echoed. Our system did not actually do this.

Another way to solve the echo resumption problem is for the concentrator to remember unechoed characters for a while after sending them to the CHIO, and to echo all the characters following the one specified by the **cseq** when it receives the **rer** command. This would get rid of the acknowledgment to the CHIO and the need to retry, at the cost of some buffering for each terminal in the concentrator—enough to cover the maximum round-trip delay in a message sent between per-terminal processes. This is quite a lot when the worst case for all the terminals is considered, because the delay can be very large if a burst of errors on the 4800-baud line forces repeated retransmissions. Our desire to minimize the amount of buffering for low-speed terminals led us to reject this method. The problem of local echo resumption has also been discussed elsewhere [5].

The foregoing analysis assumes that there are no interruptions in terminal interactions; i.e. each party waits for the other to finish, and if the user types ahead, the terminal system buffers the typing until the computer is ready to listen. While this assumption is valid most of the time, each party will occasionally wish to interrupt the other.

The user can interrupt the computer, for example, to stop a program in an infinite loop, by typing a **quit** character, which generates a special signal to the user's program. Presumably the program 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 program may also want to interrupt its user, for example, to notify him that some asynchronous event such as the printing of a file has been completed. 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 program 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 program outputs this control character, the concentrator turns off local echoing and sends the **tag** back to the program. This achieves two things. First, since local echoing was turned off, the typescript will be readable. Second, the program can synchronize with its user's concept of input because it knows that characters received after the **tag** comes back were typed after the **tag** was processed by the concentrator. In practice, the program 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 communicates with the CPU through memory which both processors can access. 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 an 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 use the same technique to send a signal requesting the wakeup of the proper user program.

The logical 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 set, speed and character structure, and the name of the process to wake up when the channel needs service.

There are three basic CPU-to-CHIO commands: *ReadString, PeekString,* and *WriteString*. These commands are issues by the supervisor in response to system calls made by a user program. *ReadString*(c, n) reads, and removes, characters from the CHIO's buffers for channel c. It stops at the first break character or the nth character, whichever is first, to ensure that the reading program won't get more input than it is prepared to deal with. *PeekString* (c, n) is identical to *ReadString,* except that the characters are not removed from the buffer. *WriteString*(c, s) writes string s into the CHIO's buffer for channel c.

Internally the CHIO has a buffer for each input and output channel. Each CHIO buffer is a list of 21-character blocks. If too many of these blocks are being used by an output channel after a *WriteString* is completed, the CHIO will return an indication that the CPU should send no more characters to this channel. When this happens, the supervisor will normally block the user program which is generating the output. When the CHIO finds that its output buffer is nearly empty, it will send the program a wakeup so that it can generate more output. Since all the buffer blocks are allocated from a common pool, the decision as to when a single channel is demanding too many of them is based on the speed of the channel and the current demand for buffer space.

This scheme, like many other features of the CPU-to-CHIO interface, requires that the CPU program be friendly. For this reason, user programs are not allowed to send commands directly to the CHIO, but must filter them through the system's supervisor, which does the necessary error checking—in this case by blocking processes which uncooperatively refuse to stop outputting when requested.

4. The Communication Link

The communication network consists of one CHIO connected to several concentrators via 4800-baud telephone lines. Characters go from the CHIO directly to the destination concentrator; there is no forwarding capability. The next few sections describe the communication link between the per-channel processes. This link consists of the processes shown in bold in Figure 1; it involves the CHIO, one concentrator, and the connecting telephone line. It is convenient to divide this link into two parts:

(1) The Error-Free Communication Link (EFCL), consisting of (a) identical modules in the CHIO and concentrator and (b) the connecting telephone line. Its function is to provide (an acceptable approximation to) 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 concentrator, 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 concentrator. However, we felt it would be best to keep the translation centralized in the CPU until we had some experience with the terminal system.

Characters in the range 0–37 octal are used internally as control characters by the terminal system. Some of these control characters, like the previously mentioned **rer,** have internal meaning to the terminal system and will be rejected by the CHIO if the CPU tries to send them. Others, like **tag,** can legally be sent by a user program but will result in some action by the system. Data characters in this range must be sent as two characters: the control character **shift,** followed by 40 plus the desired character. Thus character code 13 would be sent as **shift** 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. The **shift** characters are inserted and removed by the terminal service processes in the concentrator and by the user program in the CPU.

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. This name should not be taken too literally, of course, since the error detection and retransmission strategy we use can only reduce the frequency of uncorrected errors, not eliminate them altogether.

From the viewpoint of its users (the multiplexing and demultiplexing processes), the EFCL is a full-duplex channel which processes a character stream segmented into 13-byte messages. It does not interpret these messages in any way, except that three control characters 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.

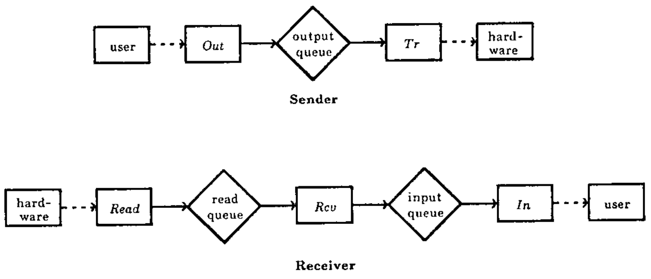
To minimize the bandwidth used for error control, only negative acknowledgments, called retransmission requests **(rtrs)**, are transmitted. A receiver sends an **rtr** whenever it receives anything other than a legal message. Messages are sequence-numbered within the EFCL. Message n always follows message $n-1$ unless the EFCL is recovering from an error. Thus the receiver always knows which message it expects next and sends an **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-acknowledgment scheme, but at the cost of more complex logic.

The timing information which makes the negative acknowledgment scheme work is provided in the following way. Each (full-duplex) EFCL contains 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 link. Note that one consequence of this arrangement is that data bytes must flow through the EFCL at the same rate in both directions, within the slop provided by the 32 windows. Dummy data bytes are supplied if necessary to balance the flow.

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 the copy can be discarded. Envelopes are sent in order, envelope $n+1$ following envelope n (mod 32), except when a retransmission occurs. The (implicit) positive acknowledgment of a message is the successful receipt from the other computer of a message with the same number, i.e. in the same envelope. Since envelopes are not explicitly identified except at the start of a retransmission, no bandwidth is used for the positive acknowledgment.

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 (dummy ones at initialization time). 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 enve-

Fig. 2. Error-free communication link.



lope buffer n contains a copy of the message which was sent in that envelope.

Free envelope buffers, which correspond to available envelopes, are kept on a *free* queue; full ones waiting for transmission are kept on the *output* queue; and full ones that have been sent but whose receipt has not been acknowledged (i.e. whose envelope has not yet come back) are kept on the *lookback* queue. The concatenation of the free, lookback, and output queues always contains all 32 envelope buffers, and buffer n is always followed in this list by buffer $n+1$ (mod 32). Input messages are stored in a different set of buffers, called in-buffers. These have no permanent numbers.

When either end of the EFCL detects an error, it resets the receiver to wait for resynchronization of the line. The sender stops what it is doing and sends a resynchronization message, followed by a request for retransmission **(rtr)** of the next envelope the receiver is expecting. The sender then transmits idle **(ign)** characters until an **rtr** arrives from the other end, at which point it sends a retransmission acknowledgement **(rta),** followed by the usual stream of envelopes, beginning with the one which was requested. In the meantime, the other end is doing the same thing. The remainder of this section describes the implementation of this scheme.

Figure 2 is an idealized picture of the EFCL's structure, in which the boxes represent processes, the dashed connections are coroutine linkages, and the diamonds are queues connecting modules which can execute in parallel. We proceed by describing each process in turn.

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.

Read takes characters from the input hardware, recognizes messages, and puts them on 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 zero bits are passed on in data bytes in the usual way; note that the byte with 8 zero bits is the **null** byte and that at least one **null** byte will result from 16 zero bits. If there are more zero bits, 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 checksummed block (15 non-null bytes after **ign**s are filtered out). If it sees such a block, *Read* puts it into an in-buffer which it appends to the read queue. If it sees anything else, it puts an error block on the read queue, throws everything away until the sychronization sequence **null syn** appears, and then starts looking for a correct block again.

Rcv is a little more complicated. It can find one of four things on the read queue: a data block (with no errors), an **rtr** (retransmission request), an **rta** (retransmission acknowledgment), or an error block (anything else, but most likely a block with a bad checksum). *Rcv* can be in one of three states: expecting a data block, expecting an **rta,** or waiting for an **rtr.** *Rcv* takes an in-buffer from the front of the read queue. Then:

(1) If *Rcv* expects and gets a data block, it (a) puts the in-buffer on the end of the input queue and (b) moves the envelope buffer on the front of the lookback queue to the end of the output free queue, thus recording the positive acknowledgment that the corresponding envelope has been received at the other end of the link. *Rcv* will still expect data blocks.

(2) If *Rcv* expects and gets an **rta** for envelope n_{in} (for which it had previously sent out an **rtr** in case (4), *Rcv* expects data block n_{in}.

(3) If *Rcv* is waiting for an **rtr,** it discards anything else. When an **rtr** arrives for block n_{out}, an **rta** for envelope n_{out} is appended to the output queue. Buffer n_{out} must be on the lookback queue; it and all its successors on the queue are appended to the output queue, which is initially empty, except perhaps for the **rtr** put on by case (4). *Rcv* then expects an **rta** for envelope n_{in}, which was determined in case (4). Note that an **rtr** on the read queue will always be preceded by an error block; so the actions of case (4) are always taken just before those of this step.

(4) Otherwise *Rcv* deletes any **rtr**s or **rta**s on the

output queue and moves any envelope buffers to the end of the lookback queue. It puts a synchronization block (see *Read*) on the now empty output queue, which will force the *Read* at the other end of the link to synchronize. Then it generates an **rtr** for the envelope n_{in} whose buffer is on the front of the lookback queue, or on the front of the free queue if the lookback queue is empty, and puts this on the output queue. *Rcv* now expects an **rtr.**

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.

Finally, we clear up a loose end. The scheme just described 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. This timer is turned off when an **rta** is received. If the timer goes off first, however, *Rcv* is forced into case (4), thus ensuring a new attempt to resume normal communication.

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 two block times plus a round-trip time on the telephone line, or about 100 milliseconds. Since available telephone lines and modems promise no more than 1 error per 10^5 bits, we can expect one error every 20 seconds on a 4800-baud line, or an efficiency of 99.5 percent.

4.2 Multiplexing

The choice of methods for converting the single EFCL channel into a channel for each terminal is dominated by the demands placed on two scarce resources: bandwidth on the EFCL and buffer space in the CHIO and the concentrator. Input multiplexing is fairly easy to handle because the volume of input is low, and the CHIO has a large amount of buffer space which can be shared among the terminals attached to all the concentrators. Output is hard because the volume is large, bandwidth must be shared equitably, delays in starting output to any terminal must be short, and the buffering done in the concentrator 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. Futhermore, the maximum interval between break characters is short (about 150 characters), and since the concentrator 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 must 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 do not want to have much buffering in the concentrator. 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 concentrator 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 consists of a burst marker **(bm),** a device number, and a sequence of input characters terminated by the next **bm.** 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 bursts of several characters can be sent even for low-speed devices. A **bm** immediately followed by another **bm** serves as an idle message if there are no data to send.

4.2.1 The Meta-Multiplexing Algorithm. The output multiplexing algorithm is based on the simple fact that output messages tend to be long. Another way to say this is that the set of *active* channels, on which output is in progress, changes slowly relative to the rate at which characters can be transmitted along these channels. For simplicity, we begin by considering an interval of time during which this set doesn't change at all. If sender and receiver agree on which channels are transmitting and on the order in which they will share the channel, we don't have to transmit any multiplexing information at all. This is a form of time-division multiplexing, in which the rules for allocating time slots may be arbitrarily complex.

We can split the multiplexer and demultiplexer each into two modules, a *Meta-Multiplexer* and a *Bandwidth Allocator* in the sender, and a *Meta-Demultiplexer* and an identical allocator in the receiver. The meta-algorithm requires the allocator to determine which channel owns each character position in the EFCL stream. The sole constraint on the algorithm used by the allocator is that is must reference only state information which exists in both sender and receiver, and which has the same value in the sender when the sender is processing the *i*th character in the EFCL stream as it does in the receiver when the receiver is processing that character.

The role of the meta-algorithm is to deliver each character to the correct channel, and it will do this as long as the Bandwidth Allocator operates identically in the sender and the receiver. Thus the meta-algorithm is independent of the allocator in the following sense. The role of the allocator is to assign the proper amount of bandwidth to each channel. If it does this improperly, the channels may get too much or too little bandwidth, but the multiplexing will still be correct; i.e. every character will still be delivered to the channel on which it was sent.

In reality, of course, the set of active output channels is not fixed, but the scheme can be easily extended to the more general case of a slowly varying set of active channels. If the Meta-Multiplexer wants to add a new active channel, it adds the channel to the set of active channels and 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** message, adds the channel to its set of active channels and otherwise ignores the message.

Similarly, when the Meta-Multiplexer finds that there are no more characters for an active channel, it deactivates the channel by removing it from the active set and sending a Delete Old Channel **(doc)** character. The Meta-Demultiplexer, when it gets the **doc,** likewise removes the channel from its active set, but otherwise ignores the message. 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 multiplexed stream which are data characters. The efficiency of our algorithm is $n/(n+3)$, where n is the average number of characters sent to a channel between an activation and the next deactivation; usually this is just the length of an output message from the user's program. The 3 is the number of control characters added to the stream to activate and deactivate the channel. For example, if n is 22 characters, the efficiency of the multiplexer is 88 percent.

4.2.2 The Bandwidth Allocator. This section describes the Bandwidth Allocator used in our system. It was designed around three criteria:

(1) It must have the properties demanded by the meta-algorithm. These properties are implied by the requirement that identical copies of the allocator 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 makes things tricky.

Time is arbitrarily divided into intervals of t seconds over which the flow of characters is averaged; in our system t was 0.1 seconds. For each output channel we keep the number of characters r which that channel's terminal can accept in t seconds. Let ir be the integer part of r and fr the fractional part. For example, a channel driving a 10 cps terminal has $r = 10t$, and one driving an IBM 2741 has $r = 14.8t$. We also keep the number of characters c to be sent in the current interval; again let ic be the integer part of c and fc the fractional part. Initially c is set to zero. The basic idea is to alternately send ir and $ir + 1$ characters in each interval in such a way that the average number of characters per interval will be just r. The following algorithm will give ic the values ir and $ir + 1$ with the proper distribution.

At the beginning of each interval, we set $c = fc + r$ for each channel. The Bandwidth Allocator will then try to send ic characters to the channel in that interval. It does this in two passes. In the first pass it sends each active channel $\min(d, ic)$ characters. The choice of d determines the rate to which high-speed devices are restricted when there is not enough bandwidth to serve everyone. The allocator will finish this pass even if it has to stretch the interval beyond t seconds.

If there is still time left in the interval after the first pass, channels that can accept more than d characters are sent $ic - d$ characters until the interval is over. 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. Channels get this extra service in round-robin fashion, but of course not more than once per interval.

After the output has been generated for each interval as described above, one or more Check Synchronization **(chs)** control characters are inserted to fill out the interval. This checks and resets the synchronization of the Multiplexer and Demultiplexer (in theory, loss of synchronization would only occur if the EFCL failed to detect an error) and provides padding if there is no output to do.

5. The Concentrator

The concentrator was designed to:

– Efficiently handle input and output to a large number of low-speed (up to 300-baud) devices;
– Provide flexibility, especially in interfacing with a variety of devices;

- Be controllable from the CPU so that operator intervention is not required except in case of hardware malfunction.

The concentrator is implemented by a small computer which has specialized read-only microcode to implement the EFCL algorithm, multiplexing, and bit-scanning for low-speed devices. The rest of the work is done by a collection of *tasks* coded in the machine's assembly language and scheduled by a simple priority scheduler.

Low-speed devices are bit scanned by microcode. The assembled character is echoed if appropriate and then stored in the input buffer for the device. However, if the input buffer is full, which might happen either because of a communication line malfunction or because of an unusually heavy load on the Multiplexer, the character is neither stored nor echoed. Thus the user does not get false feedback if his character was lost by the concentrator. The Multiplexer removes characters from the input buffer and multiplexes them for transmission to the CHIO when requested to do so by the EFCL. Output is similar to input: the Demultiplexer puts characters in the device's output buffer, from which they are later removed by the microcoded output bit scanner. Recall that the Bandwidth Allocator design ensures that characters will not be delivered faster than the output bit scanner can dispose of them.

Line printers, card readers, and other devices whose speed is too high for the bit scanner are handled differently on the device side of the concentrator. A device-specific task inputs characters from these devices and stores them in the input buffer for the Multiplexer to pick up. Similarly, for each output device, a task gets characters from the device'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. In addition to interfacing with devices that are not bit scanned, tasks are used for answering the phone, initializing the concentrator, and buffer allocation. Some of these are done in conjunction with a controlling CPU process, using one of the channels for communication.

The third function of the concentrator is initialization from the CPU. This is slightly tricky because we would like the initialization to work regardless of the state of the concentrator. Since it is possible for the concentrator to turn off the hardware interface to the 4800-baud line, or to get into a microcode loop, this is not entirely practical, but we do quite well at the expense of putting a glitch into the EFCL. Initialization proceeds in four steps.

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

Second, whenever the EFCL microcode is about to read an input character from the hardware, it checks for the control character **init.** If it gets three 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 EFCL and the Demultiplexer to operate, albeit in a rudimentary way. It also turns off all tasks, because they cannot be expected to run properly until their programs and data have been loaded.

Now the CPU can load the concentrator's memory by sending special messages consisting of a Load Remote Concentrator **(lrc)** control character followed by loading information. Part of the loading information is a flag that indicates whether tasks should be allowed to run. Thus tasks can remain turned off until memory has been properly set up and then turned on with a last **lrc** so that a just loaded initialization task can run. Finally, this task can interact with the CPU to complete the initialization.

6. Some Facts

The concentrator contains about 500 microinstructions, each 82 bits wide, which implement the EFCL, multiplexing, low-speed device service, task scheduling, and an emulator for a standard minicomputer instruction set. The CHIO has about 900 microinstructions. The microcode for the EFCL and the Bandwidth Allocator is identical in both machines. The concentrator has less microcode than the CHIO because great effort was expended in minimizing the concentrator microcode so as to reduce the cost of replicating the machine. This was less important with the CHIO, where efficiency and straightforwardness took precedence.

The system was implemented and run in an experimental mode before the demise of BCC. Since then it has become fully operational on the BCC-500 at the University of Hawaii, although in a much smaller configuration than was envisioned in the initial design.

Received April 1974; revised September 1975

1. Carr, C.S., et al. Host-host communication protocol in the ARPA Network. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 589–597.
2. Crocker, S.D., et al. Function-oriented protocols for the ARPA computer network. Proc. AFIPS 1970 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 271–279.
3. Heart, F.E., et al. The interface message processor for the ARPA computer network. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 551–567.
4. Ornstein, S., et al., The terminal IMP for the ARPA computer network. Proc. AFIPS 1970 SJCC, Vol. 40, AFIPS Press, Montvale, N.J., pp. 243–254.
5. Tymes, L. Tymnet: A terminal oriented communication network. Proc. AFIPS 1971 SJCC, AFIPS Press, Montvale, N.J., pp. 211–216.