Software for Multicomputer Communications

J. W. Layland
Communications Systems Research Section

This article contains a brief discussion of the alternatives which exist for software support of intercomputer communications in a multiple minicomputer network, and a detailed description of a software package which implements a relatively flexible option on the Xerox 910/920/930 and Sigma computers. The material should be applicable to the intercomputer communication needs of the tracking stations as the number of subsystem control minicomputers within these stations increases in the near future.

I. Introduction

The tracking stations of the Deep Space Network currently use a few computers for handling telemetry and command data streams, for control of the large antenna, and for monitoring various performance measures of the station. It is expected that computerized control and monitoring of station functions will greatly increase in the near future in order to generally improve maintainability, and to increase the fraction of time that the station is available for spacecraft tracking. Given current trends, this increased computerization will likely be accomplished through a number of small computers, each of which is interfaced to some specific subset of the tracking station equipment. Each of these subsystem computers would be interfaced in some fashion with the other computers in order to enhance the coordinated operation of the station, and to enable the pooling of certain resources, such as computer unit-record peripherals. Multiple minicomputer networks have been used successfully in industry in a variety of process control and monitoring applications. A similar multiple minicomputer network is under development as the DSN Network Control System (NCS), wherein each of the minicomputers performs a dedicated monitor/control function for the network operation as a whole. In the NCS, the separation between functional entities effects a well-defined physical interface between software development teams for each of the functions. This separation has been viewed as a means to increasing software reliability and easing some of the problems inherent in developing large multi-function systems. The communication paths which exist between the minicomputers must support the interface between the various functions of the system, hence the organization of these paths, and the associated features, represents a significant item in the overall system design.

A previous article (Ref. 1) described one approach to the capabilities and features of an intercomputer communication structure usable in these applications, and a second article (Ref. 2) is a detailed description of our "twin-coax" intercomputer communications hardware and interfaces. Within this framework, a large amount of flexibility still exists for communication modes to be supported by the software, and the overall system organization. Section II of this article contains a brief discussion of the alternatives which exist in software support of intercomputer communications; succeeding sections contain a detailed description of the software which implements one of the more complex and flexible options on our Xerox 910/930 and Sigma computers.

II. Features and Capabilities

The communications paths which are established between computers in a multicomputer structure have one primary reason for their existence: to provide distributed access to resources and/or data which are maintained as a private or localized resource for reasons of economy, reliability, or policy. In a tracking station, examples of such resources could include the current-exact antenna position, the most recent estimate of the telemetry downlink signal-to-noise ratio, the floating-point arithmetic capability needed to evaluate predict polynomials, data-storage devices, and computer input/output devices. One of the more important considerations in defining capabilities is that the overall system must be robust and tolerant of both data errors in messages and failures inside any one of the subsystems. A straightforward system organization, and one which can be the most resistant to propagation of errors is a simple hierarchy: Each subsystem minicomputer "owns" a small segment of the overall system, and performs all control, monitoring, and data handling functions associated with that segment of the system. Groups of the subsystem minicomputers are in turn owned by a supervision-type minicomputer, and so on. Capabilities which are needed by a subsystem are mostly contained within the subsystem minicomputer, and those capabilities not contained are provided by a higher level of the hierarchy. Physical communication paths exist only along vertical lines of the hierarchy. The simplicity of such an organization limits the propagation of failures through the system, makes it relatively easy to intelligently allocate resources, and restricts the growth of complexity in the software developed to support the intercomputer communications paths.

This simplicity also restricts flexibility for changes and for recovery from certain types of computer failures. The greatest flexibility would be obtained by providing a universal communication bus which allowed any computer to communicate with any other computer and/or any item of system hardware. Such flexibility enables "instant" failure recovery but also enables extensive error propagation through the system. It also appears that it would be quite difficult to build. Our overall approach (Ref. 1) utilizes a functional equivalent of the universal bus between computers, but retains "ownership" of subsystem hardware by the subsystem control computer. Logical communication paths would be established on this bus for current operations, and revised as needed for mission changes. Intercomputer communications software could be organized into a simple hierarchy using this bus, or the software could be made transparent to the bus organization, retaining its flexibility and complexity for the "applications-level" software. In the software package to be described later, the logical communication paths are not "built in" the software, but are established dynamically during program execution.

During initialization and system start-up, these communication paths would carry programs and operating data bases from a pooled storage facility to each of the subsystem control computers, and, in response to a failure in any part, additional diagnostic programs could be transferred from the same facility to the one affected unit or units. Activation and deactivation of various program parts in each subsystem would also occur in response to messages transmitted on these communication paths.

During normal operation, data describing the current status of parts of the system would be transferred from the subsystems which measured or computed those data to other subsystems which need them. Assuming a hierarchically organized system, such communication would only occur along the lines of authority within the system.

During software and system development, yet a third mode of operation occurs, where a subsystem control computer has some, but not all, of the resources needed for software development. The software loaded into the subsystem computer from the central repository is in this case the assembler/compiler for that machine's language. When this program is activated, the computer communications paths provide it access to three or more distinct and independent devices which exist as peripherals to one or more of the other computers, or which are emulated through software or data storage devices on other computers. This is closely analogous to a "remote batch" operation, except that here the computing element is local to the subsystem engineer's interest, and the unit-record peripherals are remote.

An intercomputer communications driver package has been developed which appears in principle to be capable of performing in any of these operational modes. Complete error control logic is included for data errors and errors in addressing of messages. Some features are provided, such as data type identification, and multiple logical paths between any computers, for which as yet only a hypothesized need exists—in the belief that such features were easier to include in the initial package, and then remove if not needed, than they would be to add at a later date.

III. Link-Network Software Package

The communications software package consists of a collection of interrelated subroutines which provide the direct interface between a subsystem's main, or "applications-level" program, and the intercomputer communications hardware which physically connects the computers. In addition, this package provides error detection and control logic, message formatting, buffer management, and a variety of lesser services for the calling program. The computer link-network is organized conceptually as in Fig. 1, although the physical organization may differ. Each minicomputer has a unique identifying number which is used to address messages to it from any other minicomputer through the switch "X." Up to seven distinct bi-directional logical channels may exist between any pair of computers, as determined during program execution.

Table 1 lists the subroutine entry calling sequences for the two currently existing implementations of the link-network communications package. The principal entry points are those for establishing read-requests, or requests to receive data; for writing data along a path to a specified destination computer on any of the logical channels; and for checking the completion of prior requests. Entry points in the lower half of Table 1 provide for general management of the communications paths. A description of these subroutine calling sequences and their effect appears in Section V, together with a description of the tables which must appear in the calling program. The descriptions which follow in Sections IV to VI are in sufficient detail to serve as a manual for the user of the link-network package, and may be skipped by the casual reader.

IV. Message Protocol

The adopted message format is shown in Fig. 2. It is a close derivative of the formats of the IBM Multileaving protocol, with the addition of the two address bytes. The Multileaving terminology is used to identify the other overhead bytes, where applicable.

Both address bytes are of the form[1] X'CO' + n, where n is the computer identity number, in the range from 1 to 63. They serve to uniquely identify a communication path from computer "SRC" to computer "DST." The reverse path is denoted by a reversal of the address codes, and is always activated simultaneous with the forward path. The Block Control Byte (BCB) is of the form X'CO' + m, where m is a 4-bit block sequence counter. Blocks are numbered sequentially on any SRC/DST path-pair, and are transmitted alternately in a half-duplex fashion. Along any path-pair, the lower computer identity number designates the "master," which receives only even-numbered blocks, and sends only odd-numbered blocks. An alternate BCB form of X'FO' is used to initialize the communication path, and a BCB of X'FF' is used to terminate. Either end of a path-pair may initiate communication, but both must send and receive the initialize BCB before data communication can begin.

The Record Control Byte (RCB) and Functional Control Sequence (FCS) bytes serve to divide each path-pair into seven logically independent duplex channels. The RCB also identifies the type of message block. Each bit of the FCS signifies, when set to 1, that data may be transmitted on the corresponding logical channel. It is reset to 0 when the requested data have been received, and set to 1 when data are again requested. For example, an FCS value of X'40' signifies a data request on channel 1, and an FCS of X'31' signifies data requests on channels 2, 3, and 7. The FCS value of X'80' is used in terminating. The format of the RCB is shown in Fig. 3. The least-significant three bits of the byte identify the logical channel to which this message belongs. A type code of B'10' indicates that the message block contains data, and that the String Control Byte (SCB) contains the total byte count for the block, including overhead bytes. For data blocks, $B_1 = 1$ indicates binary (non-text) mode data, $B_2 = 1$ indicates string-compressed data, and $B_3 = 1$ indicates machine-dependent data format. $B_1 = B_3 = 0$ indicates American Standard Code for Information Interchange (ASCII) text. String-compressed data are not supported by existing software, but could be implemented within user programs. A type code of B'01' or B'00' indicates that the message block is a signal or channel operation which requires user-program action. The total message package is the six overhead bytes. Bits B_1, B_2, and B_3 and the SCB specify the signal value. No specific response to the signals is

[1] In the following, X'yy' means hexadecimal constant yy, and B'yy' means binary constant yy.

implemented within the communications software, but conventions should be established for specific signals to implement device/file positioning operations. A feasible set appears as Table 2. A type code of B'11' indicates a null or idle message block, which may appear during initiation, termination, or to maintain synchronism when no data are available to be transmitted on channels with active requests.

Data transferred through the link are maintained error-free and in-sequence as far as possible. Any message blocks that are received in error, are truncated, are out of sequence, or have been misrouted are discarded by the receiving error control logic. Message blocks which have been transmitted in error, or for which no reply block is received within a pre-determined time period, are retransmitted.

V. User/Software Interface

This section describes the interface between a user or applications-level program for the link-network communications package, and that package. The description is based upon the 910/930 implementation of the link software. Variations in details for the Sigma 5 implementation will be described at the end of this section.

The user program contains several tables which define the structure and status of the multicomputer network. The first of these is the Link Table (LNKTAB), which is illustrated in Fig. 4. It contains one word for each computer, which is defined to the link-network package. The most significant byte of each word contains a text character, which is the software name for the associated computer. The less significant bytes contain a pointer to the Link Control Block (LCB) for the associated computer, or they may contain zero if no LCB exists. The software name characters have been assigned sequentially starting with 'A' to facilitate table operations. The current assignments appear in Fig. 4. The LNKTAB must be set up by the user programs. The user program must also provide an address word labeled "MYNAME" which contains that computer's link name in the form X'CO' + *n* in the second most significant byte.

The Link Control Block contains the data used by the link-network package to control the transfer of data along the associated path-pair and maintain synchronism between the attached user programs. The format of the LCB is shown in Fig. 5. The 29 words for each LCB are provided within the user program. The first word, the LCB chain word, is pointed to by the appropriate LNKTAB entry, and by the chain of the next lower LCB, and points in turn to the next higher LCB, or contains -1. The LCB chain word is set up by the user program; the remainder of the LCB is set up by a call to IOCLEAR. All LCBs connected to LINKTAB plus lower-level routines are initialized by calling IOCLEAR with the A-register = 0. Only the specified LCB is initialized if the A-register contains a pointer word from LNKTAB when IOCLEAR is called.

The second word of the LCB defines the current state of operation along the path-pair, and will be described in some detail in Section VI. State includes the block sequence counter, read/write mode flags, initialization flags, the error-retry counter, etc. CURBLK is a pointer to the current or most recent past transmitted block, and is -1 if said block is undefined. TIMER is the clock-count cell used to recognize lost data and is negative when ticking. The fifth through ninth words contain flags which define the state of each of the seven logical channels; they are in the same format as the FCS byte of the transmitted data block. The RDFLG and HSRDFLG, in fact, contain the next FCS to be sent, and the most recent FCS received in their most significant byte. The write-stack (WRTST$) contains a queue of data blocks which have been enabled for transmission by read-requests from the incoming FCS. The channel-write-buffer pointers (CHNWB) contain addresses of the current or most recent data blocks associated with each logical channel. If transmission is not enabled (the write is blocked or is complete), the data block address will only appear in the CHNWB. The remainder of the LCB represents a shortened dummy data block which is used for initialization, termination, and idling messages on the associated path.

Data to be transmitted by the link-network package are contained in buffers with content-specifying header. Identical buffers are used to receive data. The general format of these buffers is shown in Table 3. The QCHAIN word is used to link the buffer into FIFO queues, as needed in handling. OWNER contains the software name, computer path plus channel, to which the data buffer belongs. BUFSZE is the available data space within the buffer block, in words. Oversize incoming data blocks will be truncated to this value. WCW contains the number of words from the data portion of the buffer block which should be transmitted, and is meaningful only when data in the buffer block are being prepared for transmission.

AWCW and STATW are meaningful only after data have been transferred in or out of the buffer block. AWCW contains the number of words actually transferred, and should be equal to WCW for transmission, and

less than or equal to BUFSZE for receiving. STATW contains the link hardware cumulative status at the end of I/O transfers. The remainder of the buffer block contains the data words, including overhead bytes, actually transferred on the links. A subroutine IOBUFSET is provided to carve up an arbitrarily sized storage area into preformatted buffer blocks with space for 132 data bytes. The beginning and ending addresses of the buffer storage area are specified to IOBUFSET in the A- and B-registers, respectively. Three of these buffers are immediately transferred to a low-level subroutine for incoming data. The remainder are queued in a buffer pool. Buffer blocks are placed into this pool by calls to BUFPUT, and removed from it by calls to BUFGET with the A-register containing the buffer block address. Return from BUFPUT is to the calling-address + 1. Return from BUFGET is call + 2 if successful, and call + 1 if no buffer is available. It is a user-program responsibility to avoid losing buffers and emptying the buffer pool.

Data transfer between user programs and the link-network package is handled by five subroutines, with auxiliary services handled by three others. Arguments are transferred to these subroutines in the A- and B-registers, with the A-register identifying the intended computer/channel pair by its software link name, and the B-register pointing to a buffer block, if needed. The software link names are composed of the channel number in the least-significant byte, and the text character identifying the destination computer in the next least-significant byte. Return from the subroutines is to the call-location + 2 if the requested operation is accepted, and to call + 1 if the request was invalid or rejected. With minor exception to be noted, buffer blocks are returned to the user program via pointers in the A-register. End-action for any previously completed transfers is performed on entry to these routines.

IOSTART requests the start of operation of the identified logical channel on the specified path. If it is the first channel on the specified path to be started, a path initialization is also performed. IOSTOP terminates operation on the identified logical channel. If this is the last active channel on the specified path, that path is terminated. IOSWAIT is a convenience routine which rejects all calls until the opposite end of the path-pair has been initialized.

IOGET establishes a read-request for the identified logical channel, which is subsequently communicated via the FCS to the opposite end of the path-pair. IOPUT attaches the buffer block in the B-register to the identified logical channel and prepares to transmit the data contained in that block along the appropriate path. The identified channel is immediately set Busy. The transmission is blocked until a corresponding read-request has been received from the opposite end of the path-pair. IOSIGNL transfers the signal bytes in the B-register into an available buffer from the pool, and then proceeds similarly to IOPUT. Any transfer request on an already busy channel is rejected.

IOCHEKR is called to test the status and availability of the incoming half of the identified logical channel. A returned value of A = -8 occurs if the channel has not been started. The value A = 0 occurs if there is no completed incoming data transfer. If a data block has been received for the identified channel, it is returned via an A-register pointer. If a signal has been received, the A-register returns the identified channel's software-link name with the addition of the bit-flag X'08' in the most-significant byte. The signal value is returned in the B-register.

IOCHEKW is called to test the status and availability of the outgoing half of the identified logical channel. A returned value of A = -8 occurs if the channel is not busy, and no read-request has been received from the opposite end of the path-pair. The value A = 0 occurs if the channel is not busy, but an active read-request has been received. The value A = -1 occurs if the channel is busy and the transmission is not complete. If the channel is busy and the transmission is complete, the channel is set to not busy, and the current buffer-block pointer (or signal value) is returned in the A-register together with a completion flag of X'08' in the most-significant byte.

A summary of subroutine calls for the 910/930 appears in Table 1, together with a rudimentary description of its action. The primary reason for the unpleasant complexity which has appeared here is the need to maintain synchronism over the independent logical channels between user programs at the opposite ends of the path-pair.

The principal difference between the link-network software package described above for the 910/930 and the corresponding package for the Sigma 5 is that the latter is embedded within a semi-permanent operating system, whereas the former is a part of the transient user program. In the Sigma 5, each defined path-pair is

considered as a re-usable resource which may be transiently owned by a user program.

The link software is accessed via the system-call instructions CAL3,12 N from either a background program or a terminal-user program. The LNKTAB and the defined LCBs are a part of the resident monitor, as are a minimal set of buffer blocks through which the actual I/O is done. Data are exchanged between the monitor buffers and corresponding user-space buffers via the IO calls. Table 1 also lists the link package calls for the Sigma 5. Two new routines have been added to manage the "ownership" of the link paths, and several routines which were accessible in the 910/930 have been subverted to these routines, or to monitor initialization. ATTACH establishes ownership of a specified path, if it is not already owned by another active job, and RELEASE frees the specified path from ownership by the current job. All owned paths are released at the end of any job step. Arguments are transferred via Register 8, corresponding to the 910 A-register, and Register 9, corresponding to the 910 B-register. Requests for operation on a path which has been terminated cause the job to be aborted. All calls, arguments, and actions are basically the same as in the 910/930 version. The principal change to be noted is that Register 9 must contain a buffer-block address within the users' area when IOCHEKR is called, to provide a place for the received data to be transferred from the monitor's buffer block.

VI. Software Internal Structure

The link-network software package has been designed to provide an elastic interface between the user-level programs which communicate through it, and the time-and-event-driven realities of the physical link communication signals. Three distinct levels of operation exist within this software. Each level consists of a collection of complete non-interrupted processes. The interface between the asynchronous processes at adjacent levels is effected via a limited set of queues and flags. Each single process was implemented using a "Structured Programming" approach (Ref. 3) to increase the understandability during design, and during possible later modifications. The interface between asynchronous processes was specified using finite state machine representations for the cause and effect relationships which had to be defined (Ref. 4). Structured programming was avoided for multiple asynchronous processes because the structuring introduces more irrelevant complexity than it removes. Requests for services by a lower level are best regarded as primitive within the upper level. The actual performance of those services proceeds in parallel with operations on the upper level, and only the events of request and completion are relevant to the progress of the upper level.

A change-of-state of the bi-level interfaces actually occurs when the lower-level process has acted upon a service request from the higher level. The top-most software level was described in terms of its calling or user interface in Section V. These calls serve to transfer the care of a data buffer between the calling user process and the queues which interface to the single intermediate-level process. One such process exists for each path-pair, and its total state is contained within the Link Control Block. The format of the state word is shown in Fig. 6. The contents of this word are altered when a block has been sent or received over the path-pair. The block number contains the block-sequence number associated with the current transmit block. The LCB is in "write mode" if a block is actively being transmitted; it is in "read mode" if a block is to be transmitted to it; or it may be in "no-mode" if there is no data to send, an idle message has just been sent, and an idle message has just been received from the opposite end. The bad-block number and associated flag are used to facilitate recovery, if possible, if a totally out-of-sequence block is received. The transmission-retry counter allows seven repeated attempts to transmit over a given path before it is declared inoperative. The path may also be declared inoperative through a terminate operation, or on receipt of a terminate message. The full-initialize bit must be set before data transfer can occur, and can only be set after initialize messages have been both received and sent on the path. It is reset when the path is declared inoperative.

The intermediate-level process functions as an end-action or "CLEANUP" operation for any messages which have been transferred over the communication paths. It interacts with the user-level interface routines via the data placed in the LCB. It requests services from the next lower-level process via calls to interface subroutines L9READ and L9WRITE, and obtains completely transferred blocks from the queues by calls to L9GET. The A-register (Register 8 for Sigma 5) for these calls contains a pointer to a data buffer block of the format described in Section V. If the link hardware is not active after the buffer in the read/write calls has been properly queued, it is activated by the software. Transfer of the data is then under control of signals from the link hardware. Whenever link hardware activity terminates with completion of data transmission for a given block, the hardware is reactivated for any block then contained in the queues of the lowest-level software process. It should be noted that the hardware can override the direction of an activation kick from the software whenever the opposite end of the path is simultaneously activating.

VII. Status and Plans

The software described herein is currently operable on three computers: the Sigma 5, Xerox 910 and 930. This small network is centered by the Sigma, which has an expandable four-way multiplexer allowing it to selectively communicate with up to four other computers. Only two machines are currently defined to the Sigma operating system, and only the 910 and 930 are currently connected. The 910 computer is interfaced via the GCF TTY lines to Goldstone, and it and the Sigma will be used with the software described here to monitor and control an automated pulsar track at the Venus Tracking Station. These two computers were previously used with a simpler intercomputer communications structure in the monitoring of Spacecraft Ranging Operations with Mariner 1971 (Ref. 5).

Several additions to this link-network structure are planned for the near future. In the software area three things will be done: the designed support for ASCII as a common-denominator text-data type will be implemented within the Sigma system and made available as part of the 910/930 package. The CLEANUP process within the Sigma 5 will be integrated with the job scheduler of the Sigma operating system so that Sigma jobs can be initiated from the opposite end of the link. Store-and-Forward logic will be added to the Sigma CLEANUP process to allow communication between the other machines without requiring an active job within the Sigma. It is also expected that the link-network software package will be translated to execute on other computer types, principally the DEC PDP-11, and the MODCOMP-II. Hardware interfaces to the twin-coax intercomputer communication links and the L9READ/WRITE level of software already exists for the PDP-11 (Ref. 2). Hardware twin-coax link interfaces for the MODCOMP-II will be implemented using the JPL-DSN 14-line interface (Ref. 6).

The entire software package occupies 1300 words on the 910/930, and slightly more on the Sigma 5 because of the complexity of the monitor interface. The required tables and buffer blocks are not included in this figure. Slightly over 1000 of the instructions are associated with the link-network structure and are independent of the physical intercomputer communications hardware. The remaining instructions represent the lowest-level hardware-driven process, and as such is tied to the use of the twin-coax links. The link-network software package is unpleasantly large, much larger than expected when initially defined in function, and as a result, a desideratum under consideration is the shrinking of this software package by limiting communication to one logical channel and one computer path for some application in the 910.

References

1 Layland, J. W., and Lushbaugh, W., "A Multicomputer Communications System," in *The Deep Space Network Progress Report*, Technical Report 32-1526, Vol. XII, pp. 195–199, Jet Propulsion Laboratory, Pasadena, Calif., Dec. 15, 1972.

2. Lushbaugh, W. A., "A Driver/Receiver Unit for an Intercomputer Communications Link," in *The Deep Space Network Progress Report*, Technical Report 32-1526, Vol. XV, pp. 109–115, Jet Propulsion Laboratory, Pasadena, Calif., June 15, 1973.

3. Dijkstra, E. W., "Structured Programming," in *Software Engineering Techniques*, pp. 83–93, NATO Science Committee, 1969.

4. Holt, A. W., et al., *Data Structure Theory and Techniques*, pp. 16–62, Final Report to Rome Air Development Center, Contract AF30(602)-4211, Task 459403, Applied Data Research, Inc., Princeton, N. J., 1969.

5. Erickson, D. E., and Layland, J. W., "An Experiment in Remote Monitoring of Mu-Ranging Operation at Mariner Mars 1971 Superior Conjunction," in *The Deep Space Network Progress Report*, Technical Report 32-1526, Vol. XV, pp. 156–166, Jet Propulsion Laboratory, Pasadena, Calif., June 15, 1973.

6. *Detail Specifications for Deep Space Network Control System, Standard Interface*, JPL Specification ES508534, 1974 (JPL internal document).

Table 1. Summary of subroutine calls

Name	Sigma CAL3,12	Arguments		Skip[a] return?	Response
		A-register (R8)	B-register (R9)		
IOGET	6	PATH/CHANNEL	–	Y	Establishes read-request
IOPUT	10	PATH/CHANNEL	BUFFER POINTER	Y	Queues buffer for writing
IOSIGNL	11	PATH/CHANNEL	SIGNAL	Y	Queues signal for writing
IOCHEKR	8	PATH/CHANNEL	BUFFER POINTER[b]	Y	A = −8, not available
					A = 0, available
					A > 0, buffer complete: B = signal value
IOCHEKW	12	PATH/CHANNEL	–	Y	A = −8, not available
					A = −1, busy
					A = 0, available
					A > 0, A = buffer pointer
IOSTART	4	PATH/CHANNEL	–	Y	Open specified channel
IOSTOP	5	PATH/CHANNEL	–	Y	Close specified channel
IOBUFSET	N/A	START	END OF SPACE	N	Initialize buffer space
IOCLEAR	N/A	0	–	N	Full initialization of all LCBs and system
		LCB-POINTER		Y	Clear specified LCB
BUFGET	N/A	–	–		A = system buffer pointer
BUFPUT	N/A	BUFFER POINTER	–	N	Returns buffer to pool
ATTACH[b]	0	PATH/CHN = 1	–	–	Claims job ownership of path
RELEASE[b]	1	PATH/CHN = 1	–	–	Releases job ownership of path

[a]910/930 only.
[b]Sigma 5 only.

Table 2. Feasible signal conventions

Operation	T	B_1	B_2	B_3	Channel	SCB
OPEN	01	0	0	0	#	I.D. Byte
CLOSE*	01	0	0	1	#	0000 0001
END FILE*	01	0	0	1	#	0000 0010
REWIND*	01	0	0	1	#	0000 0100
undefined*	01	0	0	1	#	xxxx x000
POSITION	01	0	1	0	#	I.D. Byte
SKIP	01	1	F	R	#	Number
unassigned	00	X	X	X	#	Any

* Starred operations may merge

F Indicates *File* operation if 1, record if 0

R Indicates *Reverse* direction if 1, forward if 0

Table 3. Buffer block format

Label	Contents
QCHAIN	Block Chaining Link
OWNER	Channel/Path ID
BUFSZE	Size of buffer, words (44)
WCW	Number of words to send
AWCW	Number of words sent/received
STATW	Hardware status at termination
DWD1	Data
*	to
*	send/receive

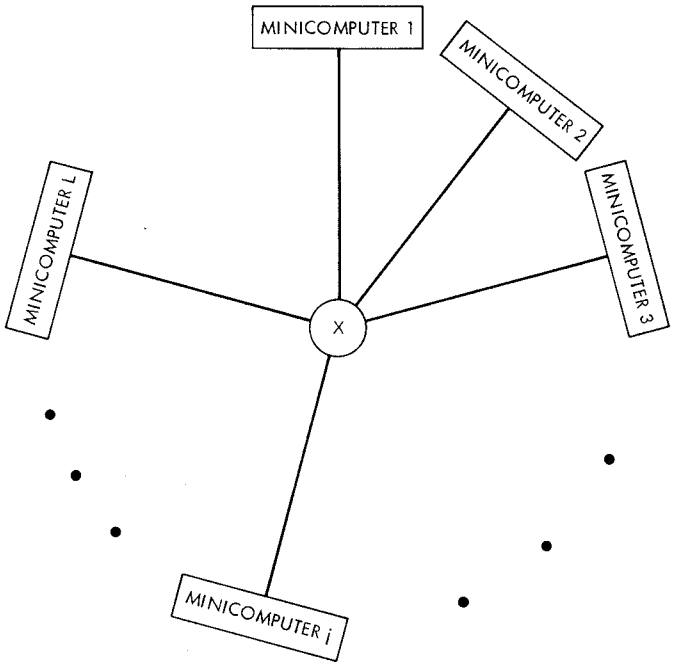
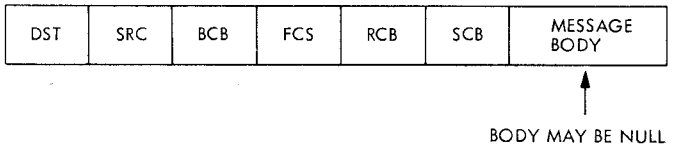


Fig. 1. Link network structure



DST = DESTINATION ADDRESS
SRC = SOURCE ADDRESS
BCB = BLOCK CONTROL BYTE (BLOCK SEQUENCE NUMBER)
FCS = FUNCTION CONTROL SEQUENCE (READ FLAGS)
RCB = RECORD CONTROL BYTE (MESSAGE CONTENT IDENTIFIER)
SCB = STRING CONTROL BYTE (MESSAGE BYTE COUNT, OR "SIGNAL")

Fig. 2. Message format

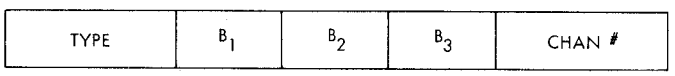


Fig. 3. RCB format

LABEL	VALUE	ASSOCIATION
LNKTAB	+4	NUMBER OF DEFINED LINKS
	'A' ADDRESS OF LCBA	SIGMA 5
	'B' 0	930
	'C' ADDRESS OF LCBC	910
	'D' 0	UNASSIGNED
LNKTBT	-4	NEGATIVE INDEX

Fig. 4. Link Table structure for four machines

LABEL	CONTENTS	WORD NUMBER
LCBA	CHAIN POINTER	1
	STATE	2
	CURRENT BLOCK	3
	TIMER	4
	CHANNEL-ON-FLAGS	5
	READ FLAGS	6
	HIS READ FLAGS	7
	BUSY WRITE FLAGS	8
	BLOCKED WRITE FLAGS	9
	WRITING STACK HEAD	10
	WRITING STACK TAIL	11
	$ -2	12
	CHANNEL BLOCKS 1-7	13 – 19
	$ +1	20
	BLOCK CHAIN 'A1'	21
	SHORT BUFFER BLOCK	22 – 29

Fig. 5. LCB format

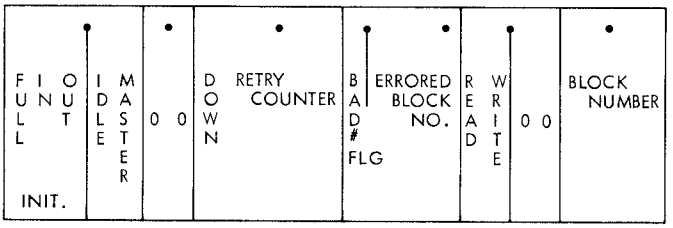


Fig. 6. LCB state word format