Bolt Beranek and Newman Inc.

Technical Information Report No. 99

The ARPANET Pluribus IMP Program

Volume I: Introduction, the IMP Algorithm, the STAGE System

May 1978

Prepared for:
Defense Communications Agency

Technical Information Report No. 99

THE ARPANET PLURIBUS IMP PROGRAM

Volume I

Introduction, The IMP Algorithm, The STAGE System

May 1978

Volume I

TABLE OF CONTENTS

Volume II

Table of Contents

Volume I

Table of Figures

Volume I

Table of Tables

Volume II

Table of Tables

Chapter 1
Introduction

During the past decade, the technology of packet-switching has come into increasing use in the design and construction of computer networks. Networks using this technology are generally characterized by:

1. A subnetwork of communications processors to which the host computers are connected.

2. A high degree of connectivity among the communications processors forming the nodes of this subnetwork.

3. The division of messages into packets, typically 1000 bits in length, by the communications processor to which the originating host is connected.

4. The dynamic routing of packets to the destination communications processor for reassembly and delivery to the host to which the message is addressed.

The pioneering, largest, and most advanced of these networks is the ARPANET(1) whose approximately 60 nodes currently net over 120 host computers; it has been in operation since 1969. Its basic technology has been adopted by computer networks in government and industry, both here and abroad, e.g., AUTODIN II, CTNE, CYCLADES, DATAPAC, DDX, EIN, EPSS, NPL, TELENET, TRANSPAC.

Some networks, e.g., EDN, COINS, PLATFORM, PWIN, use the same (or substantially the same) equipment and terminology as the ARPANET, including the communications processor which, for these networks, is called the Interface Message Processor (IMP). Two kinds of IMPs are in use: the Honeywell H-316(2) and the Bolt Beranek and Newman (BBN) Pluribus.(3)

---

(1) "Selected Bibliography and Index to Publications about ARPANET," Becker and Hayes, Inc., Los Angeles, California, 1976. NTIS AD-A026900.
(2) F.E. Heart, et al., "The Interface Message Processor for the ARPA Computer Network," AFIPS Conference Proceedings, Vol. 36, 1970, pp. 551-567.
(3) F.E. Heart, et al., "A New Minicomputer/Multiprocessor for the ARPA Network," AFIPS Conference Proceedings, Vol. 42, 1973, pp. 529-537.

The message packet processing functions performed by a Pluribus IMP are the same as those of a H-316 IMP, but there are significant differences in the manner in which their programs are designed and implemented. These differences are dictated by the architectural differences between the two types of computers. The H-316 is a single-processor, interrupt-driven machine while the Pluribus is a multiprocessor which uses parallel processing and a priority ordered Pseudo Interrupt Device (PID) to control the sequence of execution of all program tasks and the servicing of I/O devices to achieve real-time response.

The Pluribus consists of processors, memory modules, I/O devices, buses over which these communicate, and bus couplers to interconnect the individual buses. Within this framework, a wide variety of systems can be configured, ranging from small single-bus machines to large multibus systems with tens of processors, up to 1024K bytes of main memory, and many diverse I/O devices. All Pluribus processors are functionally equivalent; any processor can perform any system task and control any device.

The principal responsibility for maintaining reliability in the Pluribus is placed on its software. The Pluribus hardware was designed to provide an appropriate vehicle for the software reliability mechanism. When hardware errors are detected, the software exploits the redundancy of the hardware by constructing a new logical system configuration which excludes the failing resource, using redundant counterparts in its place. A small hierarchical operating system called STAGE(4) coordinates the software reliability mechanisms involved.

The H-316 IMP program has been documented in BBN Technical Information Report No. 89, The Interface Message Processor Program, March 1977 (periodically updated). The purpose of the present report is to document the Pluribus IMP program. The report is divided into two volumes. Volume I contains, in addition to this introduction, descriptions of the IMP and STAGE system programs; the debugging system DDT, detailed program descriptions, and data formats are included in Volume II. A general familiarity with packet switching and with the Pluribus architecture, e.g., as described in Pluribus Document 2, System Handbook, (BBN Report No. 2930), will be helpful in using and understanding the present report. Unless otherwise stated, all

---

(4) J.G. Robinson and E.S. Roberts, "Software Fault-Tolerance in the Pluribus," AFIPS Conference Proceedings, Vol. 47, June 1978.

numbers are decimal; where hexadecimal numbers are used, they are followed by a "!".

As a final note, it is pointed out that some networks, e.g., Platform, use Pluribus IMPs with memory and I/O devices on the same bus, called an M/I bus. The Pluribus program treats these buses as if they were separate I/O and memory buses. In line with this approach, the discussion in the body of this document is functionally oriented and the reader should keep in mind that, for machines with M/I buses, all references to memory or I/O buses actually refer to the memory or I/O space of the M/I bus. The single instance involving a routine that deals specifically with the M/I bus structure is discussed in Section 3.5 (Stage CD).

Chapter 2
The IMP

This chapter describes the algorithms that the IMP uses in performing its functions as a network communications processor. The flow of messages through the network is illustrated in Figure 1. The host sends the IMP a message up to 8063 bits long, preceded by a leader which specifies its destination. The source IMP accepts the message in packets up to 1008 bits long. Each packet has a header to allow for the transmission from IMP to IMP.(5) Figure 1 demonstrates how message 1 is transferred from IMP to IMP in three packets, numbered 1-1, 1-2, and 1-3. When a packet is successfully received at each IMP, an acknowledgement or ack is sent back to the previous IMP. Inter-IMP acks are shown returning for each packet. Finally the message arrives at the destination IMP where it is reassembled: that is, the packets are recombined into the original message. The message is sent to the destination host and a Ready for Next Message (RFNM) is sent back to the source host. A RFNM is a unique, one-packet message and it is acknowledged at each IMP to IMP hop on its return path.

As shown in Figure 2, several layers of nested protocols are involved in transmitting a message between application programs in different hosts, as follows:

1. Host-to-Host. Protocols between source and destination hosts.

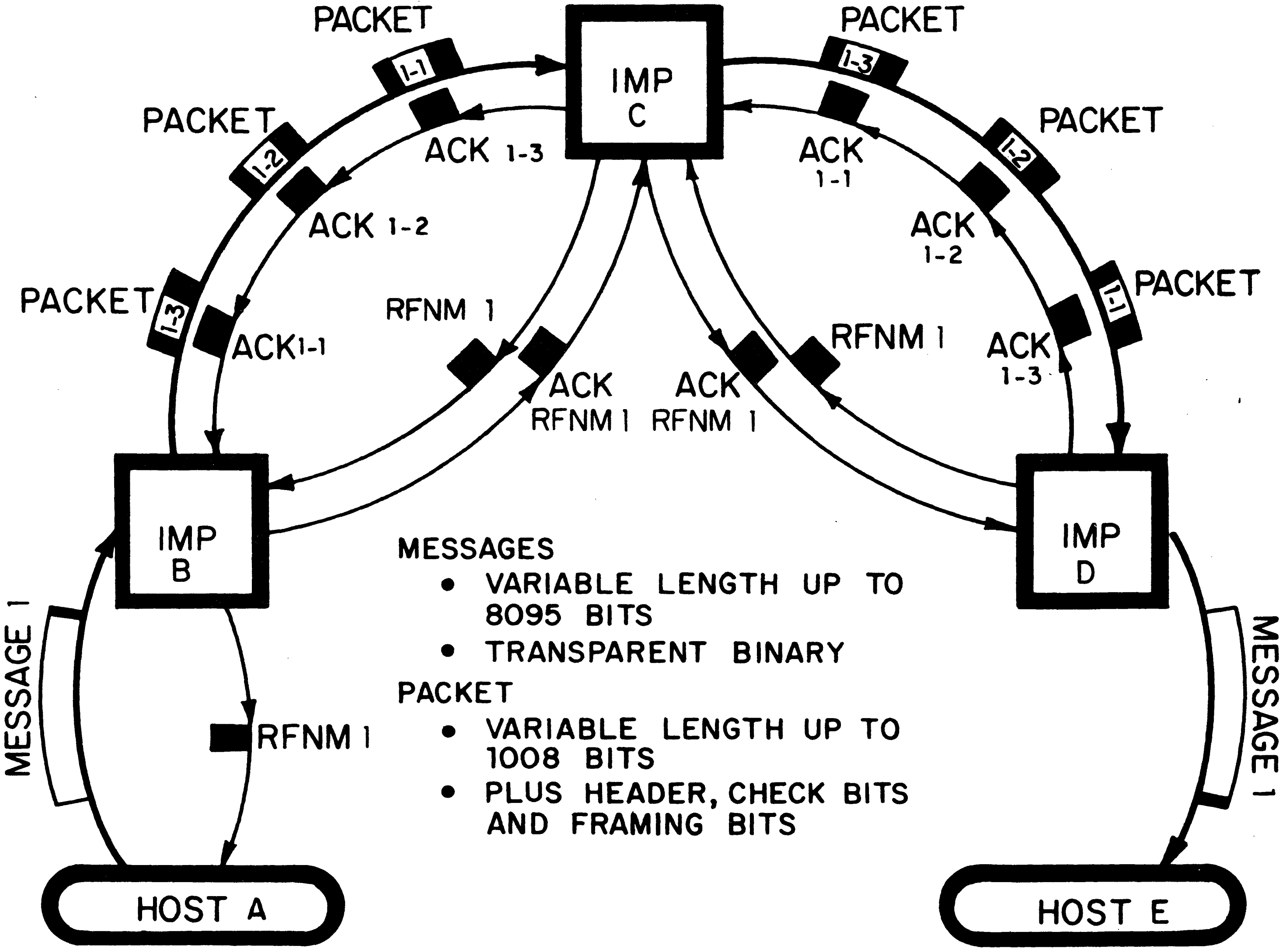
2. Host/IMP. Protocols between a host and its local IMP.

3. Subnetwork. Protocols between IMPs.

4. End-to-End. Protocols between source and destination IMPs.

5. IMP-to-IMP. Protocols between adjacent IMPs serving as store-and-forward nodes.

(5) Note the distinction between the leader, which appears at the beginning of a message as it passes between a host and an IMP, and a header, which appears in front of a packet in an IMP or in transit between IMPs.



MESSAGES
- VARIABLE LENGTH UP TO 8095 BITS
- TRANSPARENT BINARY

PACKET
- VARIABLE LENGTH UP TO 1008 BITS
- PLUS HEADER, CHECK BITS AND FRAMING BITS

Figure 1. Network Message Flow

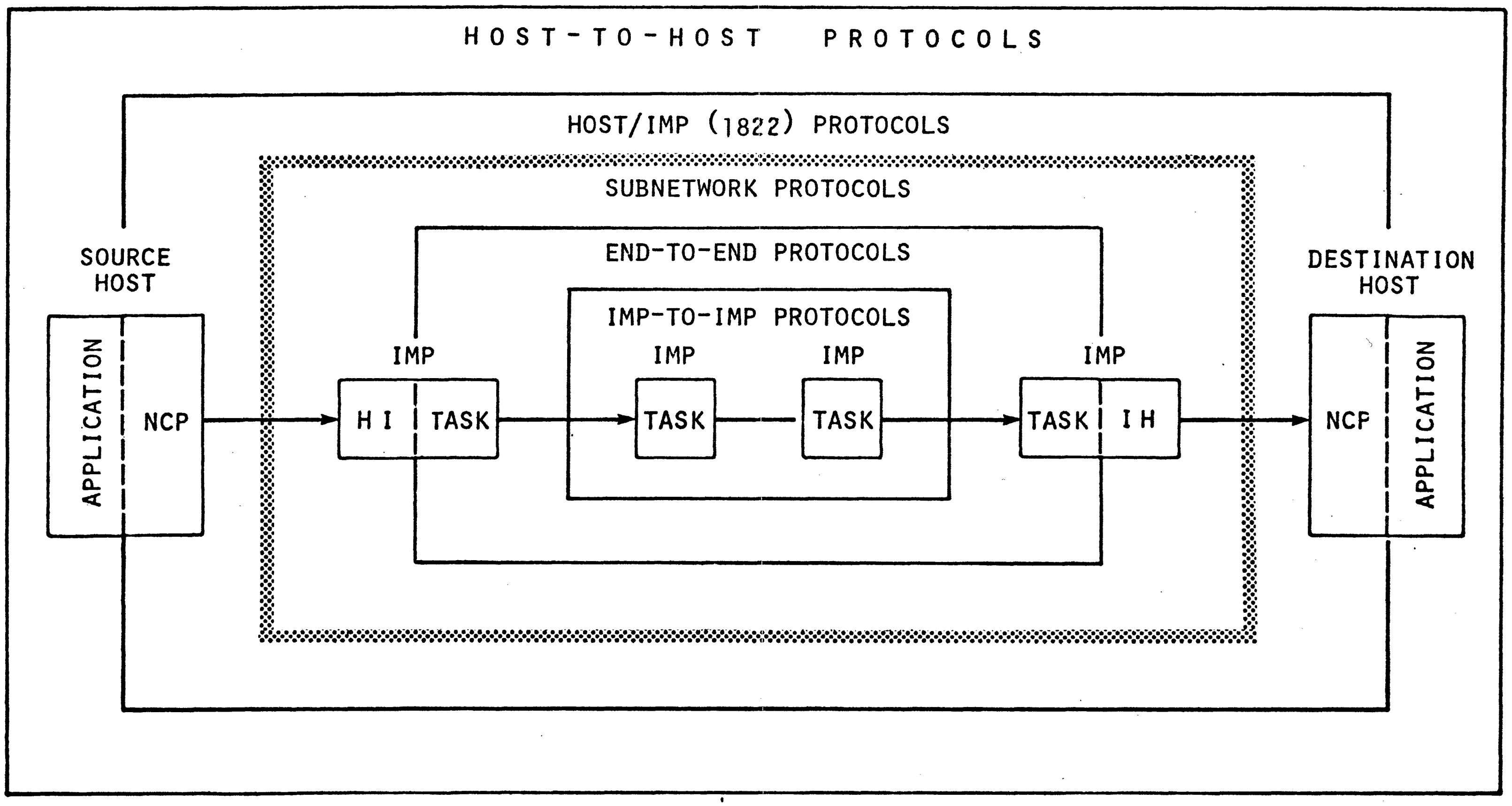


Figure 2. Network Protocols

The highest level of protocol in the network is the host-to-host protocol governing the transmission of messages from a source to a destination host. A message originating in a host's application program is passed to that host's Network Control Program (NCP) which performs all communication between its host and the network. The host-to-host protocols are implemented in the NCP. They establish the rules by which the conversation between the source and destination hosts will be held. This permits two architecturally different hosts to communicate.

The next protocol level is the host/IMP protocols which enables the host's NCP to pass a message on to the local IMP. This is accomplished by following the so-called "1822" protocol described in BBN Report No. 1822, "Specifications for the Interconnection of a Host and an IMP." That document defines the hardware interface between a host and an IMP and specifies the protocol to be followed in transmitting messages between them. The host's NCP casts the message into the proper format and precedes it by a <u>leader</u> which specifies such data as the destination host's network address, the message's priority, etc. It then takes the appropriate hardware actions to transmit the leader and message to the IMP. At some subsequent time, the IMP to which the receiving host is connected receives the message and passes it to the host through its own 1822 interface.

The remaining lower protocols are implemented between the IMPs of the subnetwork and no longer concern the participating hosts. This is indicated in Figure 2 by the shaded box.

The next protocol level is the <u>end-to-end</u> message protocol. The source IMP receives the message through the 1822 interface and passes it to the host-to-IMP (HI) module. The information from the leader is stored in a transaction block, a data area reserved for the purpose, and the leader is examined to determine the addressee. For data to flow in the network, a conversation must be initiated with the destination IMP unless such a conversation already exists. A series of protocol messages is transmitted back and forth between the source IMP and the destination IMP, resulting in a transmit message (TM) block in the source IMP and a receive message (RM) block in the destination IMP. When each IMP knows of the existence and identity of the relevant block in the other, the conversation has been initiated. Note that "conversation" is a technical term referring to a one-way transmission of data. Although control messages pass in both directions in a conversation, data move in one direction only. (A two-way exchange between hosts actually involves two conversations in the IMP subnetwork.) Once a conversation has been initiated, it may be used for many messages; it is terminated automatically by the IMP subnetwork when it falls into disuse.

Having initiated the conversation, the host-to-IMP module (HI) in the source IMP breaks the message into packets up to 1008 bits long and invokes the basic store-and-forward module TASK to send each packet to the destination IMP. TASK is called upon to transmit only one packet at a time. Once the message has been received at the destination IMP, the module TASK in the destination IMP invokes the IMP-to-host module (IH) to transmit the message to the destination host. Notice that this level of discussion starts with a message received in the IMP through HI which is passed off to TASK in that IMP, and that TASK in the destination IMP passes the message through IH to the receiving host. All processing just described takes place even if both hosts are connected to the same IMP.

The next level of discussion involves the transmission of a single packet from one IMP to another. TASK is invoked by handing it a packet which it is to move towards its ultimate destination. Such packets may arise from either of two sources: from HI as just described, or from an adjacent IMP. The effect is the same in either case.

When an IMP receives a packet, that packet is either addressed to a host connected to this IMP or to a host connected to some other IMP. In the latter case TASK must determine the next leg of the packet's route. Each IMP has several communication links connecting it to adjacent IMPs, and TASK must determine which of these links is the best one to the destination, given the present state of the network. The <u>routing algorithm</u> gathers data about the current status of network traffic and transmits these data periodically from one IMP to another. The results of this data gathering operation are used to create tables which TASK can interrogate to determine, for any given destination IMP, the best line over which to transmit the packet. Since routing messages are received periodically by each IMP, successive packets of a message may be transmitted via different routes, leading to the possibility that the packets are received at the destination IMP out of order. (New routing information may reveal a less busy route, so that a subsequent packet may travel more rapidly than an earlier one.) For this reason, the IMP algorithms are designed to reassemble the packets of a multi-packet message in the proper order.

Having determined the line over which to transmit the packet, TASK then transmits it to an adjacent IMP (which may or may not be the destination IMP). TASK in that IMP goes through an identical process. This process continues until the packet arrives at the destination IMP, at which time TASK passes it to the module FORUS which performs the necessary reassembly of the packets into a message. FORUS also insures that messages are passed to the host in proper order.

A final protocol is the low-level IMP-to-IMP protocol, which governs usage of the communication links between IMPs. Each physical link between IMPs is divided (by the software) into up to 128 logical channels, so that up to 128 packets at a time may be in transit in each direction between each pair of IMPs. The channel concept permits each IMP to initiate transmission of subsequent packets over a link before receiving acknowledgement of the successful transmission of the first. Note the distinction between the end-to-end protocol and the low-level IMP-to-IMP protocol. The former is concerned with the concept of the conversation, the latter with channels.

2.1 The IMP Algorithm: Introduction and Overview

The data flow through the IMP is shown in Figure 3, a schematic drawing of packet processing. The processing programs are described below.

The host-to-IMP routine (HI, shown in the lower left corner of the figure) handles messages being transmitted into the IMP from a local host. The routine first accepts the leader to construct a header that is prefixed to each packet of the message. It then accepts the first packet and, if no allocation of space exists for the destination IMP, constructs a request for buffer allocation which it places on TASK's queue. A single-packet message is placed directly on the task queue regardless of allocation status and a copy is held in the transaction block until either a RFNM or allocation is returned. A returned RFNM causes the packet to be released (since the message has been received), while a returned allocation for the single-packet message causes retransmission by TASK. Requests for multi-packet allocation are sent without actual message data. The request is recorded at the destination IMP and an allocation message is returned by a background process as soon as space is available. A returned allocation causes HI to place the first packet with header on TASK's queue. Input of the rest of the message is then accepted from the host. HI also verifies the

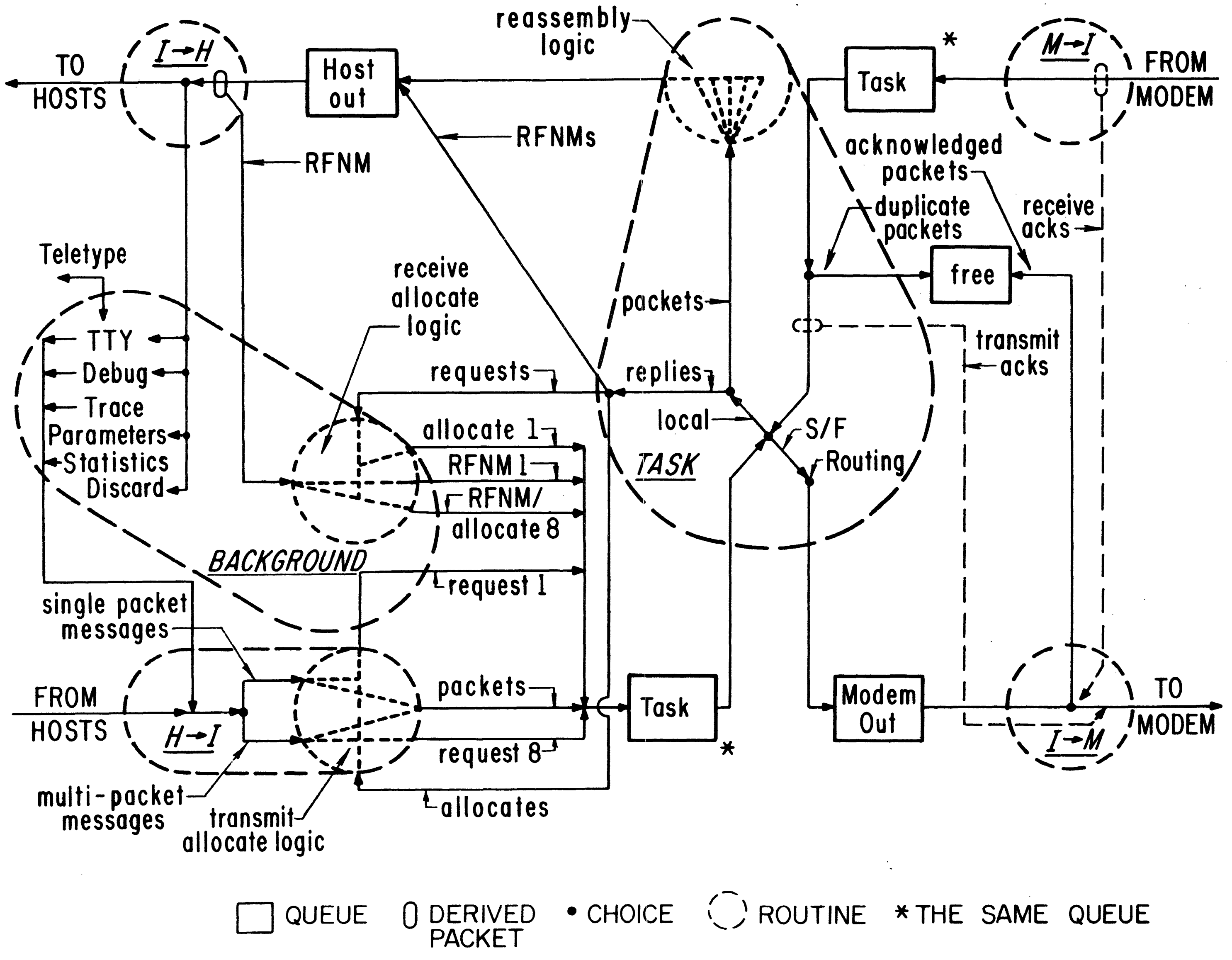


Figure 3. Packet Flow and Processing

message format. The routine is reentrant and services all hosts connected to the IMP.

The modem-to-IMP routine (M2I, shown in the upper right corner of the figure) handles inputs from the modems through which the IMP is connected to the communication links. This routine first sets up a new input buffer, previously obtained from the free list. (That is, M2I performs double buffering.) If a buffer cannot be obtained, the received packet is not acknowledged and the buffer is reused immediately to read in the next packet. The discarded packet is retransmitted later by the distant IMP as soon as a timer runs out. M2I processes returning acknowledgements for previously transmitted packets and either releases the packets to the free list or signals their subsequent release to the IMP-to-modem routine. M2I then places the buffer on the end of TASK's queue.

TASK uses the header information to direct packets to their proper destination. It routes packets from the task queue either to a local host queue or onto an output modem determined from the routing tables. If the packet is for non-local delivery, TASK determines whether sufficient store-and-forward buffer space is available. If not, buffers from modem lines are flushed and no subsequent acknowledgement is returned by I2M. (Normally, an acknowledgement is returned with the next outgoing packet over that modem line.) Packets from hosts which cannot get store-and-forward space are freed by TASK and requeued at a later time by HI.

If a packet from a modem line is addressed for local delivery, its message number is checked to see whether a duplicate packet has been received. Each IMP maintains for each connection a window of contiguous message numbers which it will accept from the other end of the connection. Packets with out-of-range numbers are considered duplicates and are discarded. The receipt of a RFNM for the oldest message at the source IMP permits the window to be moved up by one number.

Replies such as RFNMs or Dead Host messages are placed in transaction blocks. TASK then pokes IH to initiate output to the host.

Message packets for local delivery are linked together with other packets of the same message number in a reassembly block. When a message is completely reassembled, the leading packet is linked to the appropriate host output queue for processing by IH.

Incoming routing messages are processed by the routing program with high priority so that outgoing routing messages and the routing directory immediately reflect any new information received. M2I generates I-heard-you messages to indicate to the neighbor receipt of the routing message.

The IMP-to-modem routine (I2M) transmits successive packets from the modem output queues and sends piggybacked acknowledgements for packets correctly received by M2I and accepted by TASK.

The IMP-to-host routine (IH) passes messages to local hosts and informs a background process when a RFNM should be returned to the source host.

A fake host is a program in the IMP which acts like a real host in many ways, including being the source or destination of network traffic. The four fake hosts are:

1. Fake Host 0--The terminal connected to the Pluribus IMP.

2. Fake Host 1--The debugging process DDT.

3. Fake Host 2--The packet core process used to reload part of the memory should it be found to be incorrect.

4. Fake Host 3--Used for miscellaneous purposes such as reports to the NCC, message generation, etc.

Selected hosts and IMPs, particularly the Network Control Center (NCC), find it necessary or useful to communicate with one or more of these fake hosts.

The TTY fake host assembles characters from the terminal into network messages and decodes network messages into characters for the terminal. TTY's default message ("crosspatch") destination is the DDT fake host at its own IMP. It can, however, be connected to any other IMP terminal, any other IMP's DDT fake host, or to any host program with compatible format.

DDT permits the operational program and its data to be inspected and changed. Although its normal message source is the TTY fake host at its own IMP, DDT responds to a message of the correct format from any source. This program is normally inhibited from changing the operational IMP program; NCC intervention is required to activate the program's full power.

The STATISTICS fake host collects measurements about network operation and periodically transmits them to a designated host. This program sends but does not receive messages.

The PACKET CORE fake host loads and dumps portions of its own IMP's memory, or acts as an intermediary in loading and dumping portions of the memory belonging to a neighbor who is unable to communicate via the normal IMP-to-IMP protocol. The PACKET CORE facility allows for dissimilar machines to coexist as IMPs on the network; reloading and diagnostic dumping of a malfunctioning IMP can be done without the requirement that one of its neighbors be of the same machine type.

Background Hosts. These are modules which are run periodically and search the IMP's data bases for certain tasks to perform. They send connection protocol messages, incomplete transmission messages, allocations, and RFNMs, as well as returning GIVEBACKs closing unused connections. The background hosts run in a slightly different manner than the fake hosts in that they do not simulate the host/IMP channel hardware. They do not go through the host/IMP code at all, but put their messages directly on the task queue. Nonetheless, the principle is the same.

2.1.1 IMPs and Hosts

The software interface between an IMP and a host will now be defined; the details of the hardware interface are to be found in BBN Report No. 1822. Each IMP serves hosts whose cable distances from the IMP are less than 2000 feet. A modem channel must be used for greater distances; this latter type of host connection is termed a Very Distant Host (VDH) and is also discussed in BBN Report No. 1822.

Connecting an IMP to a wide variety of different local hosts requires a hardware interface, some part of which must be custom tailored to each host. The interface is therefore partitioned so that a standard portion can be built into the IMP which is identical for all hosts, while a special portion of the interface is unique to each host. The interface is designed to allow messages to flow in both directions at once; a bit-serial interface is used.

The host interface operates asynchronously, each data bit being passed across the interface via a four-way Ready-for-Next-Bit/There's-Your-Bit handshake procedure. This technique permits the bit rate to adjust to the rate of the slower member of the pair and allows necessary interruptions when words must be stored into or retrieved from memory.

A message from a host consists of a leader followed by data bits. The leader format was changed in late 1976 to accommodate more than 63 IMPs in the network and more than 4 hosts per IMP. Since some hosts have yet to be reprogrammed for the new format, the IMPs support either format although all internal processing in the IMP assumes new format. Any leader in old format is translated to new format immediately upon receipt by HI, and the format is changed just before transmission of a message by IH to a host using old format.

The format of the message leader provides a 16-bit field for IMP numbers, so that in principle there can be as many as 2**16 IMPS. In practice, other restrictions limit the IMPs to being numbered between 1 and 67.(6) Host numbers are in an 8-bit field and may range from 0 to 255. Because of storage limitations, 26 (including fake hosts) is the maximum possible number of hosts. Four of these host numbers are reserved permanently for fake hosts, numbered 252 through 255, as follows:

252	local terminal
253	DDT, the debugging process
254	packet core
255	statistics, message generation/discard

The software for the fake host simulates 1822 hardware.

Messages intended for dead hosts (which are not the same as dead IMPs) cannot be delivered; they require special handling to avoid indefinite circulation in the network and spurious arrival at a later time. Such messages are purged from the network at the destination IMP. A host computer is notified about a dead host only when it attempts to send a message to it.

(6) IMP number 0 is not usable because of certain conventions in message fields, and 67 is the maximum number of fields transmittable in certain parts of routing information.

2.1.2 Network Flow Control

A major hazard in a message-switching network is congestion, which can arise either from system failures or from peak traffic flow. Congestion typically occurs when a destination IMP becomes flooded with incoming messages for one or more of its hosts. If the flow of messages to this destination is not regulated, the congestion backs up into the network, affecting other IMPs and degrading or even completely clogging the communication service. To avoid this problem, the IMPs incorporate a quenching scheme that limits the flow of messages to a given destination before congestion begins to occur.

This quenching scheme requires that buffer space be allocated where it will be needed before a message may enter the system. If buffering is provided in the source IMP, one can optimize for low delay transmissions; while if the buffering is provided at the destination IMP, one can optimize for high bandwidth transmissions. To be consistent with the goal of a balanced communications system, the approach used utilizes some buffer storage at both the source and the destination as well as a request mechanism from source IMP to destination IMP.

Specifically, no multi-packet message is allowed to enter the network until enough storage for the message has been allocated at the destination IMP. As soon as the source IMP realizes that a message is multi-packet, it sends a control message to the destination IMP requesting that reassembly storage be reserved at the destination for this message. It does not take in further packets from the host until it receives an allocation message in reply.(7) The destination IMP queues the request and sends the allocation message to the source IMP when enough reassembly storage is free; at this point the source IMP accepts the (rest of the) message from the host and starts to send it to the destination.

Effective bandwidth is maximized for sequences of long messages by permitting all but the first message to bypass the request mechanism. When the message itself arrives at the destination and the destination IMP is about to return the Ready For Next Message (RFNM), the destination IMP waits until it has

(7) This is not completely accurate, since the double buffering scheme employed in HI permits two packets to be read in while waiting for the allocation. In the usual case, the allocation arives before the host interface must be blocked.

adequate buffer and reassembly space for an additional multi-packet message and then piggybacks a storage allocation onto the RFNM. If the source host is prompt in answering the RFNM with its next message, an allocation is ready and the message can be transmitted at once. If the source host delays too long, or if the data transfer is complete, the source IMP returns the unused allocation to the destination. With this mechanism, the inter-message delay has been minimized and the hosts can obtain the full bandwidth of the network.

The delay for a short message has been minimized by transmitting it to the destination immediately upon its receipt from the host, retaining a copy in the source IMP. If there is space at the destination, the message is accepted at once and passed on to a host and a RFNM is returned, the source IMP discarding the message when it receives the RFNM. If there is not enough space, the destination IMP discards the message and queues within itself a request for allocation. (Effectively, it treats it as a request for space.) When space becomes available, the source IMP is notified that the message may now be retransmitted. Thus, there is no setup delay at all in the vast majority of cases in which storage is available at the destination.

These mechanisms make the IMP network fairly insensitive to unresponsive hosts, since holding the source host to a transmission rate equal to the reception rate of the destination host prevents clogging the network with messages. Further, reassembly lockup is prevented because the destination IMP never has to turn away a multi-packet message destined for one of its hosts; reassembly storage has been allocated for each such message prior to its entry into the network.

2.1.3 End to End Communications

To communicate, both source and destination IMPs must establish a record of the connection between them. This simplex connection, consisting of a Transmit Message (TM) block at the source, and a corresponding Receive Message (RM) block at the destination, is created and later removed using a special protocol which detects duplicate or missing messages. Each message transmitted as part of a conversation contains the index of the relevant block (TM or RM) at the far end. The connection is disallowed if the host/host access control mechanism does not permit that host pair to communicate.

Every IMP maintains for each of its hosts a pair of Host Access Control (HAC) words in which each of the 16 bits represents one of sixteen logical subnetworks. The bits in one word signify membership in, and in the other word signify permission to communicate with, the subnetworks. A pair of hosts may communicate with each other only if they are members of the same logical subnetwork or if one is allowed to communicate with hosts in a subnetwork of which the other is a member.

A conversation is a one-way message path from a source IMP to a destination IMP, where "one-way" means that data are transmitted in only one direction although control messages move in both directions. The basic control loop in HI that deals with conversations is shown in Figure 4. After going through some

```
initiate conversation
          |
          |
          |
request allocation <-----
  (or have it)          |
          |             |
          |             |
          |             |
   send packets ---------
```

Figure 4
Basic Loop in HI

initial protocol to set up the conversation, the two remaining steps are to request an allocation and to send packets. The action of requesting an allocate can be by-passed in the event that an allocation is available. (It may have been provided on the RFNM returned for the previous message.)

The message sent to the remote IMP to initiate a conversation is GETABLOCK, which asks that an RM block be set up. The receiving IMP replies either GOTABLOCK or CANT, depending upon whether or not there is an available RM block which it can allocate for this conversation. If it finds such a block, then a conversation has been successfully initiated.

For each conversation, an independent message number sequence is maintained by each of the two participating IMPs, the originating IMP maintaining it in the TM block and the destination IMP in the RM block. The counter is incremented by one for each message sent over the conversation. The transmitting IMP assigns the message numbers in sequence, and the receiving IMP uses the same message number as part of the acknowledgement. Since it is an inherent property of the store-and-forward protocol that messages may arrive at the destination out of order, a window of eight messages is maintained, and the reassembly process is willing to accept any message within that window. (FORUS assures that messages are delivered to the host in proper order.) As it receives a message and acknowledges it to the source (by a RFNM), the transmitting IMP moves the window up past each successfully acknowledged message. The end-to-end protocol permits up to eight messages per conversation to be in the network at any time, thus allowing a host to send messages rapidly accross the network despite delays in returning RFNMs. If a host tries to get ahead by more than eight messages, the transmitting IMP blocks it. Messages arriving at a destination IMP with message numbers outside of the current window or with message numbers already marked as received are duplicates to be discarded. The message number concept serves two purposes: it orders the messages for delivery to the destination host, and it provides for the detection of duplicate and missing messages. The message number is internal to the IMP subnetwork and invisible to the hosts.

A sequence control system based on a single source/destination connection, however, does not permit priority traffic to go ahead of other traffic. More generally, a host may wish to request special treatment for a message; thus, a separate connection is created for each "handling type." Currently, there are two possible handling types: regular (for high bandwidth) and priority (for low delay).

When a request for an allocate comes in to an IMP with an associated message number, that message number should be in an idle state. (If the message number is busy, the allocation is a duplicate and is discarded.) FORUS puts this conversation in the state "need an allocate."

When an allocate is returned, it goes into the TM block. That is, a record is made in the TM block of the amount of space allocated. The originating host is blocked until the allocate is returned. When the source IMP receives the allocate, it starts to send packets. Because of the double buffering in the host-to-IMP interface, the source host may have already sent two packets. However, the IMP cannot receive more than that and does not attempt to do so until it has received word from the destination IMP that the storage allocation is available. It is for this reason that it is important that allocation processing be expedited; therefore, the destination IMP is willing to wait for up to half a second to be able to piggyback the allocation on a RFNM for a previous multi-packet message.

At the destination IMP, the packets are collected as they are received. The reassembly block contains a pointer to the first packet received, and each successive packet is threaded onto this packet list in the proper order for the message. Thus duplicate packets can be discarded as soon as they are received when the attempt to thread them onto the list reveals an already received packet with the same packet number. If a packet is lost, the source IMP sends an Incomplete Query message after 30-45 seconds.

Special processing is provided for single packet messages so that they can be transmitted expeditiously. Upon receiving a single packet message from the host, HI transmits it to the destination IMP as a request for an allocation of one block but also retains a copy in the transaction block. However, all of the data accompanies this request for allocation. The destination IMP attempts to find a buffer for the data and (if it is not the next message to go to the host) a reassembly block. If it is able to find both, then the message is complete; it is sent to the destination host and a RFNM is returned. If there is no available buffer and reassembly block, the message is treated as if it were merely a request for an allocation and the data that accompanied it is discarded. Eventually a reassembly block and a buffer are allocated and an allocation of one is sent to the source IMP.

2.1.3.1 Error Recovery

Since packets are sent between IMPs via potentially unreliable communication links, procedures have been developed to detect and account for a lost packet or message. The IMP algorithm takes various steps to continue smooth operation even if packets or protocol messages are lost in transit. Some of the techniques employed are presented in this section.

The source IMP keeps track of all messages for which a RFNM has not yet been received, and the destination IMP keeps track of the replies it either has yet to send or has already sent. When the RFNM is not received for too long (about 30-45 seconds), the source IMP sends an "Incomplete Query" protocol message (using the same message number) to the destination, a message which inquires in effect, "What is the status of this message number?" If the destination has already received that message (that is, if the acknowledgement was lost in the network), then a duplicate acknowledgement is sent. If some part of the message was lost in transmission, the destination replies, "I've just received an incomplete message." This reply includes enough details about the error so that appropriate corrective action can be taken. At the very least, the originating host can be informed that the message was lost in transit. The source IMP continues inquiring until it receives a response. This technique generally insures that the source and destination IMPs keep their message number sequences synchronized and that any allocated space is released should a message become lost in the subnetwork because of a machine or communication line failure.

A conversation is terminated either after a prolonged period of inactivity, or after a somewhat shorter period of inactivity coupled with the need for the message block by some other connection, or by the need to resynchronize a message number sequence that has been broken. The special termination protocol can be initiated by either the source or the destination in the first two of the cases mentioned above, or by the source in the third case upon the receipt of an "out of range" response to an Incomplete Query. Upon closing a conversation, both source and destination IMPs release all resources held or allocated for that conversation.

2.1.3.2 Raw Packets

The network provides a facility outside of the normal host/host connection mechanism for sending and receiving "raw packets." These messages are identified by a special host-IMP and IMP-host code and bypass the connection mechanism. They are routed normally through the subnetwork, but no sequencing, error control, reassembly, or storage allocation is performed. Thus, they may arrive out of order at the destination host, some packets may be missing or duplicated, or packets may be thrown away by the subnetwork if insufficient resources are available to handle them. No RFNMs or other messages are sent back to the source host about such raw packets. Since there is no flow control, a host can overload the net with raw packets and cause it to fail; therefore, a special privilege bit (in the host access word) is required for a host to be permitted to send them.

2.1.4 IMP to IMP Communication

The preceding section dealt with the end-to-end protocol of sending messages from a source IMP to a destination IMP. At a lower level is the IMP-to-IMP protocol involved in sending individual packets from one IMP to another. This protocol is now discussed.

The mode of operation in connection with IMP-to-IMP transmission of packets is as follows: when a packet is transmitted from one IMP to another, the sending IMP retains a copy. When the IMP at the other end of the link has successfully received the packet, it acknowledges it to the sender. On receipt of that acknowledgement, the sender is able to release the buffer space in which the packet copy is held. If an acknowledgement is not received in time, the sending IMP merely retransmits the packet. The exact length of time an IMP will wait before retransmission depends on the type of line being used (land vs. satellite) as well as the bandwidth of the link (low speed vs. high speed).

A simple acknowledgement discipline applies to a channel,(8) a connection between one IMP and another. Over each channel, each packet is assigned a gender as being either even or odd;

(8) "Channel" is a technical term; a full definition will be found later in this section.

associated with each packet is a single bit which specifies the packet's gender. For a channel from IMP A to IMP B, every data packet or null sent back from B to A contains an acknowledgement (ACK) field specifying for each channel the gender of the last packet received from A. It is this field which provides the acknowledgement required by the IMP-to-IMP protocol. For example, after a packet of gender even has been transmitted from A to B, the ACK field in each packet sent back from B to A is examined. When that field shows that the channel status is even, then IMP A knows that the packet has been received correctly and that it is able to discard its copy and to transmit additional packets over the channel. The next packet it transmits then has gender odd, and again the acknowledgement is detected when the ACK field in any packet from B to A indicates that the gender of the channel is odd. Note that loss of a packet along with its associated ACK bits causes no harm other than a slight delay until the sender notes the acknowledgement, since the gender status is repeated in the next packet.

The purpose of multiple channels is to reduce the total delay in transmission over links with long delay, such as satellite links. Multiple channels allow an IMP to begin transmission of a second packet over a given link before it has received acknowledgement of the first one, with up to 128 packets in transit at any instant as already described. The number of channels available is dependent on the particular link and is contained in the MAXCHN word of each modem parameter block. On links with very long transit times it is appropriate to have large numbers of channels. The Pluribus IMP software supports eight channels on terrestrial links and up to 128 channels on satellite links.

Each physical link connecting one IMP to another is divided into 8 to 128 logical channels, and each packet sent over the link is assigned a channel number. Each packet transmitted has a channel field which identifies its channel number and has in addition a gender bit which specifies whether the packet is even or odd. Additionally, the ACK field is piggybacked onto every packet transmitted (other than routing) between IMPs. Thus IMP A is sending messages on each of up to 128 channels to IMP B, and each packet from B to A contains an ACK field specifying the gender of the last packet successfully received from A on each of the eight channels. At any particular instant, there can be between 0 and 127 packets pending receipt between any two IMPs in either direction. (This count of packets waiting is used in the routing algorithm as a measure of link activity, as described in section 2.2.4.) Since the algorithm in the 316 IMPs limits the maximum number of channels to 8 (16 with a special configuration option), more than 16 channels may be used only on links between two Pluribus IMPs.

Each packet is individually routed from IMP to IMP through the network toward the destination. At each IMP along the way, the transmitting hardware generates initial and terminal framing characters and hardware checksum digits that are shipped with the packet and are used for error detection by the receiving hardware of the next IMP. The format of a packet on an inter-IMP channel is shown in Figure 5.

Errors in transmission can affect a packet by destroying the framing and/or by modifying the data content. If the framing is disturbed in any way, the packet either is not recognized or is rejected by the receiver. In addition, the check digits provide protection against errors that affect only the data. The check digits can detect all patterns of four or fewer errors occurring within a packet, and any single error burst of a length less than twenty-four bits. An overwhelming majority of all other possible errors (all but about one in 2**24) is also detected. Thus, the mean time between undetected errors in the subnet should be on the order of years.

The network is designed to be largely invulnerable to circuit or IMP failure as well as to outages for maintenance. Special status and test procedures are employed to help cope with various failures. In the normal course of events the IMP program transmits "hellos" (routing messages) periodically to each of its neighbors. The acknowledgement for a "hello" packet is a null packet in which the I-heard-you (IHY) bit is set.

A dead link is detected by the sustained absence (approximately 3.2 sec) of IHY messages on that link. No new packets are routed onto a dead link, and any packets awaiting transmission are rerouted. Routing tables throughout the network are gradually adjusted to reflect the loss. Receipt of consecutive IHY packets for about 30 seconds is required before a dead link is defined to be alive once again. Section 2.2.1 contains further details.

A dead link may reflect trouble either in the communication facilities or in the neighboring IMP. Normal link errors caused by dropouts, impulse noise, or other similar conditions usually do not result in a dead link, because such errors typically last only a few milliseconds and only occasionally as long as a few tenths of a second. Therefore, it is expected that a link is defined as dead only when serious trouble conditions occur.

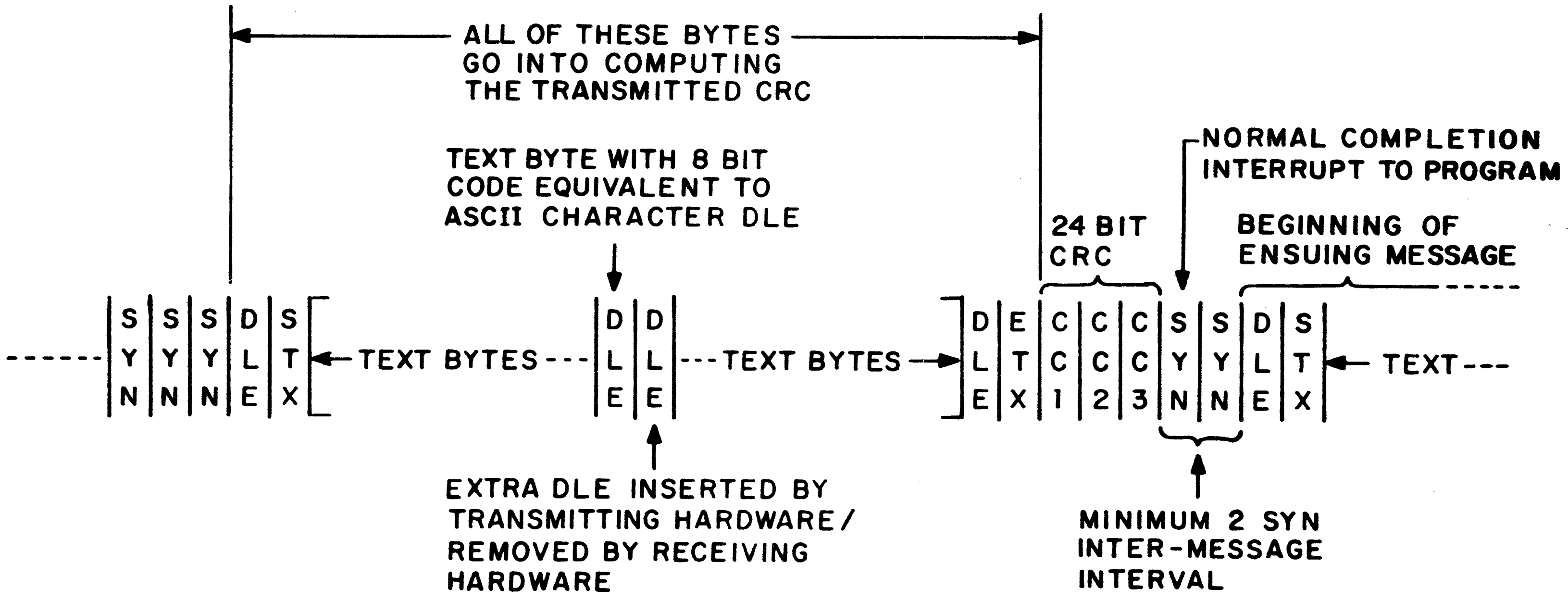

ALL OF THESE BYTES GO INTO COMPUTING THE TRANSMITTED CRC
TEXT BYTE WITH 8 BIT CODE EQUIVALENT TO ASCII CHARACTER DLE
NORMAL COMPLETION INTERRUPT TO PROGRAM
24 BIT CRC
BEGINNING OF ENSUING MESSAGE
SYN SYN SYN DLE STX
TEXT BYTES
DLE DLE
TEXT BYTES
DLE ETX CC1 CC2 CC3 SYN SYN DLE STX
TEXT
EXTRA DLE INSERTED BY TRANSMITTING HARDWARE/ REMOVED BY RECEIVING HARDWARE
MINIMUM 2 SYN INTER-MESSAGE INTERVAL

Figure 5. Format of a Packet

2.1.5 Routing

The purpose of the routing algorithm is to gather the necessary data so that TASK can readily direct each packet to its destination along a path for which the total estimated transit time is minimized. This selection is made by a fast and simple table lookup procedure. For each possible destination, an entry in the table designates the appropriate next leg. The values of these entries reflect line or IMP trouble, traffic congestion, and current local subnet connectivity. This routing table is updated periodically by the routing algorithm, as described below.

Each IMP estimates the delay it expects a packet to encounter in reaching every possible destination IMP over each of its output links. It selects the minimum delay estimate for each destination and periodically passes these estimates to its immediate neighbors in a "routing message." Each IMP then constructs its own routing table by combining its neighbors' estimates with its own estimates of the delay to each neighbor. The former is in the received routing message; the latter is based upon both queue lengths and the recent performance of the connecting communication circuit. For each destination, the table is then made to specify that selected output link for which the sum of the estimated delay to the neighbor plus the neighbor's delay to the destination is smallest.

Finally, the IMPs perform the routing computation on an incremental basis as each routing message is received. This strategy assures that the routing message output on a given link is as up-to-date as possible. The routing messages carry serial numbers to permit the IMPs to detect that a new set of routing data has arrived which is then used, with the current data, to form the next routing message.

2.1.6 IMP Reliability

The Pluribus IMP system includes software whose purpose is to maintain IMP reliability. This software must be contrasted with the STAGE system described in Chapter 3 which serves a different purpose. STAGE is independent of the application and serves to insure that the hardware is working properly. The reliability part of the IMP attempts to insure that certain things have not gone wrong with the application. The IMP reliability package assumes that the STAGE system is working properly and that the required hardware is available and working correctly. Figure 6 shows the relationship between the three major components of the Pluribus software.

The components of the IMP program dedicated to improving reliability have two main functions. First, the software is built to be as invulnerable as possible to hardware failures. Second, the software isolates and reports what failures it can detect to the NCC. With intermittent failures, it is important to keep the IMP program running and to diagnose the problem rather than letting the IMP go down for long periods to run special hardware diagnostics.

The discussion that follows describes the three major types of reliability mechanisms used.

1. Checks are made in line in the code to detect events "that can't possibly happen" and to take sensible actions when they do. For example, the routine that frees a buffer objects if its caller does not own the buffer.

2. Checks are performed by background hosts (as well as by other routines) which are poked periodically by TIMEOUT.

3. So-called "watchdog" timers are used to insure that the time between successive occurrences of an event is not excessive. Although details vary depending on the application, a typical technique is to set a timer positive whenever the associated event occurs. A background process decrements the timer periodically, **complaining if it ever reaches zero.**

In all three cases above, the action on detecting an error is to rectify the situation as much as possible, given the amount of context available. Also, a message is sent to the NCC reporting the occurrence. See section 2.2.8 for further details.

The IMPs use the technique of software checksums on all transmissions to detect errors in packets, protecting the integrity of the data and isolating hardware failures. The end-to-end software checksum on packets operates as follows:

-- A checksum is computed at the source IMP for each packet as it is received from the source host.

-- The checksum is verified at each intermediate IMP as it is received over the circuit from the previous IMP.

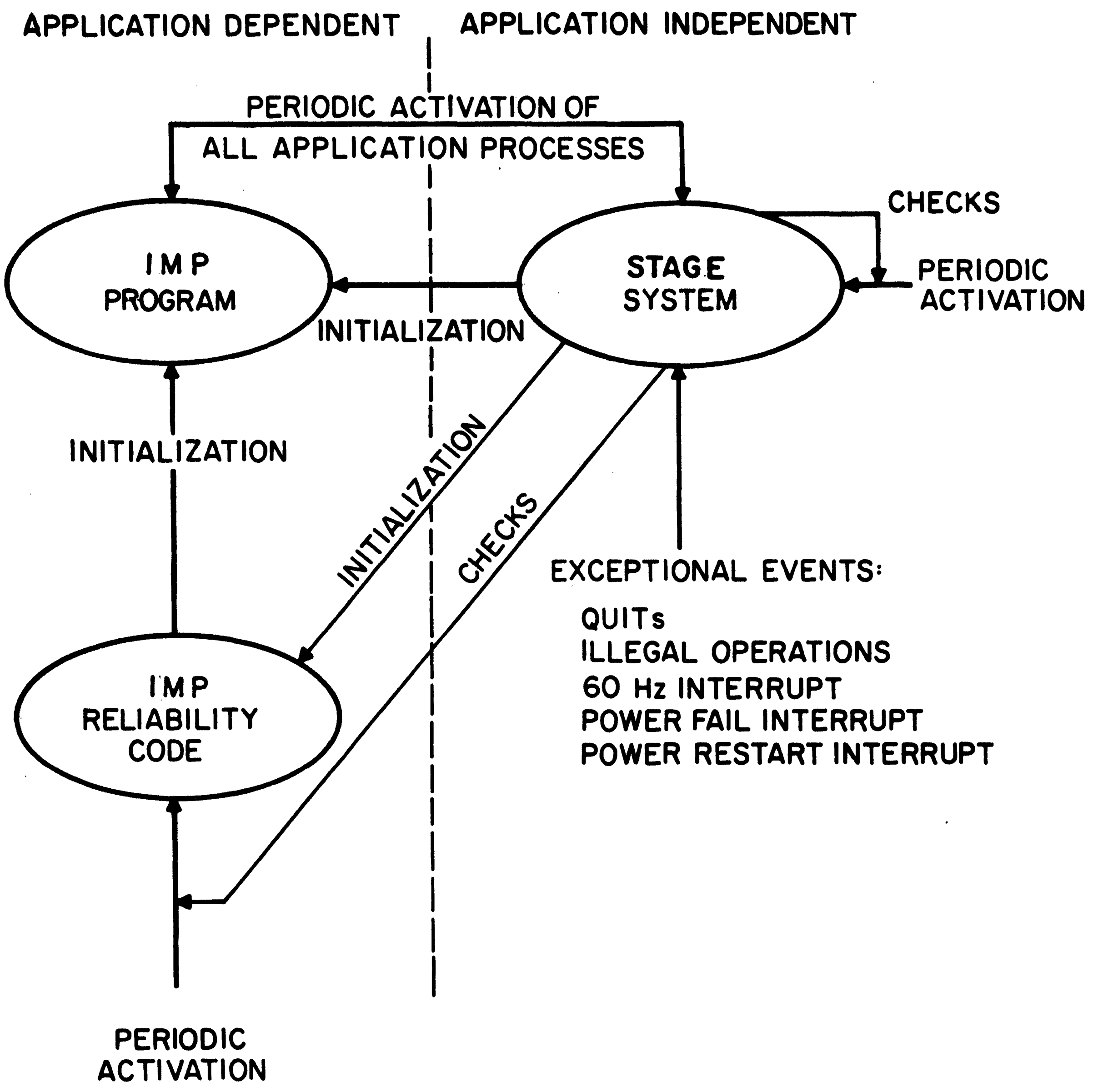


Figure 6. Reliability Software

-- If the checksum is in error, the packet is discarded, and the previous IMP retransmits the packet when it does not receive an acknowledgement.

-- To cut the number of checks in half, the previous IMP verifies the checksum of a packet only when it must be retransmitted and not before the original transmission. If a checksum error is found, an intra-IMP failure has been detected and the packet is lost. If not, the original transmission was lost due to an inter-IMP failure, circuit error, or was simply refused by the adjacent IMP. The previous IMP holds a good copy of the packet, which it then retransmits.

-- After the packet has successfully traversed several intermediate IMPs, it arrives at the destination IMP. The checksum is verified just before the packet is sent to the host.

This technique provides a checksum from the source IMP to the destination IMP on each packet, with no gaps in time when the packet is unchecked. Further, the length of each packet is verified as part of the checksumming operation. Any errors are reported to the NCC in full, with a copy of the packet in question. This method helps to make the IMPs reliable and fault-tolerant, and it provides a maximum of diagnostic information for use in fault isolation.

When an IMP has received and acknowledged a packet and fails before it is able to transmit the packet to the next IMP, the message is lost to the network and cannot be recovered; the receipt of the acknowledgement by the previous IMP has caused the release of the only copy of the packet. The source IMP informs the originating host that something went wrong with the message when it checks why no RFNM has been received, but recovery is possible only if effected by the originating host.

It is possible that a packet is received, sent on to the next IMP, but has not yet been acknowledged when the IMP goes down. In that case the sending IMP never receives an acknowledgement and ultimately retransmits the packet. However, it retransmits it via a different route so that the destination IMP may receive two copies of that packet. The algorithms are designed so that such duplication is detected and the second copy of the packet is discarded.

2.1.7 Timeout

Both the application program and the STAGE system require that certain items be checked periodically. A variety of timeout mechanisms are used to insure that such checks are performed.

The Pluribus real-time clock (RTC) provides the system's basic timing mechanism as well as a common time reference for all processors. The RTC generates a distinct PID level every 25.6 milliseconds; the routine invoked as a result is called the fast timeout strip. Some of TIMEOUT's responsibilities include checking for the correct operation of each RTC in dual-RTC machines, poking all modem output routines and fake host processes, and maintaining line state timers.

Every 128 milliseconds (5 clock ticks) TIMEOUT insures that certain other routines be performed. These "medium" timeout functions include maintaining the host interface hardware watchdog timers.

The slow timeout PID is poked by TIMEOUT every 640 milliseconds (25 clock ticks). Slow timeout is responsible for polling various routines which perform timing functions, reliability checks, and other assorted checks. Slow timeout polls these routines by scanning each common memory code page. The timeout table on each page, if any, lists the addresses of routines to be polled every slow timeout period. Slow timeout continues to poke itself until it has completed a pass through all the routines contained in the timeout table.

2.2 Organization of Major Modules

Section 2.1 presented the various algorithms of the IMP in overview form; the present section describes the IMP's major modules in greater detail.

Section 2.2.1 addresses modem input/output, the modules that interface the IMP to the communication lines that lead to other IMPs. Module M2I interfaces a modem to the IMP, and module I2M interfaces the IMP to a modem.

The IMP's software interface to the host is implemented in two modules, module HI which interfaces from the host to the IMP, and module IH which interfaces from the IMP to a host. These are discussed in section 2.2.2.

Section 2.2.3 presents TASK, the routine that provides packets to the low level IMP-to-IMP protocol. TASK receives inputs from, and directs its output to, local hosts and modems. The host software communicates with the hosts through the 1822 interface, and communicates with TASK through the TASK queues. TASK in turn communicates with the store-and-forward routines described previously. The communication in all cases consists of both data and control flow.

An overview of the routing algorithm was presented in section 2.1.5; section 2.2.4 contains the details of its operation.

Background hosts are host-like modules that run periodically to perform tasks not conveniently performed elsewhere; they are described in section 2.2.5.

Fake hosts are also host-like modules, but they communicate with other hosts while background hosts are invisible from outside the IMP. The fake hosts include the local terminal, a diagnostic debugger, a module capable of loading the IMP′s core should it be damaged, and a module that gathers statistics and performs certain other tasks. They are discussed in section 2.2.6.

Section 2.2.7 discusses Very Distant Hosts which are used when the cable distance between a host and an IMP is greater than 2000 feet.

Finally, section 2.2.8 addresses IMP reliability issues.

2.2.1 Modem Input/Output

This section presents first the modem-to-IMP module M2I and the IMP-to-modem module I2M, and then discusses the strategy for determining the status of a link between IMPs. The data and attributes concerning any given modem are contained in an associated modem parameter block shared by both M2I and I2M which resides on the variables page. Each entry in the data base M2PBLK is a pointer to the parameter block for that modem.

<u>M2I</u>. The module that receives packets from the modem and passes them to the IMP is called M2I. The I/O device on Pluribus pokes M2I′s PID whenever a packet has been received. Additionally, M2I is poked every 25.6 milliseconds. The following activities are performed by M2I:

1. Assuming the current input has completed, input from the modem is started to a new buffer, using a double buffering scheme. This new buffer has been allocated as described in step 4 below.

2. A check is made for hardware errors. Any transmission error or checksum failure is detected here. Such errors are tabulated as a measure of the quality of the link.

3. A check is made to insure that the length of the packet is greater than the minimum (5 words) and is not greater than the maximum (71 words). If this check fails, a problem has been detected. Such an error might be caused by a loss of character sync in the modem interface.

4. Another buffer is obtained from the buffer pool. This buffer is used (see step 1 above) as the buffer to fill after completion of the next packet.

5. A software checksum is calculated. If this is incorrect, the packet is known to be in error.

6. If the packet contains an ACK field, this field is saved.

7. A dispatch is made on the packet type, as follows. Packet types 0 and 1 (data packets and control messages) are placed on the TASK queue. Type 2 packets (routing messages) are placed on the routing queue, unless they are null in which case they are discarded. (Such packets serve the special purpose of carrying ACKs and IHYs.) Packets of type 3 are packet reload messages and are placed on the packet core queue.

8. In all cases, on completion of the processing of packets to the TASK queue, as well as the packets which are null, the acknowledgements are processed.

<u>I2M</u>. The IMP-to-modem output module I2M has the job of transmitting assembled packets through the modem interface to an adjacent IMP. It is awakened by the hardware on completion of sending a packet, awakened by the software whenever a packet is found in TASK to be sent to the modem, and poked every 25.6 milliseconds. Its work is done as follows:

1. I2M determines whether the modem hardware is currently in use. If so, processing for this module is completed since it cannot do anything if the hardware is busy.

2. A check is made for hardware errors. If one has occurred, it is reported.

3. The next action depends on the type of the packet which has just been transmitted over the modem. A dispatch is performed as follows:

   - Routing Packet - A use count for the routing buffer is decremented.

   - Data Packet - If the flush switch is set, the packet last sent has been acknowledged while in transit and the buffer is freed. Otherwise the packet is saved on SENTQ for possible retransmission.

   - Null Packet - Certain timing actions are taken.

4. The algorithm determines the next packet to be sent. The order of importance (from most important to least important) is as follows: packet core messages and reload demands, routing packets, null packets which are sometimes required for sending an IHY or for sending ACKs when there is no data packet to carry them, retransmission of packets previously sent but unacknowledged for MRTIME*100 microseconds, and new data packets with priority data packets ahead of non-priority data packets. After selecting a packet, I2M sends it.

The above processing is performed by I2M on each entry.

Link Up/Down Status. Links between IMPs have failures which must be detected and compensated for by the software. There are four possible states of a link: up, down, up but not very reliable, or looped. The first consideration is whether the link is up or down and how both ends of the link can be in agreement about its status.

To be considered up, a link must be useful for transmission in both directions. It is therefore important that the link-up protocol insure that the IMPs at both ends of the link agree on the link's status. When a link is up, each IMP from time to time sends a "hello" message which is replied to with an "I heard you" (IHY) message. These packets must be exchanged periodically or link trouble is suspected. The "hello" message is a routing message, and the "I heard you" message is a null packet sent in reply.

When a link is down, each IMP spends a certain period of time in a link hold-down state in which it refuses to transmit anything at all over the link, thus insuring that the IMP at the other end also sees link trouble. (Certain modem failures can leave the link operable in one direction but not the other, and it is important that such links not be used at all.) It then enters a "coming up" state in which a protocol is used to bring up the link. The IMP sends "hello" messages and looks for IHY messages. When enough of these have been successfully transmitted and received, the IMP declares the link up and goes ahead and uses it. If the mechanism for bringing the link up fails, the IMP returns to the link hold-down state for an appropriate period.

If while the link is up there occurs an extended period without IHY messages, the IMP declares the link as going down and stops using it, immediately entering the hold-down state.

The algorithm for bringing a link up is based on an 8-bit state counter (LSTATE). At the beginning of the coming up phase, the state counter is set to a "half-count" value H. Table 1 shows the values of H for various line speeds used. The IMP

Speed (kbs)	H
4.8	16
7.2	16
9.6	16
48	64
50	64
120	64

Table 1
Half-count Values for Various Line Speeds

starts to send out "hello" messages. Each time it receives an IHY message, it decreases the counter by 1. If the "hello" message is not acknowledged, it increases the counter by 1. If the counter ever gets to twice its H value, the IMP declares the link dead and sets the counter to 128. This is the hold-down state and the IMP stays in that state until the counter reaches 141, (i.e., the line has been down for about 8 seconds), at which time it reverts to H. If IHY messages are received and the counter gets reduced to 4, it is left at 4 which is the "link up" state. If IHY messages are missed, the counter is decreased by 1. If the counter passes 0, the link is then declared as being down and the state is set to 129.

This algorithm insures that a link cannot be declared up unless both IMPs believe in it. A failure of transmission in either direction causes the link to be declared dead. It is the link hold-down state that guarantees this effect. For any line that has gone down, all pending packets are rerouted by resubmitting them to TASK. Any routes that were using this line are marked for maximum hops or delay, and hold-down is entered so that the "bad news" about this line will propagate to other IMPs in the network.

The IMPs determination of a missed IHY message (the basis for the above counting mechanism) is dependent upon a basic clock cycle. For a 50 kilobit land line the cycle is the slow TIMEOUT, 640 milliseconds. A three-second cycle is used for a 9.6 kilobit link. The algorithm is set for each link at the indicated clock rate.

2.2.2 Host Input/Output

This section presents the two modules that provide communication between the IMP and its local hosts. Module HI provides communication from a host to the IMP, and IH from an IMP to a host. The data concerning a given host are kept in a single host parameter block. Each host parameter block contains a variable FAKE which has four states: real host, fake host, background host, or VDH. The parameter block for a real host is a 56-word structure shared by HI and IH. Most of the entries in this block are variables reflecting some aspect of the state of the connection to the host. The host parameter block also contains about 10 words of temporary storage. A single data base in the IMP called H2PBLK contains an entry for each host; the entry being a pointer to the parameter block for that host. All host parameter blocks reside on the variables page.

HI. In Figure 7, State W is a state in which the IMP is waiting for a leader. The hardware has been initialized to read data from the host interface into a transaction block. This state may persist for arbitrarily long periods of time, since the host may not be generating net traffic. It is for this reason that transaction blocks do not time out as do other kinds of buffers.

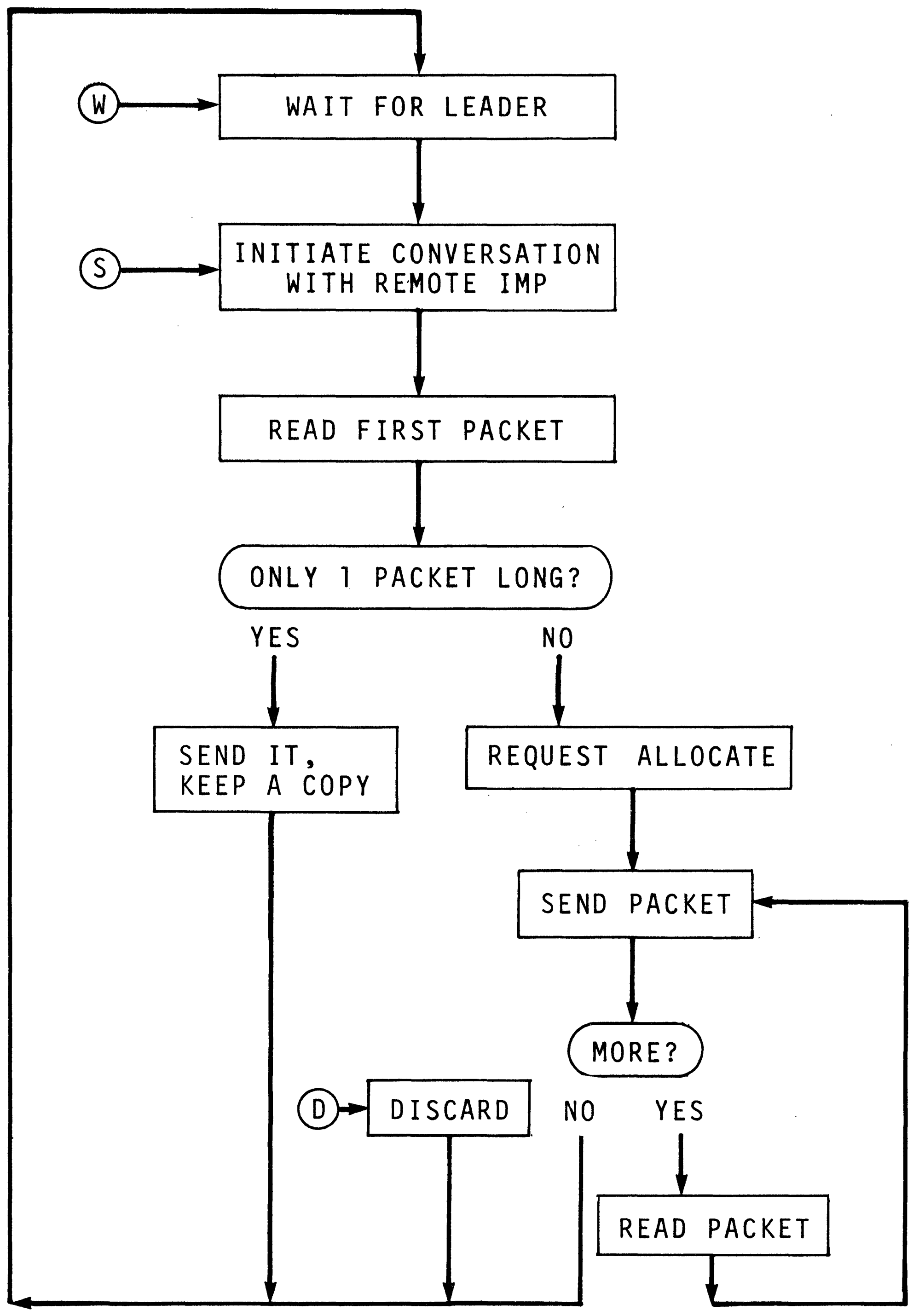


Figure 7. States of Module HI

If the source host's parameter block indicates that the host uses old leader format, HI makes the conversion to new format.

A transmit message block (TM block) is a block of data in the IMP which keeps track of the status of a given conversation or message stream. There is in the receiving IMP a receive message block (RM block) which records the status of that same conversation from the receiver's point of view.

State S is entered when the hardware signals that the leader has been read into the transaction block. (The host interface is blocked as soon as this has happened.) Upon examination of the leader, HI uses the subroutine MESGET to initiate a conversation. MESGET sets up a TM block at the source host, and sends a message to the destination host requesting that the conversation be initiated. The destination host then sets up a RM block. When the RM block has been acknowledged, MESGET gets a message number, cycling through 256 possible numbers. It may enter a wait state until a free message number is available. Normally, the conversation is already open and MESGET simply returns the next available message number.

Each packet sent from the source IMP to the destination IMP holds a pointer to the RM block at the destination, and each acknowledgement from the destination to the source contains a pointer to the TM block at the source. The pointer is an 8-bit index indicating which RM block or TM block is referred to. The receiver of a message always verifies that the block pointed to is appropriate for the message by checking the consistency of the identity of the IMP at the far end.

The method for sending a multipacket message is presented first, since the one packet message is a special case and is discussed in section 2.2.3. The system proceeds as follows:

1. It must be determined whether or not the destination IMP has space to receive the message. In some circumstances (explained in 4 below), the IMP already knows that such space is available. If not, HI sends a message to the destination IMP requesting space for an eight-packet message. The same TM and RM blocks are used but a new transaction block is needed as well as a new message number. The IMP must then wait until it receives the allocation.

2. Upon receipt of the allocation, the packets of the message are sent out sequentially.

3. After all packets of the message are received at the destination IMP, the message is acknowledged with a RFNM. The RFNM is generally accompanied by an allocate of space for another eight-packet message, providing the destination IMP has space for it.

4. If the source IMP receives a multipacket message soon enough from the host, it can be sent immediately using the allocate that was sent with the last RFNM.

5. If no new traffic comes from the host within a suitable period, the source IMP gives back the allocation with the GIVEBACK protocol message.

State D is entered to throw away the rest of the message if something has gone wrong. A single junk buffer is maintained in the IMP into which any hardware device can be directed to send its data. As no routine ever looks at this buffer, it can in fact be simultaneously a destination for more than one hardware device. This serves as a place to put incoming data which is known to be invalid.

If the IMP is unable to transmit a message to the destination or the message is lost in any way, the host is notified with a suitable IMP-host protocol message. If the leader in a message from the host to the IMP is invalid or has other problems, the host is notified immediately.

IH. The IMP-to-host routine IH takes messages from its input queue and sends each message to the appropriate host. It maintains a regular queue RQ as well as a priority queue PQ. IH sends to the host, in the order named, control messages (such as RFNM, destination dead, etc.), messages from the priority queue as long as there are any, and messages from the regular queue. Ordering of messages is not an issue since FORUS insures that messages appear on IH's queue in the proper order. IH changes the status of the message as recorded in the RM block to reflect either that the message has been transmitted successfully to the host or that for some reason the transmission failed, as for example if the host is dead. Having detected this change, Background Host 5 (as described in section 2.2.5.1) then sends to the transmitting IMP the appropriate RFNM, RFNM with Allocate, or other relevant control message.

The other task performed by IH is conversion to old leader format if necessary. If the host is marked (in the host parameter block) as using old leader format, IH makes the conversion just before sending the packet to the host. If the originating host is connected to an IMP with number greater than 63, or if that host's number on its IMP exceeds 3, then it is not possible to incorporate these data in the old leader format (the fields are not wide enough) and the two hosts cannot communicate. IH discards the message and returns the appropriate protocol message to the originator. This function is normally performed at connection SETUP time (i.e., GETABLOCK, GOTABLOCK).

2.2.3 TASK

Each packet received in the IMP from another IMP (or host) is dispatched to TASK, the central routine that decides what to do with it. There are two possibilities: either the packet is for a host at this IMP (FORUS), or it is to be stored and forwarded to another IMP (S/F).

Moving a packet towards its destination requires placing the buffer holding it on TASK's input queue, a queue of buffers waiting to be processed. Each buffer is self-contained in that the buffer header contains the data which tell TASK what to do. The input to TASK originates either from a modem or from one of the local hosts. Similarly, the destination is either a modem or a local host. The code in TASK which prepares a buffer for a local host is called FORUS. (Module IH also does part of this work.) S/F is that part of TASK that implements the packet switching aspect of the IMP. The next two subsections describe TASK. Section 2.2.3.1 describes TASK's basic loop and presents the details of the store-and-forwarding operation; section 2.2.3.2 describes the FORUS part of TASK that handles packets for a local IMP.

2.2.3.1 Store-and-Forward

The store-and-forward operation of TASK is to process buffers one at a time as they are found on the input queue, as follows:

1. The top packet is taken from the queue of packets waiting to be processed and TASK's PID is poked again so TASK will continue to run.

2. The destination of the packet is examined. If it is for the local IMP, the FORUS module is entered directly. (See the next section for further details.) Otherwise, processing continues.

3. The routing tables are examined to determine over which link (i.e., which modem) to send the packet. If no route is found, the destination IMP is dead; the packet is ACKed, flushed, and a trap is recorded. If there is inadequate store-and-forward buffer space, the packet is flushed; it will be retransmitted later by the source IMP. The channel tables for that modem are now examined. Packets which cannot be transmitted because of no available channel are placed on a separate auxiliary TASK queue which is periodically placed at the beginning of the TASK queue in anticipation of a channel becoming available later. In this way the packet is retried about five times as often as the source retransmits it.

4. The current packet has now been successfully received (in that the program is able to send it on to the next IMP) and must therefore be acknowledged. If the source of the packet is a local host, it is acknowledged by setting the TSKFOK flag in the host's parameter block. If the packet arrived from another IMP, the acknowledgement protocol is followed. Details are provided in section 2.1.4.

5. The packet is queued onto the relevant modem for output, either on a priority or a regular queue.

6. The proper modem output process is poked.

2.2.3.2 FORUS

The FORUS discussion which follows is keyed to the labels on the blocks in Figure 8.

In block W the code looks at the packet header to determine which RM or TM block is referred to. A brief check on the validity of the block is made to insure that it deals with a conversation with the IMP which originated the current packet. If this validity check is not passed, block G is entered. This need not indicate an error, since the GETABLOCK packet of necessity does not have an associated RM block. If this packet is in fact GETABLOCK, then the proper GOTABLOCK reply is constructed. If not, then it is possible that the remote IMP is sending some sort of protocol query packet. If this is the case, an appropriate reply is constructed; if not, the current packet is merely discarded.

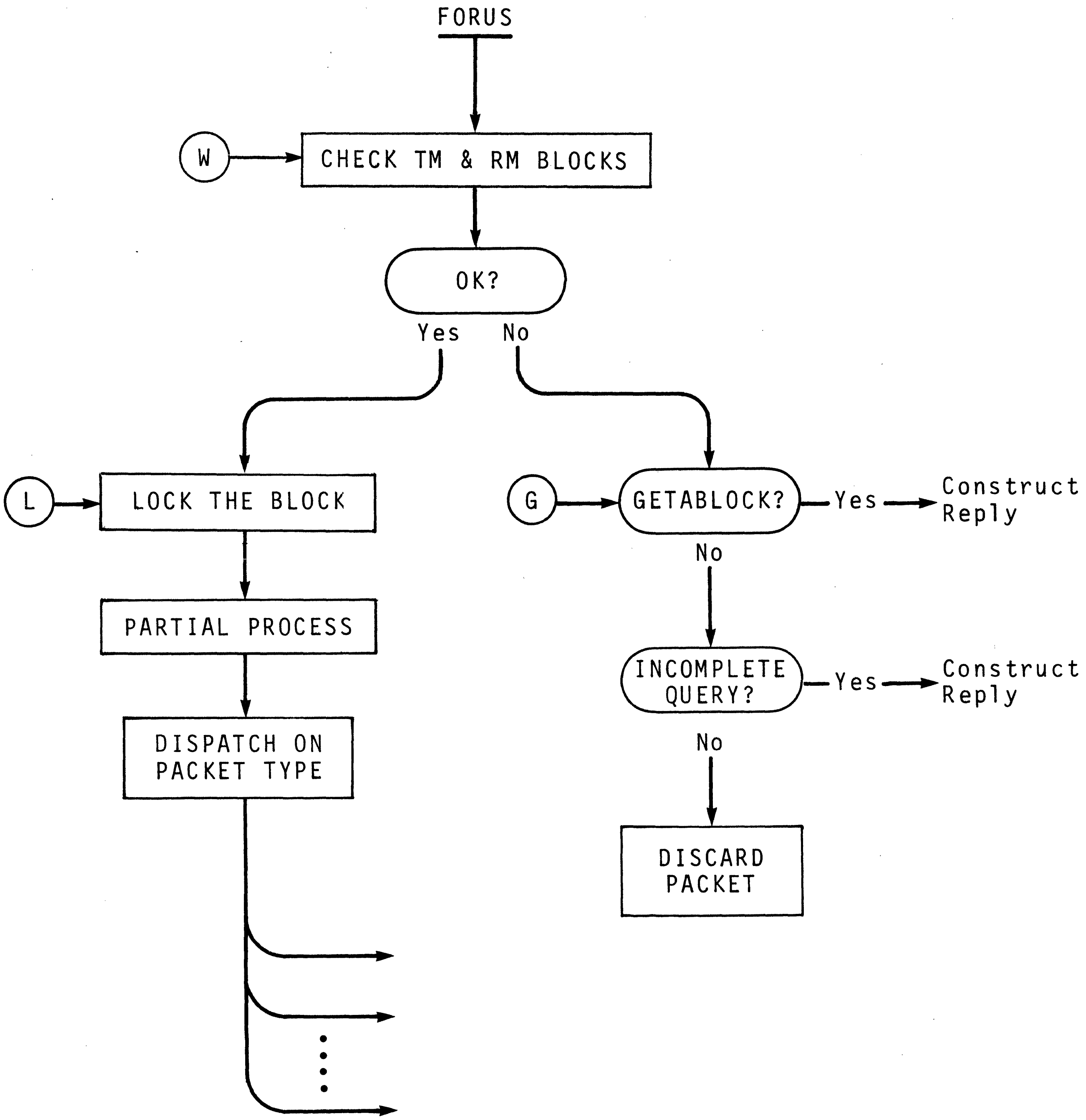


Figure 8. States of FORUS

Continuing with the successful processing of a correct packet, block L is entered. The TM or RM block is locked and then certain preliminary processing is performed which is common to several of the possible packet types. Finally, the program dispatches on the packet type to an appropriate routine. The packet types are as follows:

DATA PACKET. If the packet is part of a multi-packet message, it is expected and there is guaranteed to be adequate buffer space for it. FORUS searches for a reassembly block that is associated with (points to) the RM block. It is this reassembly block which in turn provides the mechanism for storing the buffer. When all of the packets of a message have arrived, FORUS looks at the RM block. If this message is the next one to be sent to the host, it is put immediately on IH′s input queue and the reassembly block is released. FORUS then looks to see if there is another complete message waiting to be sent to the host. If the present message is not the next one, indicating that an earlier message has not yet been received in its entirety, then the present message is merely left waiting until its turn arrives. Messages must be sent to the destination host in the same order they were emitted by the sending host.

REQUEST FOR ALLOCATION. If the allocation is for a multi-packet message, flags are set up which are looked at later by Background Host 5. Since the actual allocation is performed by this background host, FORUS is finished. If the request for allocate is for a one-packet message, the data have been transmitted as part of the request for allocation. If the IMP has room, it queues the message for the destination host and replies with a RFNM. If there is no space available, flags are set to inform Background Host 5 of the request. The data are discarded.

GIVEBACK. This serves to giveback an allocation. A reassembly block for this conversation is freed and the allocated buffers returned.

INCOMPLETE MESSAGE. When the source has detected a failure once the message number has been assigned, it is necessary to free that message number. FORUS tells the destination to free up any ? In? pending fragment for this message and an Incomplete Reply is queued.

INCOMPLETE QUERY. This is an attempt to find out what happened to a message number that has not been acknowledged. If the message number in question is out of range, an Out-of-Range reply is sent. In the case where the message number is within the last eight messages and the reply state is idle, a duplicate reply is sent. If the reply state is not idle, an Out-of-Range reply is sent. When the message number is in the current message window, FORUS performs a cleanup of reassembly resources and marks the reply state for an Incomplete Reply.

GETABLOCK. Although the flow chart of Figure 8 suggests that box G intercepts this kind of message, this is not always the case. If the block number field in the GETABLOCK message happened to refer to a block which passed the relevant validity checks, control would have passed to block L as opposed to block G as previously described. Nothing bad has transpired and the dispatch in this case is to block G.

RESET. This resets a block that is no longer in use; a RESET REPLY is sent immediately.

RFNM. This indicates the successful receipt of the current message. The transmit message number is marked complete. If it was a real data message, FORUS queues a RFNM control message for the host.

RFNM WITH ALLOCATE. If this was sent upon receipt of the last packet of a multi-packet message, it indicates that the destination has adequate resources (eight buffers and a reassembly block) to receive another multi-packet message and contains an allocate. If it is the reply to a single-packet Request for Allocation, the saved copy of the message is retrieved and sent. If it was a real data message, FORUS queues a RFNM control message for the host.

DESTINATION DEAD. This indicates that the host addressed cannot receive the message. The message is thrown away. If it was a real data message, FORUS queues a RFNM control message for the host.

INCOMPLETE REPLY. This is a reply to an Incomplete Query or Incomplete Message. It indicates that the message was not successfully delivered and why. The condition codes in the transaction block are adjusted accordingly. If it was a real data message, FORUS queues a RFNM control message for the host.

OUT-OF-RANGE. This indicates a bad message number in reply to an Incomplete Query. A RESET message is sent immediately.

GOTABLOCK. This is a reply to the GETABLOCK message. The TM block state is appropriately modified to open the conversation.

A sub-case GOTNOBLOCK, which is indicated by status bits in the header, results in the local host being notified of the dead destination.

RESET REQUEST. This is the destination IMP's way to initiate a reset of a conversation. It is sent under certain timeout circumstances, as described in section 2.1.3.1. The source IMP may ignore the request if the conversation has just become active again.

RESET REPLY. This is the destination IMP's way of replying to a RESET. It indicates that it has reset its end of the conversation and freed the RM block. FORUS can now mark the TM block idle.

2.2.4 Routing Algorithm

The routing algorithm is identical in both the Pluribus and 316 IMPs. It is necessary that both use identical algorithms since they exchange routing information and no IMP knows the machine type of its neighbor.

The basic algorithm, as implemented by the Pluribus IMP, consists of a strip; each execution of the strip processes one entry of a routing message. The strip may be poked by M2I upon receipt of a routing message, slow TIMEOUT every 640 milliseconds, or itself when there are more entries of the routing message to process. A state word indicates whether or not a routing message is currently being processed. When the strip is entered, it proceeds as follows:

1. The routing lock is locked.

2. If there are additional entries to be processed for the current routing message, the routing strip pokes itself. If there are no more entries to be processed, the routing message queue is examined. If it is empty, there are no further incoming routing messages to deal with and the strip constructs the IMP's new routing message to be sent; if it is not empty, the top message from the routing queue is taken off and that message is marked as the current routing message to process. The counter of entries is initialized to the first entry of the routing message.

3. The current routing message requires processing, so the next IMP number in the routing table to be processed is noted. The PID level for this process is poked.

4. The routing lock (set in step 1) is unlocked so that another processor (if available) can start executing this same strip.

5. The delay and hop calculation for this IMP are calculated as described below. The routing calculation for each node is independent of that for all others, so several processors may be executing this phase of the operation in parallel.

6. The routing table is locked, the answers calculated above are stored, and the routing table is unlocked.

The routing table consists of three entries for every IMP in the network. The first entry is a count of the number of links on the shortest path from this IMP to that one. This is one greater than the number of intermediate IMPs through which a message must pass to reach the destination. The second field is the so-called "delay" field. This is an approximate measure of the delay a packet can expect when traveling through the network to the destination. The third entry specifies the modem which should be used to transmit a packet to the destination IMP.

Each IMP periodically sends to each of its neighbors a copy of its routing table. It therefore follows that each IMP periodically receives a copy of the routing data as perceived by each of its neighbors. In this way, any change in the network's situation is gradually propagated throughout the entire network.

It should be noted that only the first two of the three routing table entries mentioned above are actually transmitted, since only the number of hops to the destination and the delay expected are of interest. (Modem information is unique to each IMP.) On receipt of a routing packet, the following processing is performed for each IMP in the network; i.e., for each line of the routing table.

1. The IMP's count of the minimum hops to the destination is compared with the count received in the routing packet. If the IMP's count is less than the received-count-plus-1, no change is made; if it is greater, the IMP concludes that the sending IMP knows a better way to the destination and sets its new count to

the received-count-plus-1. On completion of this calculation, the IMP's count represents the shortest route to the destination.

2. The received delay value is examined. The IMP calculates N, such that N = 4 + the local delay on the link to the sending IMP. (The local delay is the number of channels, between 0 and 127, currently in use between the IMP and the sending IMP.) If the IMP's delay entry is less than N, no change is made to the routing table; otherwise, N becomes the new delay. At the conclusion of this calculation, the table entry for delay to the relevant IMP is the shortest expected delay.

3. If the delay value was decreased by entering the value N based on the routing message just received, the IMP updates the table of modem links to use. This insures that for this destination, the IMP will use the link over which it has just received the routing message.

When an IMP notes that a link goes down (see section 2.2.1), it is unable to tell whether it is the link or the IMP at the far end that has failed. In either case, it reacts by setting both the minimum hop length and the delay time to very large values in the routing table for the IMP at the other end of the link.

Difficulty arises because information about an IMP or a link going down propagates extremely slowly through the network, and there can be very bad effects while the message is propagating. Through use of a technique called "hold-down" as previously described in section 2.2.1, the IMPs delay the route changeover process for a few seconds and in this way permit a faster and smoother cutover. When the best route is about to change, the IMP first makes sure that the neighboring IMPs know that the old route has gone bad before it attempts to change; this strategy prevents the adjacent IMPs from slowing down the process by transmitting old information.

Each IMP measures the bandwidth and loading of each of the circuits to which it is connected, sending routing proportionately more often on faster links. Thus, the percentage of link bandwidth used for routing varies between 3% and 15%, approximately, as a function of link use.

If dead links eliminate all routes between two IMPs, the IMPs are said to be disconnected and each discards messages destined for the other. As disconnected IMPs cannot be rapidly detected from the delay estimates that arrive from neighboring IMPs, the hop count is maintained as well. If this count ever exceeds the maximum expected number of network nodes, the destination IMP is assumed to be unreachable and therefore disconnected.

2.2.5 Background Hosts

Certain IMP activities relating to end-to-end message processing must be performed periodically. When control messages require reserving resources, or timing out idle resources, TASK itself cannot perform the function. The background hosts were created for these resource-reserving and freeing functions.

A background host is a process, complete with PID levels, which runs periodically, examining some data base. When necessary, it generates an appropriate message for transmission over the network. The use of background hosts makes it possible for the originating processes, which are not in a position for one reason or another to generate a message, to be able to arrange internal software states so that the message will ultimately be sent. Since the background hosts are only originators of messages and never destinations, they do not have host numbers in the local IMP. Each background host is poked by fast TIMEOUT every 25.6 milliseconds, and some are poked more often as needed by routines which have work for them.

Each background host has a parameter block which is very similar to a real host parameter block but about half the size. A background host uses its host parameter block to communicate with an IMP in a manner similar to the HI software. The background host simulates the host-to-IMP part of the 1822 interface.

The individual background hosts are now described.

2.2.5.1 Background Host 5

BACK5 is the background process that sends RFNMs, allocates, Destination Deads, and Incomplete Replies. BACK5 scans all of the RM blocks continuously, starting one after the last block serviced (for fairness). If it finds an RM block whose state is "need an allocation of 8," it looks to see if eight buffers are available. It also gets a reassembly block for the message. If one is available, and the eight buffers are available, it sends the requested allocate.

If BACK5 finds an RM block whose state is "need an allocate of 1," it attempts to find one buffer and a reassembly block and sends an allocate-1. This state can only occur if a one-packet request was received and the IMP did not have space to accept the message at that time (as described in section 2.1.2).

BACK5 also sends RFNMs for messages whose packets have all arrived. If space is available and the message was a multi-packet message, BACK5 instead sends a RFNM with Allocate.

Upon receipt of an Incomplete Query or an Incomplete Message, BACK5 responds with an Incomplete Reply which indicates that the message was not successfully delivered and the reason why. In addition, BACK5 is capable of sending a Destination Dead which indicates that the destination host is unable to receive the message. In this case, the undeliverable message is thrown away.

If BACK5 is processing a Request for Allocation and the space required is not available, it waits one-half second in an attempt to get the space. It does this by going to sleep, knowing that it will be reawakened by TIMEOUT every 25.6 milliseconds. After a half second it gives up and proceeds to the next RM block. It is appropriate for BACK5 to make such a strong attempt to supply the allocate, since the originating host at the source IMP is blocked by the host-to-IMP interface.

BACK5 operates by making up a message in its own work area. It then gets a buffer (waiting as long as necessary for one), and uses that buffer to send the message.

2.2.5.2 Background Host 6

If a packet or a RFNM gets lost in the network, there is an outstanding message number and BACK7 can never free the conversation; the transmitting host is prohibited from proceeding ahead by more than seven additional message numbers (see section 2.1.3). BACK6 solves this problem by scanning through all of the TM blocks, looking at the incomplete timer. This timer is held off by any progress on message numbers. If the timer reaches zero, BACK6 locates the relevant transaction block and sends an Incomplete Query message. This message, which is accompanied by all data relevant to the conversation, asks the destination IMP what it knows about this message number. If the destination has received that message, it sends a duplicate response. If the message is only partially received, a packet has been lost and, therefore, the message is lost. The destination IMP sends an Incomplete Reply and the message number is then considered complete.

2.2.5.3 Background Host 7

BACK7 computes AGE "clips" beyond which to reset RM and TM blocks, based on how many free blocks there are left (i.e., how busy the IMP is in terms of the number of active conversations). RESETS (for TM blocks) or RESET REQUESTs (for RM blocks) are sent for any blocks that have reached the corresponding clip.

BACK7 proceeds by first scanning all the TM blocks, counting blocks which are free or in the process of being reset, and computing the associated age clips accordingly. It then scans through all of the TM blocks a second time. A consistency check is made on each TM block to see if it looks broken. The AGE field is examined to determine whether the block is old enough to discard. If the conversation appears to be quiescent (the TM block has reached its clip), a RESET is sent to the destination. The destination IMP then frees the relevant RM block and responds with a RESET REPLY. The source IMP can then free the TM block. This reset protocol is never entered if there are outstanding message numbers or message numbers to be returned.

BACK7 then scans all of the RM blocks to determine the AGE clips in terms of received conversations.

It makes a second scan of all the RM blocks to see which ones are too old. For each such conversation, it sends a "let′s terminate" message (RESET REQUEST) to the source IMP. At that IMP, FORUS sets the AGE of the associated TM block to be very high, so that BACK7 on the source IMP will ultimately initiate the freeing operation described above.

If no more RESETs or RESET REQUESTs need to be sent, BACK7 waits 640 milliseconds before trying again (since this is how often the TM or RM blocks can be aged).

2.2.5.4 Background Host 9

When a conversation has ended, it is necessary to perform certain clean up operations and free all blocks no longer needed.

BACK9 scans all of the TM blocks. Each TM block has an AGE field as described in the previous section which is set to 4 whenever the block is used. A timer increments AGE in a non-linear fashion, at a slower rate as it gets older. A value of AGE that is too high is an indication that the conversation has lapsed into disuse. If, upon examining AGE, BACK9 determines that the TM block is too old, all the buffers are given back.

BACK9 must free storage allocation held by the source IMP if they are no longer needed. An allocate timer in the TM block runs out in 250 milliseconds but is held off by use of the allocate by a host. BACK9 sends a GIVEBACK when three or more allocates are being held no matter the state of the timer. This can happen if the transmitting IMP has started the transmission of several multi-packet messages before the acknowledgement of the first one is received, since each of these multi-packet messages can produce a RFNM with Allocate. When three allocates accumulate in the transmitting IMP, it sends a GIVEBACK to the destination IMP.

Each GIVEBACK has a message number and its reply is a RFNM without an allocate. When the destination IMP receives a GIVEBACK, it releases the relevant reassembly block and the buffers.

2.2.6 Fake Hosts

The IMP contains within itself certain "fake hosts," software modules which simulate many of the functions of real hosts in that they accept or produce messages through simulated 1822 interfaces. Each fake host consists of two processes; one for IMP-to-host messages and one for host-to-IMP messages.

Communication with fake hosts is very similar to communication with a real host. Each fake host has an 8-word block in memory formatted very much like the I/O block for a real host hardware interface. The difference, of course, is that writing into a hardware I/O block causes the I/O hardware to take some action, whereas writing into a fake host's block has no immediate effect. Thus a program which wants a fake host to do something must first write into the fake host's communicatons block and then poke the relevant PID.

The fake hosts are referred to as Fake Hosts 0 through 3. In net traffic they are addressed as hosts 252 to 255, the largest host numbers available in the 8-bit host field. These fake host numbers and their functions are the same in both the Pluribus and 316 IMPs for compatibility of operation over the network.

The four fake hosts are now described.

2.2.6.1 Fake Host 0: Local Terminal

Fake Host 0 (host 252) is responsible for communications between the IMP and the teletype (or other terminal) connected to the IMP. Characters are taken from the teletype input buffer, passed through the fake host interface, and sent as messages (one character per message) to the current crosspatch destination. Messages destined for the TTY fake host are accepted one word at a time by the fake host interface. Characters are then passed in order, 8 bits at a time, to the teletype handler to be output on the IMP's terminal.

If a semicolon is read, the fake host uses a separate leader and sends characters as a single message until a second semicolon is read (so-called "semicolon message"). When the IMP is initialized, or if a NUL (code 80!, CTL-@) is typed and sent, the crosspatch destination is reset to be the debugging process DDT (Fake Host 1 in the same IMP).

2.2.6.2 Fake Host 1: DDT

Fake Host 1 (host 253) is the diagnostic debugger DDT. Messages to Fake Host 1 are interpreted by DDT as debugging instructions. The DDT fake host-to-IMP process accepts characters from the DDT process and sends them to the originator of the last message to DDT from the network. Characters are sent as a single message until terminated by semicolons, which the DDT fake IMP to host process sends through DDT. This ensures that a multi-character response to a single DDT command is sent to the proper source. Conversely, DDT fake's IMP-to-host process reads messages from the network and passes them to DDT. As each message terminates, it sends a semicolon which, after DDT echoes it back, will cause the DDT fake host-to-IMP process to send a message. The normal mode of operation is for Fake Hosts 0 and 1 to be crosspatched so that typing on the console terminal controls DDT.

2.2.6.3 Fake Host 2: Packet Core

Fake Host 2 (host 254) is the source and destination for packet core transmissions. Its IMP-to-host process accepts packets which control the loading and dumping of core areas within the IMP. When converted to special packet core messages, they may be sent to a specified malfunctioning neighbor IMP.

Upon receiving packet core packets that have arrived from a neighbor IMP, the fake host-to-IMP process sends them as messages into the network. In addition, it checks the block transfer state to see if packet core is active in its own IMP and if so, gets a free buffer and polls the process that constructs packet core messages from the IMP.

See also section 3.7 on packet reload.

2.2.6.4 Fake Host 3: Statistics and Discard

Fake Host 3 (host 255) provides the following services to the IMP:

1. Periodic reports are provided to the NCC. Such reports must occur every minute to keep the NCC convinced that the site is still alive. The NCC host reports to the NCC operators excessive delay between such messages.

2. The IMP originates system throughput reports and sends them to the NCC.

3. Messages for discard. Fake Host 3 originates messages whose destination is Fake Host 3 within the same IMP. On receipt of such a message, a RFNM is returned and it is the receipt of this RFNM by Fake Host 3 which holds off the watch-dog timer since this is convincing evidence that most of the software paths within the IMP are working. Excessive time lapse without receipt of such a message is an indication that something is wrong within the IMP. Fake Host 3 is also frequently the destination for a message generator, and a real host is permitted to send to discard if it wishes.

4. Message generation occurs for measurements and debugging. The length, frequency, and destination address of these messages can be controlled as operator supplied parameters.

5. Certain diagnostic IMP and queue data are reported periodically to the NCC.

6. The TLOG process sends certain detailed reports periodically to the NCC. These include snapshots when anomalies are detected and data about the Pluribus hardware configuration.

Except for function 4, all of the above functions are run continually. The frequency and nature of the reporting are controlled by parameters which can be changed dynamically (after initialization).

2.2.7 Very Distant Hosts

In instances where a host is located more than 2000 feet from the IMP, connection is made by means of the standard modem interface hardware normally used for IMP-to-IMP communication. BBN Report No. 1822 contains a detailed description of the protocol used for this type of interface.

Briefly, the method used to assure successful IMP-to-host transfers is similar to that used for the IMP-to-IMP channels. Logical channels are used as described in section 2.1.4, although in this case only two channels are employed and the order of transmission is important. Therefore, both the host and the IMP software must be aware of packets. For example, assume packet A is transmitted from an IMP on channel 0, and packet B is then transmitted on channel 1. If an error were detected in packet A, but not B, no ACK would be returned for A. The host would retain packet B until A is retransmitted to it and received successfully, thus insuring delivery of the packets to its own processes in order A-B.

2.2.8 Reliability Mechanisms

Certain of the reliability mechanisms mentioned in section 2.1.6 are described in greater detail in this section. Addressed are buffers, counters, and crossed or looped queues.

2.2.8.1 Buffer Reliability

A major part of the IMP reliability system is buffer reliability since buffers serve such an important function in the IMP. Packets are collected in buffers and placed on appropriate queues. An important design principle is that the data in a buffer are never copied from one place to another. For example, a message received from a host in HI is read directly into a buffer by the I/O hardware. A pointer to that buffer is then placed on the queue to TASK. Subsequently, the hardware will be directed to output the contents of the buffer to a modem interface. The data are never moved around in memory; what is moved is a pointer to the data.

A queue of buffers is a linked list of pointers. The pointers are not kept in the buffers themselves but in a single vector on the second variables page. (The purpose of doing this is to preclude the need to change map registers while progressing through a queue, since the buffer in general resides on a different page from the variables.) An entry in a queue of buffers in the IMP, as for example the queue of buffers waiting for processing by TASK, is a word which contains the index in various tables of the data describing that buffer. The vector POINT contains the addresses of each buffer in the usual packed format for loading into map registers. (The left 7 bits contain the map setting of the buffer's memory page to be loaded directly into a map register and the right-most 9 bits when shifted left 4 contain the offset of the buffer in the page.) A parallel table to POINT is the table CHAIN. Each entry in CHAIN indicates either the end of a chain or the index of another buffer, as just described.

There are three more parallel tables, WHERE, FLUSHD, and CHAN, whose contents are dependent on which queue contains the buffer. They, like the CHAIN words, are other data associated with the buffer but kept on the second variables page to be readily available.

A buffer itself consists of 80 words. The first 8 are header information, the next 63 words are the data itself, the next word is unused, and the last 8 words are other variables associated with the buffer. One of these words points to the end of the real data in the buffer.

The vector WHERE is bit coded and contains use bits for each buffer. There is a use bit associated with each of the twelve possible buffer users, such as M2I, HI, etc. In some cases more than one use bit is on. A use bit being on indicates that the particular module has associated itself with that buffer. If a buffer is on the free list, all use bits are 0. There is also a 4-bit count field which indexes a vector of buffer accounting variables.

A new buffer is obtained by calling the subroutine FREGET, with two parameters. One parameter is a word containing the use bits to be stored into WHERE, and the other parameter is a COUNT. The subroutine FLUSH is called with a pointer to a buffer to return it to the free list. It also takes a use bit as a parameter, and issues a trap if the buffer in question does not have that use bit turned on. FLUSH turns the specified bit off. If no other bits are on for that buffer, the buffer is re-threaded onto the free list, the count specified by the WHERE vector is decremented, and FLUSHD is set to be non-zero. The presence of other bits indicates that one or more other routines is associated in some way with that buffer; in this case the buffer is not freed and FLUSH just returns.

The FLUSHD field is used for finding lost buffers. FLUSHD is set non-zero each time a buffer is freed. A periodic process looks at each buffer every two minutes and sets FLUSHD to zero for all buffers except those on the free list. If it finds a buffer zero after two minutes have passed, it assumes that the buffer has been lost and arbitrarily puts it on the free list. Any buffers chained after it are left hanging, to be picked up later by this same mechanism if they are truly lost.

2.2.8.2 Counters

To prevent various kinds of lockup, a counter mechanism has been devised to insure that certain processes are always able to get buffers, no matter how busy the IMP is. Associated with each process, or collection of allied processes, is a count of the number of buffers guaranteed to that process. Also, a counter is maintained of how many buffers that process has allocated to it at any instant. If a process needs a buffer and the number of buffers currently allocated to it is less than its guaranteed count, it immediately gets a buffer. (The algorithm insures that in such a case the buffer is available.) If it already has more buffers than its guarantee, it is given a buffer only if there are enough remaining in the overflow pool of available buffers. There are buffers in the overflow pool only when the number of buffers in the free list exceeds the number required to meet all guarantees. The size of the overflow pool is equal to this excess. The IMP will continue to operate even if this pool is always empty.

These guarantee values are recalculated from time to time during the operation of the system. Some counts change at times, as for example when a host or a modem line comes up or down.

It is important that the system maintain a count of the number of buffers owned by each process. If the software cooperates, this is not difficult. However, when the buffer reliability code (discussed in the preceding section) forcibly places a buffer onto the free list; it is an indication that something has gone wrong and all the counts must be properly adjusted. Since buffers are continually being shuffled around, it is impossible to stop all use of buffers and scan them to determine which ones are in use and by whom. The reliability code therefore maintains the usage counts while the system is running.

The reliability code first initializes a new estimate of all the counters and new minimum guarantees based on the current state of hosts, modems, etc. It then scans all of the buffers, calculating all of the counts from the WHERE words. This takes several strip times and counts may be changing. Next it calculates the error between the counts stored in the system and the counts which it just determined. A new error is calculated to be 3/4 of the old error plus 1/4 of the new error. Should the new error represent more than one buffer, the corresponding system count is adjusted and the accumulated error is set to zero. This algorithm effects an exponential smoothing of the system counts towards their proper values.

2.2.8.3 Crossed or Looped Queues

Two serious problems which might arise are the appearance of a buffer on more than one queue and a loop in a queue. When a module (such as TASK or IH) takes the next buffer from its input queue, it makes several checks. One of these is to insure that the buffer is owned by this process, i.e., it has the correct ownership bit set. If not, the buffer is ignored and the queue it came from is made empty. This mechanism detects joined queues and breaks a loop in a queue, since ultimately a process finds (as its input) a buffer which it is not supposed to own.

When a bad buffer is detected in this way, it and the buffers to which it is chained are ignored since they are found in the two-minute timeout and returned to the free list if no one else points to them. The variable FREE points to the head of the free list and FREEND points to the last buffer on it. A buffer that is being freed is tacked on to the end of the free list, thus guaranteeing that all buffers are used over a period of time. Since buffers on the free list have no use bits set, the free list is scanned periodically to validate its structure.

Chapter 3
The STAGE System

The STAGE system consists of a set of eleven modules (stages) that allow each processor to determine the status and current configuration of the Pluribus hardware. The first section in this chapter presents an overview of the STAGE system, next the data bases are described, and then the individual stages are presented in detail. Finally, two sections address the Block Transfer routine (BLT) and the Packet Reload mechanism.

3.1 Introduction and Overview

STAGE's basic purpose is to insure that the required hardware is available to the application program. To the extent that more hardware becomes available, STAGE discovers it and makes it known to the application program. In the event of a failure causing certain hardware resources to disappear, STAGE discovers that fact and reconfigures as necessary.

The STAGE system is run in each processor under two quite different circumstances. First, STAGE is run at startup time and at any point at which a sufficiently serious failure condition arises to make it necessary for a processor to restart the checklist operation. Generally, each stage assumes the successful execution of all preceding stages and application programs cannot be run until all stages have been completed. Second, even in a smoothly running system, each processor checks the time as part of the dispatch loop and schedules STAGE at regular intervals.

Thus, at startup time, the STAGE system serves as an initialization mechanism whose function is to find out what resources are available and to make them known to the application. STAGE is also run as a low-priority task in the running system to determine whether anything has changed or gone wrong.

The hardware configuration may appear different when viewed from different processors, since many processor or bus failures may only affect certain processors or specific processor-resource pairs. To handle this possibility, the STAGE system must also maintain "consensus" information that insures that the processors interact when attempting to determine the machine configuration before taking unilateral action.

3.2 Interconnection of STAGE Modules

Throughout the operation of each of the STAGE modules there are a number of common routines and data structures that handle the sequencing of the various stages, interprocessor control and strip-time discipline, and the management of consensus checking and vote-taking. This section describes a number of the structures and routines that affect the operation of the entire STAGE system rather than being specific to an individual module.

3.2.1 Sequencing the STAGE Modules

The operation of STAGE for each processor is controlled by a variable local to each processor called WDIS. WDIS is used to inhibit certain stages from running and keep track of the progress of STAGE. Each stage is assigned a bit position (Stage LK is bit 0, and so forth), and the system may only run individual STAGE modules if the corresponding bit in word WDIS is a 0. Thus, at restart time, WDIS is set to -2 (FFFE!) to indicate that only Stage LK may run at that point. As each stage successfully runs to completion, it clears the next bit in WDIS to allow the STAGE processing to proceed. If some stage discovers trouble at some point, it can force the processor to hang in a particular stage by setting the bits for all future stages and waiting until the offending condition is cleared or until enough processors have run this stage to agree on a course of action. Two routines are provided for the STAGE modules to perform these functions: SOKAY enables the next stage while SBAD forces the processor to hang in the current stage. SOKAY initializes the next dispatch if it actually turns off a bit. The actual stage in progress for a processor is contained in the variable WSTAGE, which need not be the highest stage enabled by WDIS.

3.2.2 Interprocessor Control and Strip Timing

In order to break up the operation of stage into strip-sized chunks, STAGE contains its own scheduler routine which operates in much the same manner as the main PID-driven LOOP code. Since the STAGE system does not discover the PIDs until Stage BD, an alternative mechanism is required, which is coded as WSLEEP. At the completion of each stage, the stage routine terminates by calling WSLEEP which (1) checks to see if the system is runnable (all stages are enabled in WDIS) and, if so, returns to the main loop, and (2) if Stage RC is enabled (but not the system), calls the Block Transfer (BLT) process. Unlike the main loop, WSLEEP preserves the current stage of the computation by saving all registers in a special register block for each stage local to the processor, so that computation for the STAGE system will proceed from where it left off whenever the STAGE system is reentered. If the system is not enabled, control flows into SJ6, and the STAGE system continues to run. Ordinarily, the next stage indicated by the previous pass through the STAGE loop is run at this point. If the next stage is disabled, the last-called stage is called again (and perhaps BLT), although a timer is maintained to restart Stage LK to insure that all stages are scheduled from time to time. Thus, when the processor is verifying its environment, its attention is concentrated on the highest-level stage that is enabled, although other stages will not be totally neglected.

There are a number of entry points to the STAGE system from the rest of the system. In particular, the main loop enters STAGE at SJ6 when the timer for running STAGE has elapsed. In the case of most error conditions, the entry point WST is used, which is generally called with a JSB R4,WST. Register R4 is saved in the variable UWST and serves as an indication of why STAGE was restarted.

3.2.3 Consensus Words

To prevent one processor from taking unilateral action which may turn out to be wrong, most stages operate under the control of a consensus which keeps track of the identity of all processors participating in the decisions. For each decision to be made, the processors in the consensus then cast their votes into a FIXIT word in common memory and action depends on the joint decision (either unanimity or majority vote may be required, depending on circumstances). The consensus for each stage is maintained on either the special communications page which is determined during Stage MD and is updated as part of the standard exit from STAGE (WSLEEP), or on the reliability kernel page (Stage RK). All stages except LK make use of consensus decisions.

The consensus itself consists of three words in memory as shown below in Figure 9.

```
+----------------------+
!  smoothed consensus  !
+----------------------+
!    next consensus    !
+----------------------+
! time for next update !
+----------------------+
```

Figure 9
Consensus Words

Each processor is assigned one bit position in the consensus word which indicates that the processor is participating if the given bit is on. The smoothed consensus word is used as the basis for all decisions made with respect to a particular consensus, while the next consensus word is used to compute any change in the consensus membership. Periodically (as the timer in the third word of the consensus elapses) the next consensus is copied into the smoothed consensus word and the next consensus word is set to the bit of the active processor. To join a consensus, a processor IORs its bit into the next consensus word. Since more than one processor may try to update this simultaneously, the next consensus is in fact used as a lock (note that the word should never be zero, since it was set to at least the bit corresponding to the processor that last updated the smoothed consensus). The processor must thus continue to IOR its bit at least as often as the update rate.

3.2.4 FIXIT Words

The FIXIT words are handled in a similar manner. Whenever a processor decides that some corrective action is required for a resource, it may call one of two STAGE subroutines, SFIXIT or SFXBAD, with respect to the particular FIXIT word. If SFIXIT is called and the FIXIT word (relative to the smoothed consensus word) gives that processor the authority to make the change, then the action is taken. Otherwise, SFIXIT includes this processor's bit in the FIXIT word. Note that the processor who acquires authority does not place its bit into the FIXIT word; this implicitly gives it a lock on the corrective action routine. If the SFXBAD routine is called, it disables the following stages (via SBAD) and performs SFIXIT. Two additional routines are provided to manipulate the FIXIT words, SCLEAR and SCLROK. SCLEAR removes the active processor's bit from the FIXIT word and SCLROK clears the processor's bit as well as enabling the next stage (see SOKAY).

3.3 Data Bases

All memory in the Pluribus is divided into two 4K-word pages. An important aspect of STAGE's operation is maintenance of the pages of common memory that hold the code and data for the application program and for STAGE. The next section describes the data that are stored at the bottom of every memory.

3.3.1 Page Types

There are five kinds of pages in a Pluribus IMP. Each application requires pages of types 1 and 2, and the application cannot run unless the required number of such pages is available. Since both kinds are required, the issue of order of importance between these two is not relevant. Otherwise, the page types are presented in the order of decreasing importance. The five page types are as follows:

1. <u>Code page</u>. A code page contains code to be run; it is checksummed and never altered. (That is, all the code is pure procedure.) In some cases, part of the page is used for variables local to the code on that page, in which case only the code part is checksummed.

2. <u>Required variables</u>. These pages contain variables which are used by the application. The important distinction between a code page and a required variables page is that the latter can be created from scratch, while a copy of the code is required in order to recreate a code page.

3. <u>Desired variables page</u>. Such a page is of some use to the application in that although the application runs better with the page, it is nonetheless able to run without it. If adequate memory is available, it is used for desired variables in preference to spare code pages,

since this strategy helps all the time while spare code pages are of assistance only in recovering more rapidly in case of certain failures.

4. Spare code page. A spare code page contains an extra copy of a code page. To the extent that there is enough memory, STAGE attempts to keep an extra copy of each code page so that the code can be recreated from the spare copy should it be destroyed. Equivalently, a smashed spare copy can be recreated from the code page.

5. Optional variables. Some applications desire optional variable pages for purposes such as extra buffers. To the extent that space is available, such pages are allocated. For example, an application requiring buffer space presumably runs more efficiently if extra buffers are allocated above the minimum.

The communications page is the lowest numbered page which can be seen by all operable processors. It holds all consensus data and various other data required by the earliest stages.

3.3.2 Page Format

Not all of the words on each page are available for application program use, the first 192 (C0!) words being reserved for use by STAGE. For convenience, page addresses are referred to in the following discussion as if they were referenced through map 0 and thus have addresses from 4000! through 5FFF!. Addresses 4000! through 40BF! are reserved for system use as described below. Each is preceded by the name which it has in the code.

SMDBUC	This is a bucket into which a store may be freely made. Stage MD uses it to write into to see if the page exists. (Writing into a non-existent page results in a QUIT.)
WMLOCK	This word is used as a lock by Stage MD. It interlocks the next few words which are used for a memory test. It is necessary to insure that not more than one processor at a time is testing a given page.
SMDBLK	This block of 8 bytes is the place into which the memory test is performed in Stage MD.

[illegible]

[illegible]

SLFPTR	This is a pointer to this page. It is maintained by the STAGE system and used by application programs to determine the contents of map registers 0, 1, or 2. The technique is described at the end of this section.
SLFLK	This word is a locked copy of the previous word, with 2 in the rightmost bits. It is used with the mechanism described at the end of this section to determine the contents of map 3.
COMPTR	This is the page number of the current communication page.
COMTST	This timer word is used in a special way as a consensus in Stage MD for repairing COMPTR if it has an improper value. See the description of Stage MD.

The words discussed above are used on every page in the system; the next few words are used only on the communication page. However, since any page may, without warning, become the communication page, space for these words is reserved on every page. These variables are maintained and kept up to date on the communication page only.

SYTIME	This is the system time and is updated every 25.6 milliseconds. The updating is performed while the application is running by fast TIMEOUT. During system startup time, STAGE updates it by monitoring the reading from the real time clock (RTC).
SEGCON	This is a 3-word consensus area for Stage MD.
SEGFIX	This is the FIXIT word for Stage MD.
MEMSEG	This block of words contains the bit table of existing pages. The table is created and checked for correctness in Stage MD.
MEMTOT	This contains the total number of pages of memory available; it is maintained by Stage MD.
STGCON	This block of 3 words is used for the consensus for Stage RK.
STGFIX	This is the FIXIT word for Stage RK.

COMREL This is the address of the page in common memory containing reliability code.

The following words are used by every page and constitute its own housekeeping area.

CKSFIX This is the FIXIT word for Stage MC and indicates processors which want to fix the checksum for this page.

INTIME This is the initialization timer held by TIMEOUT. Its use is dependent on the page type and the application.

CKSUM This is the page's checksum.

TLIMIT This is the upper limit for checksumming. It is the address of the first location which does not participate in the checksum. TLIMIT itself is the first word to checksum.

PGINIT This is the address of the initialization routine for this page, or zero if none exists.

TOPNTR This is a pointer to the configuration/timeout table, or zero if there is no table.

TYPE4K This is the page type of this page, the types being described in section 3.3.1.

This completes the description of variables appearing on every page. All succeeding words are available to the application program.

The words SLFPTR and SLFLK are used to determine the contents of map registers since it is a property of the Pluribus hardware that map registers cannot be read. The convention is that every page has in location SLFPTR the value that must be loaded into a map register to address that page. (This location is maintained by Stage MD.) Thus the processor may deduce the map contents by reading that location through the map.

This simple mechanism cannot be used safely to read map 3, since attempting to do so clears the word to zero. The processor could restore the location to its previous value on the next instruction, but another processor might happen to access the word while it is incorrectly zero. Therefore, SLFLK is maintained in the same format as SLFPTR but with an extra bit (the "2" bit) at the right end of the word. (That bit is ignored if loaded into a map register.) The usual lock discipline is used, accessing the word repeatedly until it is non-zero, and the value read is immediately stored back into the location. The purpose of the extra bit is to insure that the proper value is non-zero, since otherwise page zero would have zero in SLFLK.

3.4 Interrupt Routines

Although interrupts are not used in Pluribus in dealing with input/output, they are nonetheless a necessary part of the operation. A block of cells at the bottom of local memory contains a data block for each of the six possible interrupt types. This block specifies the destination transfer location when the interrupt takes place and also provides room in which the hardware stores certain status information at the time of the interrupt, such as the contents of relevant registers. The five interrupt types (the sixth is not used) are QUIT, ILLOP, the clock interrupt JIFFY, the remote power failure interrupt, and the paper tape reader (PTR); these five are now discussed.

3.4.1 QUIT

A QUIT is an event triggered by Pluribus hardware for various reasons. If a processor attempts to access a memory location which does not exist, a QUIT is generated. If the memory location is not recognized by any of the bus coupler units on the processor bus, the QUIT is generated by the arbiter. If one of the bus coupler units recognizes the address but the memory bus is unable to return a valid content, then the arbiter for that bus generates the QUIT. In any case, the QUIT is generated by an attempt to access an address that does not exist in the available memory as seen by the processor. QUITs are also generated by the I/O system for certain relevant events. It is possible that a QUIT may occur while in the QUIT handler; this is an indication that something is seriously amiss. Since the QUIT interrupt cannot be suppressed, the QUIT handler very early sets a flag saying "I am in QUIT handler." This flag is cleared at the end of processing a QUIT. If another QUIT takes place while this flag is set, the effect is as for any unexpected QUIT.

A QUIT that occurs unexpectedly is an indication of a problem in the system and causes STAGE to be restarted. However, in many important cases QUITs can be anticipated; for example, since the code which checks to see what memory can be seen by a processor encounters a QUIT each time it accesses a non-existent page, it must be prepared to handle such QUITs. The mechanism used is as follows: By convention, any instruction which is likely to encounter a QUIT has a certain special instruction immediately following it, one unlikely to occur accidentally. The QUIT handler checks on entry to see if that special instruction appears immediately after the instruction that caused the QUIT. If so, it assumes that the QUIT was anticipated. The special instruction used is a NOP (no operation) with an address to which the QUIT handler transfers in the event that a QUIT takes place. (If there is no QUIT, the NOP is merely executed by the processor with no effect.) This mechanism effectively gives the programmer the ability to do a "load register and transfer in case of QUIT" command. It should be noted that this is an expensive command to execute if the QUIT does in fact take place, because the entire interrupt mechanism is invoked. The macro QUTPAT is used in the coding to generate this quit pattern.

3.4.2 ILLOP

The second interrupt type is for an illegal operation code, referred to as ILLOP. In general, such an interrupt is an indication of a trap. In many places in the application program (and also in STAGE) the programmer wishes a simple way to report that something seriously wrong has occurred. The Pluribus hardware traps any instruction of the form(9) EXXX! or FXXX! as an ILLOP. The ILLOP interrupt handler checks for such a word, treating it as an indication of the XXX! trap and reporting its occurrence to the NCC. For traps of the form EXXX!, there is merely a report that the trap has occurred, while reports of FXXX! traps include also the contents of certain relevant registers. (This latter is used by maintenance personnel to gather information about hard-to-track-down bugs in the hardware or software.) The interrupt handler records the trap data in a special place in local memory, and Stage AR copies those data to common memory. It is then sent to NCC by Fake Host 3. As a special case, if the debugging mode is enabled, the instruction FADE! is treated by the ILLOP handler as a desire to stop the Pluribus. A switch is set so that each processor stops as soon as it returns to LOOP.

---

(9) The following numbers are hexadecimal, with "X" standing for any hex digit; as previously noted, "!" is used to denote that a number is hexadecimal.

3.4.3 JIFFY

The Pluribus is interrupted 60 times per second by the 60 Hertz clock interrupt (the JIFFY interrupt). This interrupt is used to check that certain events are continuing to occur. In particular, the processor checks on JIFFY interrupt to determine if it is stuck awaiting a lock or stuck in some other loop. A loop is indicated by the failure of the processor to enter STAGE for longer than about 150 milliseconds. If the loop is caused by a stuck interlock, which the JIFFY interrupt handler determines by examining the pattern of instructions being executed, the lock is arbitrarily cleared. Otherwise, STAGE is restarted. Either action is reported with a trap. The same interrupt level is also used for a local power failure and for power restoring. The Pluribus power supplies are built so that the processor is given a warning a few milliseconds before the power actually leaves the buses, time enough for an orderly shutdown. When power is restored, a distinct interrupt is given so that the processor can again start up cleanly.

3.4.4 Remote Power Failure Interrupt

The fourth interrupt occurs in case of a remote power failure such as a failure on a memory or I/O bus, and also if the attention button on the operator's console is depressed. The effect of the former is an orderly shutdown of the processor (as for a local power failure) and that of the latter is to report (trap) an unexpected interrupt.

3.4.5 Paper Tape Reader (PTR)

When the paper tape loading code in DDT has been triggered, DDT enables interrupts from the paper tape reader. The interrupt handler enters incoming characters into a ring buffer in common memory. If the buffer fills, the interrupt for the paper tape reader is disabled until DDT can run and remove some of the characters from the buffer. At all other times, paper tape reader interrupts are inhibited and the handler is never entered.

3.5 Individual Stages

STAGE consists of eleven modules. Each module in the sequence examines an increasingly more complicated aspect of the system configuration, so that the STAGE processing begins by verifying the basic local state and then moves outward to check additional resources until the entire system has been examined. Each STAGE module depends on the successful completion of previous checklist procedures for correct operation, and the system must insure that no STAGE module is run until the earlier stages have been successfully completed.

The individual stages are now described in turn. They are executed at system startup time in the order presented and are identified by a two-letter mnemonic as follows:

LK	local kernel checksum
MD	memory discovery
RK	reliability kernel discovery
BD	common bus discovery
CD	coupler and processor discovery
RC	reliability page checksum
LC	local memory checksum
MC	common memory checksum
MM	common memory management
ID	I/O device discovery
AR	application reliability dispatch

3.5.1 Stage LK -- Local Kernel Checksum

Stage LK is run by a single processor and assumes only local memory. That is, it makes no assumption that there are other processors, any common memory, or any I/O. Its purpose is to initialize the processor and perform certain other tasks, and to be sure that the STAGE code in local memory is correct. Its actions are as follows:

1. Set the local interrupt vectors and enable interrupts. The code insures that the interrupt vectors have proper values and then executes the appropriate instructions to enable the interrupts. This action enables the power failure, restore, and JIFFY interrupts. (The QUIT and ILLOP interrupts cannot be disabled.) Since the JIFFY interrupt is enabled, locks encountered in succeeding stages which are held locked are properly cleared. (See section 3.4.)

2. Discover the console. Although the system does not require a console, it takes advantage of one if it exists. In particular, certain values (such as processor, host, and modem status) are displayed in the address and data lights.

3. Checksum the local kernel. This includes the code for Stage LK and the next two stages, as well as all code used in interrupt handlers and several shared subroutines. Once this area has been checksummed and found correct, it is safe for this stage to continue, for interrupts to take place, and for the next two stages to be run. None of these assumptions can be made until this checksum has taken place. If the checksum is bad, the processor halts and waits for another processor to fix things up and restart it.

4. Find a real-time clock (RTC). Once this is discovered, the variable SYTIME can be kept up to date and consensus for the later stages may be maintained.

On successful conclusion of this stage, it is known that interrupts work correctly and are enabled, and that the stage code in local memory has the correct checksum.

3.5.2 Stage MD -- Memory Discovery

Now that it has been established that the local processor is operational, that the STAGE code in that processor's local memory is correct, and that the interrupt handling is assumed to be usable, the next task is to discover what common memory exists. This stage takes place in two major steps. First, the code examines common memory and attempts to determine what is there. Each page is looked at and certain consistency checks are performed on that page. Also, the communication page is located. The second major step is to compare the results of the first step with those of the other processors, using the usual consensus mechanism. At the completion of this stage, it has been established that this processor sees at least the same memory that is seen by the rest of the system. If the processor sees extra memory, it tries to update the common memory table.

3.5.2.1 Stage MD Part 1: Memory Test

The memory test consists of looking at all possible pages in the address space. Although the Pluribus address space includes pages 0 through 127, an assembly parameter is used to establish a smaller, more realistic upper limit to save time. This limit is 63 in the IMP. The memory test is performed in steps as follows:

1. Store into word SMDBUC of the page. If a QUIT occurs, the page does not exist and is skipped; if not, continue looking at other parts of the page.

2. Lock the test area. Since it is important that not more than one processor at a time be performing the memory test, lock word WMLOCK is used. Note that if this lock word (or any other lock used by STAGE) happens to be initially zero (i.e., locked) at system startup time, the JIFFY interrupt handler detects that the processor is hung on this lock and ultimately clears it.

3. Store patterns. This tests the memory to see if various patterns of 1′s and 0′s can be stored and retrieved correctly. The block of memory at SMDBLK is used.

4. Check the locking operation. A load-and-clear reference (through map 3) is performed, and a check is made that the result is zero; if not, the memory is unusable.

5. Check that the self-pointer word SLFLK contains the proper value. If it is incorrect, it is repaired. At this point the lock set in step 2 is cleared.

6. Establish the communication page. The method used is discussed below.

The communication page, the lowest numbered page that can be seen by all processors, is used by STAGE for communicating between the processors. An important item of communication is the consensus, in which all processors must agree on some action. The communication page provides the place to store consensus data. Stage MD has the task of getting all processors to agree on which page is to be the communication page. This requires unanimous agreement of all processors, but there is not yet an agreed upon place where the processors can record their views on the matter. Thus the following method is used.

The main loop in Stage MD is to look from page 0 through some upper limit, examining each page. The first page it encounters which passes memory test steps 1 through 5 above is that processor's candidate for the communication page.

Every page in the Pluribus has a word called COMPTR which contains in its left 7 bits the address of the communication page. There is additionally a word COMTST on each page which is used to establish, by a type of consensus, just which page is to be used for communication. This is achieved by assigning each processor its own bit in COMTST in the same format as the FIXIT word. A processor running MD which concludes that the value of COMPTR on a given page is correct sets the entire COMTST word on that page to zero. A processor concluding that COMPTR is incorrect sets its bit in COMTST to 1 and then waits one minute. If at the end of that time the bit is still 1 (i.e., no other processor running Stage MD has set COMTST to zero), it sets its bit to zero and changes COMPTR to what it determines to be the communication page. If, during the one minute waiting period, another processor running Stage MD agrees with the existing value in COMPTR, it will set COMTST to zero and the waiting processor will never get the chance to make the change. In this case, the waiting processor proceeds no further in STAGE and is unavailable to run the application since it cannot agree with other processors on where to communicate.

Once all of these tests have succeeded for an individual page, the proper bit for that page is set in the MYSEGS table. This table is contained in local memory and reflects all common memory pages visible to the local processor. Further, the processor sets the COMPTR word of all pages to the page which it determines to be the communications page. If other processors are running, then the COMPTR words reflect the value agreed on by all processors.

At the completion of part 1 of Stage MD, MYSEGS contains the bit map of all pages which the current processor can see, and the normal consensus mechanism can now be used.

3.5.2.2 Stage MD Part 2: Page Map Consensus

STAGE maintains on the communication page a data base called MEMSEG which is a bit map of the pages that are available on the system. This last part of Stage MD is a comparison of MYSEGS with MEMSEG. If they disagree, the consensus and FIXIT mechanism is used to set MEMSEG to the value of MYSEGS. As noted above, it is now possible to use a consensus because processors agree on the communication page.

At the end of Stage MD, all running processors agree on which common memory pages exist and may be used, as well as on which page is to be used for communication.

3.5.3 Stage RK -- Reliability Kernel Discovery

This stage and the two previous stages run in local memory, since up until this point it is not yet known where in common memory anything can be found. The purpose of Stage RK is to find that page in common memory (known as the reliability page) which contains the code for the next three stages, a code referred to as the <u>reliability kernel</u>. Once it has been found, succeeding stages can be run from it in common memory. As local memory is a precious commodity and in short supply, it is mostly reserved for frequently executed application code. Since STAGE is not executed very often during the running of the application, it is desirable to store as little of it as possible in local memory. Of course, some of it must be in local memory since certain parts of STAGE run before common memory is discovered.

There is one other important aspect of Stage RK. Once the reliability kernel has been located, it is known that the necessary code to perform a reload exists and is correct. If at any point thereafter it is determined that parts of the code are incorrect (i.e., have an incorrect checksum) or that needed code (or other data) are not available, it is possible to get a good copy of that code or data. The code is obtained through a packet reload whereby pieces of code are obtained from elsewhere in the network. The assumption is that after Stage RK has run successfully, it is possible to do a reload, since the code needed to perform the reload is contained in the reliability kernel which is now known to be correct.

Stage RK looks through all of the pages of common memory to locate the reliability kernel. This page is recognized in two ways. While the system is running, the word COMREL on the communication page points to the reliability page. In addition, the reliability page has the password ACE!.(10) While the system is running, Stage RK need merely look at the page pointed to by COMREL to be sure that it has the proper password. During system initialization time, however, Stage RK looks at all pages in memory to find one containing the proper password. When it is found, two checksum operations are performed on the reliability page. First, STAGE RK determines the checksum for just that part of the page containing the reliability kernel, and then it performs the usual checksum operation on the whole page. The reliability kernel code is usable if just its own checksum is correct, although of course it is preferable if the entire page checksum is correct. If the reliability kernel checksum is not correct, the system can proceed no further. If it is correct and the page checksum is not correct, COMREL is set with a 1 in the least significant bit to indicate to subsequent code (Stage RC) that the page must be reloaded.

On completion of Stage RK, all processors have agreed on the location of the reliability page. The consensus mechanism is used to do a FIXIT on COMREL to be sure that it is correct and that all agree. From this point on, STAGE can run from common memory.

3.5.4 Stage BD -- Common Bus Discovery

This stage discovers the memory and I/O busses that exist in the hardware. It proceeds as follows:

1. Initialize storage. The STAGE variables which are maintained in the high addresses of the reliability page are checked for consistency (via a software watchdog timer) and initialized if necessary.

2. Scan the variables area for QUITs, reinitializing if necessary.

3. Discover the buses. First all memory buses are deduced from the data stored in MEMSEG. Then all I/O buses are discovered. A check is made for the existence of PIDs and RTCs.

---

(10) The <u>password</u> is a particular word in the checksummed part of a code page. The reliability page has ACE! in that word, a pattern which does not correspond to any reasonable Pluribus instruction.

4. If no PIDs exist, the processor reenters STAGE from the beginning (Stage LK).

5. Using the consensus mechanism, the results of Stage BD are stored in the single word USEBUS in which the leftmost bits represent RTCs and the rightmost bits represent buses.

3.5.5 Stage CD -- Coupler and Processor Discovery

This stage discovers processor and I/O bus couplers.

A coupler appearing on all I/O buses is assumed to be a processor. Its proper backwards bus coupler (BBC) password is determined from configuration tables in the reliability kernel. The code tries to set this BBC state into one of the I/O couplers. If this results in a QUIT, the processor's own coupler has been found. Note that M/I to M/I bus couplers must not appear in I/O space (but instead in memory space) on the target bus to avoid possible interference with processor discovery. Consensus agreement is obtained for the COUBUS and IOCTAB tables which are filled out as couplers are found.

Since M/I to M/I bus couplers appear in memory (as opposed to I/O) address space, they will not interfere with processor bus discovery. Once this stage has completed and reached consensus, the Block Transfer and Packet Reload routines will be polled. Under consensus agreement, this stage fills the following tables:

COUTAB is the list of processor couplers. This is the list of processor names also, since coupler addresses are assigned in the Pluribus according to the physical location of the bus.

COUBUS tabulates, for each processor bus, which of the common buses it appears on. This information is used to maintain proper amputation status. IOCTAB tabulates what I/O to memory or M/I to M/I buses have been found, and to which memory buses they connect.

Finally, as processors are discovered, the results are tabulated in PROCEX. This is a bit table, where two bits are assigned for each bus in COUTAB, corresponding to each of two possible processors per processor bus.

3.5.6 Stage RC -- Reliability Page Checksum

This stage insures that the checksum on the reliability page is correct by checking the checksum calculated previously by Stage RK and stored in COMREL. If the checksum is found to be incorrect, all following stages are disabled and an attempt is made to reload the reliability page.

3.5.7 Stage LC -- Local Memory Checksum

This stage checks that the checksum for all of the code in local memory is correct. Recall that Stage LK performs a checksum in local memory, but that check is only for the part of local memory used for STAGE. The STAGE system can run successfully even if part of local memory is broken, providing that the broken part is not needed for running STAGE. Stage LC now checks all of local memory.

Stage LC uses the usual checksumming mechanism to compute the checksum of local memory up to the limit specified in HOTLIM. It then uses the consensus mechanism but with an interpretation differing from that normally used. A processor sets its FIXIT bit for this stage to indicate that its own local checksum is incorrect. If all processors have an incorrect local checksum, the last one to discover this fact initiates the fix as usual for a consensus. The fix in this case is to use Packet Reload to reload local memory from elsewhere in the network. In the more likely situation where at least one processor has correct local memory, the bad processors hang in Stage LC. Later, in Stage AR, the local memory of these processors is refreshed by a correctly running processor.

3.5.8 Stage MC -- Common Memory Checksum

This stage checks each page in common memory. First, a specified part of each page is checksummed. If a processor finds the checksum to be incorrect, it indicates this by setting its processor bit in the FIXIT word of that page. If there is a consensus that that page has an incorrect checksum, the page type is set to -1, the page limits for checksumming are set to include only up to the type word, and the checksum is changed to the correct value for an empty page. The effect is that the page appears to be free.

The checksummed area on a common page is, in many cases, only a small part of the page, and the next step is to look at the rest of the page for QUITs. If a QUIT occurs, the word causing it is set to zero and then read back to see if it really is zero, thus clearing parity errors in memory. If a QUIT occurs for any word in the page, that page is marked for subsequent reinitialization. If the zeroing and rereading process fails, the processor stops using the page involved.

3.5.9 Stage MM -- Common Memory Management

This stage establishes that the necessary pages are in common memory. It insures that each required page for the application is in common memory and is in the proper place. It was noted in section 3.3.1 that there are five kinds of pages: code pages, required variables, desired variables, spares, and optional variables, in order of decreasing importance. Stage MM attempts to place code pages and spare code pages on different memory buses, so that the failure of the memory bus containing the code does not stop the application. Further, it is desirable that variables pages be on a different memory bus from code pages so as to minimize memory contention. For this reason, Stage MM works from both ends of memory. Code pages are loaded at the low end (small addresses) of memory, and the upper portion of memory contains (from high addresses to lower ones and in the order named) required variables, spare code pages, desired variables, and optional variables. If any space remains between the top and bottom of allocated memory, such pages are marked as being free pages. Figure 10 shows the different page types as they reside in Pluribus common memory.

Stage MM looks at the bit table which indicates which pages in common memory are available. It maintains two pointers, one to the lowest numbered free page and the other to the highest numbered free page. As a specified page type is moved to its proper place, these two pointers are adjusted accordingly. While this stage operates, it is building up a data base called LMAP, stored in local memory. LMAP shows the location of every page type in common memory. On completion, the stage checks that LMAP is in agreement with CMAP, the corresponding data base in common memory.

LMAP and CMAP have identical formats. There is an entry for each page type, and the page type serves as the index into LMAP. The entry in LMAP contains the value to be loaded into a map register in order to address the page. During program execution,

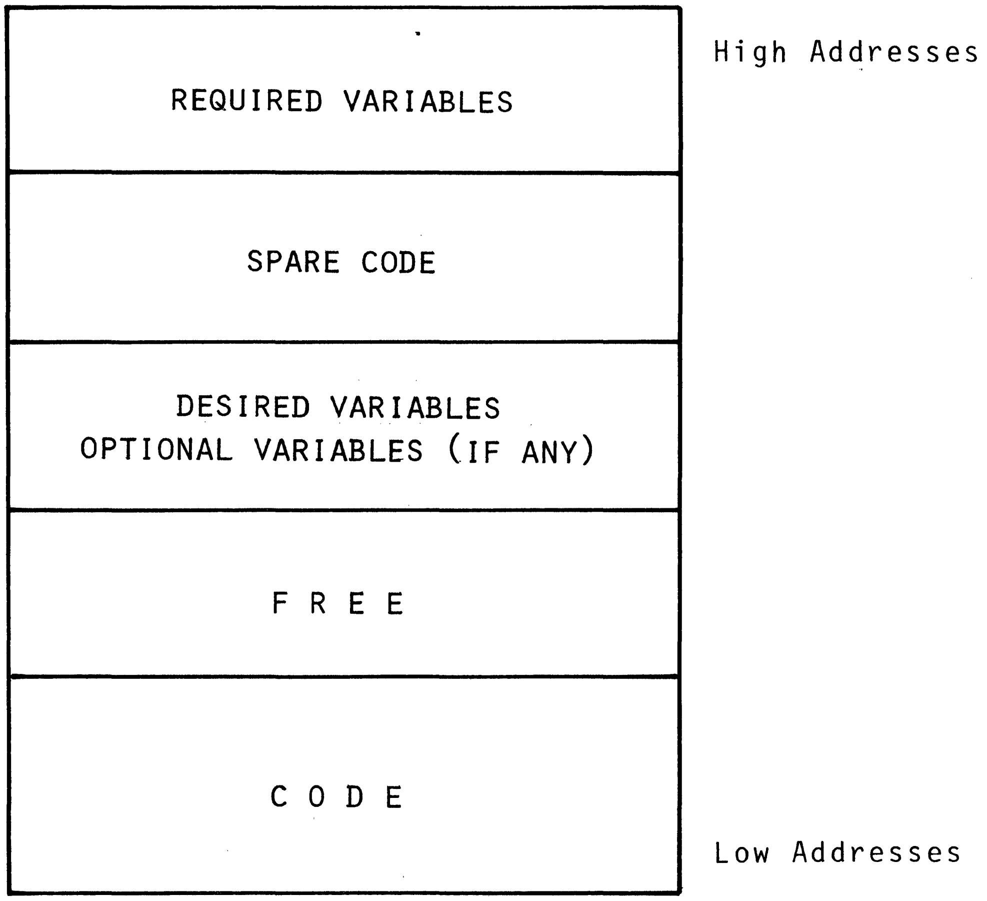


Figure 10. Organization of Common Memory

the application program uses LMAP to load map register. The sole purpose of CMAP is to provide a means by which STAGE can insure that all processors have the same picture of common memory.

A common subroutine is used to perform the processing for all five page types. The requirements for each of the five page types are described in the order in which the processing takes place in Stage MM.

1. Code pages. Code pages are placed into the low end of memory. If the necessary code page can be found, it is copied into the proper place. If not, the corresponding spare code page is searched for and, if found, copied into the proper place. Otherwise, a reload is required.

2. Required variables. Required variables are loaded starting at the top of the memory. If they are not in the proper place but found elsewhere, they are moved; otherwise, initialization is required.

3. Spares. Spare pages are loaded only after it has been determined whether or not there would be enough room left for the desired variables. If not, preference is given to the desired variables and no spare pages are allocated. If enough space exists for both spares and desired variables, the relevant pages for the spares are either moved into place or copied from the code pages.

4. Desired and optional variables. These variables are loaded from the top, following the spare code pages (if spares exist) or following the required variables (if no spare code pages exist). They are either moved from where they exist or are created and initialized. Desired variables actually appear in the optional variables part of LMAP.

5. Any pages which are as yet unallocated are marked as free pages, with page type -1.

As mentioned above, a single subroutine does the work for all five page types. It has two parameters, the page type for which it is looking and the place where it is to be put. The latter is an indication of which of the two pointers into the available page table to use, the one counting down from the top of memory or the one counting up from the bottom of the memory. The subroutine does its work as follows: It first checks that there is room for more pages by determining whether the two pointers have met. If the pointers have met (i.e., no more available memory exists) it asks whether enough pages have been allocated. That is, have all of the code pages and required variables been allocated. If so, the rest of LMAP and CMAP are filled with -1 and Stage MM is done. If the required pages have not been filled, the stage fails and a bit is set in the FIXIT word; there is not enough memory to run the system. If the place where the page is to go does not exist (i.e., that page is not in the available memory), the subroutine merely increments the place and returns. It is called again to try the next page.

The routine then looks to see if the page already at the given place has the proper type; this is the normal situation. However, if that check fails, the stage first looks in unallocated memory (the space between the two pointers) to find one of three things: a page of the correct type, a spare copy of a page of the correct type, or a free page (one of type -1). It then performs the appropriate copy operation (under consensus) to insure that the proper page is in the proper place. The JIFFY interrupt is suppressed to minimize execution time. Part of the copying operation is to change the page type (if necessary) of the final copy, for example if a spare code page is copied to make a code page. Since the page type is part of the checksummed area, both the page type and the checksum must be changed simultaneously. Another part of the copy operation is a simple memory test in which each word stored into the destination is read back and checked for accuracy. Since the code which copies the reliability page also resides on that page, care must be taken to move this code correctly. Finally, the move never takes place while BLT is in operation because Packet Reload may be attempting to reload the page in question.

3.5.10 Stage ID -- I/O Device Discovery

This stage looks through I/O space to determine which I/O devices exist. Although Pluribus hardware permits up to 768 I/O devices, this stage is constrained by assembly parameters to look at only 64 of them to save processor time. It looks at 4 windows into I/O space, each consisting of 16 consecutive I/O blocks. Each window is identified by one word which gives the address of the first device in the window and another word which indicates which of the devices starting at that location actually exist. Stage ID accomplishes the device discovery by reading the first word of the I/O block for each of its 64 possible devices. If there is no QUIT, it assumes that the device exists. It also insures that there is a PID for each window.

Stage ID reports its failure in a FIXIT word, to indicate that the view of I/O space which it has differs from that maintained in the USEIO data base on the communication page. However, this stage differs from all others in using majority logic rather than unanimous consent to decide when to perform the repair. When a processor detects a variance between its view of I/O space and that in USEIO, instead of checking to see if it is the last processor to detect this flaw, it merely adds 1 (for itself) to the number of bits already in the FIXIT word. If that sum exceeds one-half the number of 1's in the smoothed consensus word, it realizes that a majority of the processors has detected a problem and performs the proper fix in USEIO. If not, the processor puts its own bit into the FIXIT word for a consensus.

3.5.11 Stage AR -- Application Reliability Dispatch

This is the final stage before running the application program and is the stage that performs certain final checks to insure that all is ready. In particular, it is able to perform certain tasks required by the application. It proceeds as follows:

1. Recall that in Stage LC, which checksums local memory, if only one or two processors have bad memory they remain hung in that stage. The FIXIT word for Stage LC is now looked at. Any bit in that word indicates a processor whose local memory is incorrect. If any such exists, BLT is invoked to copy a correct version into the local memory of the complaining processor(s).

2. If any processors are halted, BLT is used to attempt to start them.

3. The initialization routine on each page is polled and executed. Word PGINIT in the system part of each page contains either zero or the address of an initialization routine. This initialization routine, a closed subroutine, examines certain data in memory and checks for correctness. In the case of application code pages, this routine checks certain parts of the application, thus giving the application package a chance to have STAGE perform certain checks for it. Failures reported by these initialization routines are indicated by a FIXIT bit, and the usual consensus mechanism is used. The action in the event that all processors agree is to return to the application routine to perform the required initialization.

4. In the event that the processor is an even processor, it determines whether or not the 60 Hertz clock is running. If not, it executes a trap.

5. The effect of any trap in the Pluribus IMP is to store certain information in local memory of the processor. This part of Stage AR causes that trap information, if non-zero, to be copied into a buffer in common memory for eventual transmission to the NCC. Finally, SOKAY is called to enable the application program.

6. Certain local hardware failures are noted by traps:

   a. Nonworking JIFFY.

   b. Successful QUIT retries.

   c. RTC read errors.

3.6 BLT -- Block Transfer

A special part of the STAGE system is a module called BLT, a routine used to move blocks of data from one place to another. BLT deals with three problems:

- Some of the moves take up to several seconds of processor time, much too long for a single strip. BLT must break periodically as dictated by the timing requirements of the application.
- Moving data into the local memory of another processor requires backwards bus coupling.
- When a part of memory is missing, BLT is able to reload it from an external medium (a neighbor IMP or the NCC).

Because of these considerations, BLT is the single routine used to move blocks from one place to another.

When any STAGE code wishes to move something from one place in memory to another, it first determines that BLT is not in use. If BLT is free, it sets up the parameters for the move and then goes about its business. WSLEEP dispatches to BLT periodically just as it dispatches to any stage. Whenever BLT is entered, it looks in its parameter block to see what it should be doing. If it finds nothing, it returns to WSLEEP. If it finds a task, it moves blocks of data of an amount appropriate to the maximum strip time dictated by the application. Having done so, it adjusts the parameter block appropriately and then returns to WSLEEP. This process is repeated until the task is completed.

The part of STAGE which initiates the call for BLT may well be able to go on about its business. If at a certain point it cannot proceed until the move is complete, it returns to WSLEEP. Every time the stage is re-entered by the normal dispatch, it checks to see if BLT has completed.

BLT operates by being given a source and a destination. It first copies the data to be moved from the source to a buffer area in BLT, and then copies it from the buffer area to the destination. BLT has a state word with three possible values: free, source, and destination. Thus BLT always knows what it is doing by looking at its state word, and users of BLT can determine from that state word whether or not BLT is free.

BLT is controlled by four values. These are the type of the source, the type of the destination, the address, and the length. The type of the source or destination specifies whether it is local memory, common memory, or external. The address is always the same for both source and destination. The length is the number of bytes to be transferred. In addition, if the source or destination is local memory, there is an associated mask word. The mask specifies which are the relevant processors. If the source is local memory, the mask specifies which processors may be used as the source of the data. Then when BLT is running in one of those processors, that processor does the work. If the buddy of one of the processors is running BLT, it is the buddy that does the work. Otherwise, another processor uses backwards bus coupling to fetch the required data. If the destination is to be local memory, the mask specifies which processors are to be loaded. Each time BLT loads one of these processors, it removes its bit from the destination mask.

An attempt to copy from external (i.e., a reload) is sometimes interrupted to take a dump. That is, if the system decides that a part of local memory must be reloaded, something external to the reload mechanism (such as the NCC) may elect to take a dump first of some or all of the Pluribus memory before reloading. Thus the transfer from external to Pluribus memory is trapped and a copy operation from Pluribus memory to external is performed first, followed by the originally requested reload.

Recall that Stage AR is used to reload local core in a processor and to start it if necessary. This latter task (starting a processor) is signaled to BLT by a special bit in its state word. If the bit is set, BLT starts the processor at the beginning of the STAGE system. BLT is used because it incorporates the necessary BBC code.

BLT is aware of checksummed areas. When it copies into such an area and a special state bit is set, it takes the necessary steps to change the checksum. It does this in such a way that any stage simultaneously checksumming that area does not get into difficulties.

Additionally, BLT is used by packet core in the IMP for patching a word from the NCC.

3.7 Packet Reload

If a page of memory in the Pluribus IMP is found (in Stage MM) to be defective, an attempt is made to make a repair. If possible, the repair is made using only the resources available in the IMP itself. For example, if a checksum error is found on a code page and there is a spare copy of that page, the latter is copied. Similarly, an error on a variables page is repaired by reinitializing the page and restarting the IMP. In some cases, however, the IMP is unable to effect the repair itself and requires outside help. In the usual case, the outside help is a copy of the missing code or data sent from elsewhere on the network. The Packet Reload module is the sender or recipient of such transmissions.

The Packet Reload mechanism involves three processes, as follows:

- The IMP with a problem uses Packet Reload to obtain the necessary information from the network. Providing Stage BD has run correctly, Packet Reload may be run. Essentially, the IMP sends out a special protocol message stating what part of memory it wants and the type of the IMP (316 or Pluribus) that it is. It then awaits receipt of the requested data.

- An IMP elsewhere supplies the data. When an IMP receives a reload request from another IMP of the same type, the Packet Reload module generates the proper data for transmission.

- There are possible intermediate IMPs between the one with the problem and the one with the data. These might be IMPs of the other type not capable of supplying the requested data. For example, a 316 IMP cannot supply reload data for a Pluribus IMP, but it can pass the request on to a neighbor until it reaches an IMP which can help. Similarly, a Pluribus IMP passes through itself reload requests for a 316 IMP.

A special protocol is used for moving a portion of core memory from one IMP to another over the network. A sending or receiving process is implemented as Fake Host 2 which is the recipient and generator of all Packet Reload messages.

There are only two message types in the Packet Reload protocol. One type is SETUP, which sets up internal variables that determine whether the process is sending or receiving, the location and size of the core transfer, and the network address of the foreign opposite process. The other message type is CORE, which contains one or more segments of a core image. A Packet Reload process which is idle is unlocked and neither sending nor receiving. An idle process accepts any CORE or SETUP message with the proper machine type.

If a process accepts a "SETUP send" message, it locks to the foreign process specified in the SETUP data and begins to send core segments at a rate specified by the send/receive flag in the SETUP. After the last segment is sent, it resets to the idle state. A process that is sending only accepts messages from the foreign process to which it is locked, and these may be either SETUP or CORE messages.

If a process accepts a "SETUP receive" message, it also locks to the foreign process, and waits for CORE messages. When a CORE message that completes the specified core transfer is received, the process is reset to the idle state. A process that is receiving only accepts messages from the foreign process to which it is locked, and these may either be any SETUP, or a CORE message with a start address equal to the address next expected by the process. If the address is too low (i.e., some previously received segment), the message is ignored. If it is too high (i.e. some segment was missed), a "SETUP send" message is sent to the foreign process, specifying the current address required. A "SETUP send" is also sent when no messages have been received for some time and the process is reset to the idle state after a much longer period when no messages are received.