Stu Card will present Dealer on Tuesday, June 3 at 1:30. He'll tell us about some predictions he made concerning the results of an editing experiment done by RAND.

2-JUN-75 16:24:43-PDT,187;000000000000
Date: 2 JUN 1975 1624-PDT
From: ADAMS
Subject: OCG Meeting
To: OCG:

There will be a meeting on Tuesday morning, June 3, 10:30.
See you in the caterpillar room.

Syl

3-JUN-75 10:01:23-PDT,204;000000000000
Date: 3 JUN 1975 1001-PDT
From: ADAMS
Subject: ocg meeting
To: OCG:

Due to circumstances beyond our control, the ocg meeting will be postponed until 11:45 a.m. today.
Thank you!
Syl

3-JUN-75 15:02:05-PDT,482;000000000000
Date: 3 JUN 1975 1502-PDT
From: ADAMS
Subject: Seminar - Stanford Design Students
To: OCG:, MCS:, CSL:, MCS:, LRG:

In connection with a class project this year, three design students at Stanford University have developed a split keyboard and joystick for cursor control. This is now running a test program on an Alto. The students will present the project and demonstrate their system at 4:00 p.m. on Wednesday, June 4 in the Commons, second floor.

Syl

4-JUN-75 03:39:53-PDT,4678;000000000000
Date: 4 JUN 1975 0339-PDT
From: TAFT
Subject: protocol changes and additions
To: METCALFE, BOGGS

I've been doing some more thinking about higher-level Pup-based protocols, and it occurred to me that there are an awful lot of things which aren't written down and which at least I have been carrying around in my head. It's as easy to send you a message as it is to write these things on a piece of paper, so here goes:

1) Changes to Pup at the raw packet level

a) The checksum is a ones-complement add-and-cycle, with the value 177777 being taken as meaning "unchecksummed Pup" (rather than 0)

b) Encapsulation on MCA consists of setting the sign bit of the Pup Length field and sending the whole Pup (with no other header words) as a single packet (actually, some integral number of physical packets, using the time-tested techniques of the old MCA protocol).

c) Encapsulation of Arpanet packets consists of prefixing the packet with a standard Imp-Host protocol leader (32 bits), with link number 152 (decimal) and message type 0 (= Regular message)

d) Encapsulation on Ethernet consists of prefixing the physical destination/source and type words, with the Ethernet Type of a Pup being 1000 (octal)

e) While the definition of Pup permits packets with lengths in the range 22 to 2↑16-1 bytes, in practice some limit must be adopted. The limit that has been arrived at is 512 data bytes (i.e. Pup Length no greater than 534). In the ordinary course of events, no host should be required to handle Pups longer than this, and no host should generate Pups longer than this if it expects most other hosts to be able to handle them. Of course, this does not prohibit hosts from communication using larger Pups by prior agreement amongst themselves. Nor is it necessarily the case that all hosts or all networks will be able to handle 512-byte Pups [for example, if we ever start using the Arpanet "uncontrolled packet" mechanism, we will be limited to Pups of no greater than 1007 bits or 125 bytes].

2) Changes to BSP

a) The "End" sequence is broken out from BSP and made into a separate protocol (the "End Protocol"). The only change is that "End" and "end Reply" packets use the Connection ID rather than the ID of the most recently transmitted byte.

b) I think we need some sort of protocol (probably part of BSP) for resynchronizing the two ends of the connection, in case of partial loss of memory at one end. The case I have in mind is that of the DLS Alto. I would envision being able to preserve the Connection ID and the local and foreign port numbers across most Alto crashes (by writing them on the disk or into a carefully-checksummed region of memory), but losing track of the current Byte ID's, Interrupt ID's, etc., since they change too rapidly to be maintained in a safe place. My first thought is to have a "Resync" packet containing the following information: the Connection ID in the Packet ID field, and the sender's opinion of the current Byte ID and Interrupt ID for the stream he is sending. The receiver of a Resync packet would respond with a Resync packet containing the complementary information. I haven't thought out the details, but I think such a mechanism should work ok. Comments?

3) Thoughts for other protocols

a) I think we need some sort of "Error Protocol" that allows the sender of a packet to be notified of the cause of the demise of that packet along the route from source to destination. This will be particularly important if the source process is to correctly deduce and correct problems such as Pups being too big to send through some intermediate network. In the simplest case, all we need is an "error" packet, constructed (by the process generating the error) as follows: Pup type set to "error", source and destination ports reversed, Pup ID the same as the original packet, and Pup Contents including some indication of who is generating the Error and a code (with possible arguments) signifying the cause of the error.

b) We need to work out a protocol by means of which hosts can find out what other hosts are gateways and what other networks are accessible.

c) Applications such as shipping encapsulated Arpanet Host-Host protocol packets and raw magtape records via Pup (even though such packets are bigger than the maximum-length Pup) will require some sort of "Multi-packet message" protocol, perhaps analogous to the way in which Arpanet messages get split apart and reassembled by the Imps. Such a protocol must be more efficient than BSP.

Core dump completed.
Ed

4-JUN-75 11:21:32-PDT,684;000000000000
Date: 4 JUN 1975 1121-PDT
From: HUNT
Subject: OS
To: Altousers:

There are new versions of OS.BOOT, OS.BM, and INITALTOIO.BR on <ALTO>.

The new OS has bugs associated with positioning files fixed; clarity.al is the default font; stream operations are slightly faster; the new autil is incorporated. There is a new OS function, SetFilePos, which is the inverse of FilePos; SetFilePos(s,v) where s is a stream and v is a vector. This function replaces READNEXTPAGE in the initialization file INITALTOIO. If your program calls READNEXTPAGE explicitly, it will not work with the new OS!!! Problems should be taken to Bulter or Willie Sue.

5-JUN-75 09:42:51-PDT,486;000000000000
Date: 5 JUN 1975 0942-PDT
From: DEUTSCH
Subject: MITSmobile at Rickey's
To: CSL:, OCG:, LRG:, MCS:

XEROX PARC CSL INTERNAL MEMORANDUM

TO: Boggs,Baudelaire,Deutsch,Duvall,Fiala,Lampson,Liddle,McCreight,Melvin Rider,Shoch,Simonyi,Sproull,Sturgis,Swinehart,Taft,Thacker,Zelinsky and others who may wish to find themselves involved

FROM: Bob Metcalfe

SUBJECT: Pup Revisited -- Toward a Parc Universal Packet

DATE: April 14, 1974

This memo is written and should be read with caution; its purpose is to promote an internetworking packet standard for Parc. Multi-font splendiferosity aside, *Pup Revisited* is still only a document's latest version, crudely constructed inside.

An earlier version entitled *A Proposed Pup -- Parc Universal Packet* was distributed on March 19. And before it, a directly related memo entitled *Experimental Ethernet File Transfer Protocol* was distributed on February 5. This here memo comes with (debated and debatable) revisions and further developments, making the two previous memos obsolete. Deutsch's *Tenex MCA/Ethernet Interface* memo of April 8th is also relevant.

The list of the packet networks at Parc includes Ethernets, Locainets, Arpanets, MCAnets, and EIAnets. All have been considered, more or less, in the personal turmoil leading to our current pup proposal.

Successful implementation of our internetworking packet standard will benefit from an early multi-lateral agreement among the czar(esse)s of Parc's various existing and planned packet networks (*sic*). With an early agreed-upon internetworking standard in hand, the many network-based systems yet to be built can be done right the first time -- built for the internetworking environment. The instrument of this proposed agreement is to be an object which we affectionately call a *pup*, a Parc Universal Packet.

A problem *barks* for our attention: How should we interconnect these packet networks and the computers to which they are attached?

We propose that a standard packet protocol be adopted to allow processes living on any of our interconnected computers to send packets among themselves through any of our interconnected networks. Adoption of such a standard would give us a general interprocess communication system. Not all host-host or process-process communications need use the general pup system, of course; keeping this in mind will help us prevent our pup from becoming a real *dog* ("Disgustingly Over General"). Those that use pups won't care which of our interconnected networks and computers are involved. And this can be a good deal.

Imagine the economies of being able to access the following resources from any process in the known interconnected world: Rider's "Ears" Slot printing server, Boggs's magtape-controlling Altos, terminals coming in from the DLS Altos, the Arpanet through Deutsch's Polos NCP, Polos editors and formatters, the Parc file system (Maxc now, distributed later), and a *host* of others; the *net* result is *imp*ortant, no matter how you *packet*.

1.0 InterNetworking Principles

Here are some higher-level concepts.

1.1 Heterogeneity. We recognize that Parc has its various networks, not principally because of any lingering attachment to obsolete capital equipment, but presumably because each network seems more suited to certain applications than the others. Therefore, in interconnecting these networks and their computers, we must not simply wash out the differences with emasculating standards, but breed the benefits of underlying variety.

This is to say that we do not intend the proposed pup standard to be all things to all people; many will and *should* avoid internetwork standards to get at the particular capabilities offered by their favorite network.

1.2 Encapsulation. Because we are to retain the advantages of our differing networks, pups must be in general a subset of the packets carried by each network. Thus, a pup is to be encapsulated as it passes through any one network so as to be a recognized type of packet in that network. As only one of many types, the general pup standard will invoke internetwork processing overhead only on those packets requiring it.

1.3 Gateways. The active components in the system to be built according to the standard protocol are (to use internetworking jargon) its "gateways", processes which take packets from one network, diddle them, and hand them to another. From our point of view, the processes which we often call Network Control Programs (NCPs) are already gateways; they transmit packets from the software networks of processes inside computers to the hardware networks outside. With internetwork connections we introduce additional gateways; processes which actually sit at the junction of two (or more) hardware networks.

> *Note: Gateways can be viewed as store-and-forward packet-switching nodes at the internetwork level; they are meta-Imps.*
>
> *Note: In a host connected to several networks, it is likely there would be several NCPs, one for each network, and a gateway putting them all together for internetwork switching.*

1.4 Best Efforts. We believe that when a packet is given to a raw process-process packet system, the system should promise to give only its best efforts to deliverence; the sender of the packet should expect its successful delivery only with some stated high probability (say 9/10, or 99/100, or 999/1000) conditional on the destination being ready and waiting. Higher quality communication, say with generalized process rendezvous, additional error control, and some flow control, should be provided with algorithms implemented at the application-process level.

1.5 Thin-Wire Defensiveness. If we build our systems so that their constituent processes communicate through thin wires with packets (pups hopefully), then we will have achieved a high degree of isolation -- modularity by any other name smells as sweet. Because thin-wire interprocess communication is explicit and requires the active participation of the sending and receiving parties, processes can be arbitrarily scrutinizing of their communications; they can be defensive.

Along these lines, Ed Taft suggests that one good test of a process is to send it a random pattern of randomly generated packets to see if it can keep its feet. A process which goes off the deep end on a garbage packet needs fixing.

Perhaps we are approaching something of a methodology for building reliable systems -- *best-efforts thin-wire* communication among constituent processes. See thesis.

2.0 The Parc Universal Packet

A pup is a collection of 8-bit bytes: 18 bytes of *header* and a variable number of bytes of *contents*. The first 2 bytes of a pup header carry its (encoded) *length*. Then 1 byte of *transport-control* bits. Then 1 byte of pup *type*. Then 2 bytes of *sequence number*. Then come two *port addresses*, each 6 bytes. That's the header. And then come the pup's (few) content bytes, about which the pup definition has nothing to say. When a process hands a pup to its local gateway, it expects that the packet will be delivered to the waiting destination only with some high, yet to be specified (say 99/100), probability. That's it.

> *Note: Yes, the pup header has grown larger (not smaller) since last time; it may already be a dog; but let's be patient.*

A pup is a virtual packet; virtual to the source and destination processes involved. On the wires or in the memory of some specific network or computer it may look different, encapsulated, with fields rearranged, with extra fields containing network-specific data. You'll see what we mean in the example of how a pup might be transported through an Ethernet, below.

2.1 Length. We propose that the length of a pup be carried as the number of 8-bit bytes in the pup, including the length itself, the transport-control byte, the type byte, the sequence number, the addresses, and the contents. We further propose, after Duvall's suggestions below, that this length be encoded.

The purpose of this length is to facilitate transport of the pup without disrupting its contents through the inadvertent adding or subtracting of bits. In fact, as a pup winds its way through various networks, it is likely that it will acquire a *tail*, extraneous trailing bits ("padding"), but these are OK; they can be routinely ignored.

Bill Duvall's comments on the last version of this proposal call for a redundant, constant, "this is a pup" field in the pup header and error-checks in the length. The intention is to make it easy to detect a non-pup and to help keep a gateway from going off the deep end on garbage. Our suggestion is to combine these desirable features in an encoded length, say the length with parity bits and constant pattern bits. Someone should suggest a simple algorithm, well-suited to "nova" code, which will encode and decode the length with checking.

2.2 Transport Control. It is anticipated that the gateway system will need to be able to carry certain information in pups for debugging and control. Two examples come immediately to mind. The first is that of flushing pups which have been around too long, say by using a gateway hop count in the transport-control field of the pup header; we mentioned this in our last go-round. The second (suggested by Simonyi) is that of tracing; with a trace bit in certain pups, their passage through the gateway network can be followed and reported during debugging and times of malfunction.

We suggest that other transport-control functions will come up and that 8 bits is

enough. Say let's have 3 bits of hop count and a trace bit, leaving 4 bits for other functions. *Bow-wow?*

2.3 Pup Type. The type byte indicates how a pup's contents should be interpreted. The Byte Stream Protocol sketched below defines several of these types.

The pup type byte will be useful in identifying a packet, as discussed below for the sequence number bytes. All packets will carry types in one form or another and so moving a type indicator into the pup header will not increase our overhead (as a practical matter), but will allow gateways to process pups more intelligently.

A registry of types should be kept as part of the pup standard. And there should be a pup type which indicates a non-standard pup. The type byte is generated by the source process.

2.4 Sequence Number.

Simonyi suggests that gateways will have several reasons for wanting to identify packets; tracing is one. It also appears (we observe) that pups will carry sequence numbers for error and flow control, whether or not their presence is indicated in the pup standard itself. Therefore, the sequence number is made a part of the pup standard so it can be used by gateways (say, in tracing); no overhead is added because pups would have sequence numbers anyway.

It is the responsibility of the source process to assign a pup its 16-bit sequence number and, therefore, a process's pups are uniquely identified by gateways only to the extent that the process keeps the sequence numbers unique.

In the Byte Stream Protocol below, pup sequence numbers are used for retransmission control, duplicate suppression, reassembly, and flow control.

2.5 Addresses. A pup has two addresses. The first is that of the destination port; the second is that of the source port. A *port* address is 6 bytes (48 bits) long and identifies an *area/network*, a *host* computer, and a *socket* in that host. We use the first 8 bits to identify the area/network of a port. We use the second 8 bits to identify the port's host computer in the identified area/network. We use the remaining 32 bits to identify the port's socket within the identified host. This addressing scheme is bred for generality and compatibility from those of the Localnet, the MCAnet, and the Arpanet.

> *Note: It seems that no harm is done to pups if a host on more than one network is known by several names, one for each of those networks.*

The destination port address should be specified by the source process, of course. The source port address, however, should be specified as it becomes defined. The source socket number is dropped into the pup as it passes from source process to local NCP. The source host number should be dropped in as the pup leaves the source host. And the source area/network should be dropped in if and when the pup leaves the source area/network. This way, Taft points out, a process need not know its location to operate. The assignment of source indicators in this way also provides the basic primitive of network access control, incrementally-trustable source identification. Such prevents rampant impersonation.

Simonyi and Fiala both suggest that 32 bits is much too much for socket numbers.

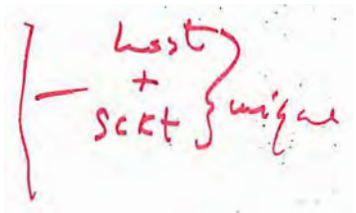
They argue that no host will ever have that many *active* ports; and they are right. They further suggest that 8 bits would be better.

Duvall, Taft, and we, on the other hand, think 32 bits is better. In Tenex, for analogy, many more files can be *named* than will ever exist. Many fewer files can be *JFNed* than can be named. Our question is roughly whether the socket number is a port *name* or a port *JFN*.

To make 8-bit socket numbers work, a distributed directory service (like Tenex's GTJFN or AT&T's 411 information operator) would be needed. We suggest that, in any event, it would be nice to have directory processes which map ascii strings into port addresses on demand. With such directory processes, a service running on some machine in some network on one particular day could grab a socket from its host, bundle it up, and send it with the service's name to the several directory service processes. Like listing a phone number in the yellow pages. This would give us location independence.

The argument for shortening socket numbers is based on the desire to shorten socket tables and to reduce pup overhead. The arguments against changing from 32 bits are based on the desirability of our not depending on the existence of directory processes and on compatibility with Noxios, Tenex MCA code, and the Arpanet.

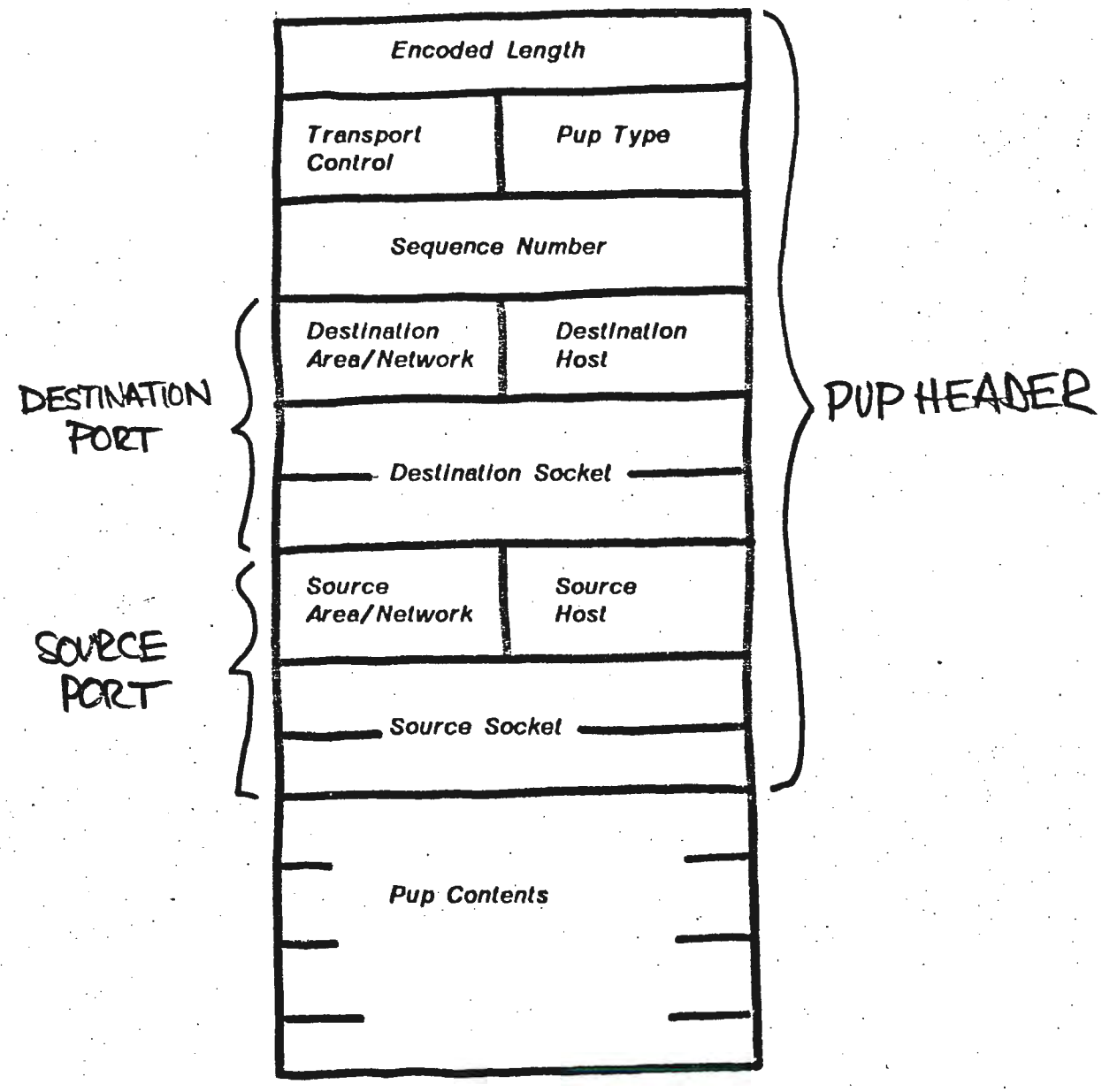
Note: We suggest that Bill Duvall's use of socket numbers is not quite the same as that intended for pups. In particular, pup sockets are host-relative; a pup socket number helps to specify a location, not a movable process name.



Taft points out that having large socket numbers reduces the probability of random garbage or "half-closed connections" messing up somebody's communication.

2.6 Contents. The definition of a pup is silent about the contents of a pup. So-called higher-level protocols, like the Byte Stream Protocol (BSP) to follow below, go further by putting meaning on certain pup contents.

Figure 1. Pup Format



2.7 Pup Problems. There are some problems which need further study. One is that of the largest pup to be permitted. And another is that of finding appropriate time-outs.

The Localnet carries a small fixed-length packet of 512 bits, some of which are used for control information. First, it may be we need to adopt a maximum size for pups so they'll fit in a Localnet's packets; this would spell doom for raw packet efficiency, especially in the lower-performance networks where it counts. (512-(80+144+32))/512 is 50%, right off the top. Or second, we could leave it to the Localnet (and others) to come up with its own schemes for fragmenting large pups so long as the pups are put together so as to appear untouched to the destination process. Or third, and this seems the most attractive right now, we could stick those communicating processes looking for high bandwidth with the job of deducing the largest pups they can use, with brute-force trial-and-error. Suggestions?

Time-outs are central to our brand of reliable communication. If time-outs are chosen too small, our networks become cluttered with duplicate, successfully delivered packets. If time-outs are chosen too large, our networks sit idle, with everyone twiddling their thumbs. Choosing good values for time-outs is complicated somewhat when the transporting mechanisms are unknown, as for pups. A pup may just take a hop through one 50 Mbps circuit in a directly-connected Localnet, or through an earth-orbiting satellite in the Arpanet, to use the extremes.

We suggest that our programs, to the extent they want to improve the efficiency of their use of the pup system, attempt to zero in on optimal time-outs (with round-trip time measurement and time-out adjustment) in the course of their various communications; a mild form of the trial-and-error approach, again.

2.8 Pup Gateway Routing. A process instructs its operating system to transmit a pup. The pup is bundled up in the local gateway, often called the NCP, and its destination address examined. If the area/network field of the destination port is seen to match that of a locally connected network, then the pup is appropriately encapsulated and sent to the indicated destination host. If the area/network field of the destination port does not match that of a locally connected network, the gateway must consult its own routing table to discover how to get the pup on toward its destination, to discover which host on a directly connected network is in a position to take the pup into another area/network closer to the destination. Gateway routing tables need only contain a number of entries equal to the number of existing interconnected networks, not one for each host or (*ugh!*) process.

A gateway could route a pup into that great bit-bucket in the sky. There are several reasons why a packet might be discarded and lost at a gateway: (1) nobody here by that name, (2) pup too big to fit in transporting packet, (3) congestion too high right now, (4) can't get there from here, or (5) unrecoverable transmission error. A source process can't, no, shouldn't care which of these strikes his packet down; the recovery procedure is the same, just time-out and retransmit. It is possible, of course, that we might choose to define some pup types for sending by a gateway to a source process for reporting the cause of a pup's demise, but this would be a somewhat complicating choice to be made reluctantly later.

3.0 Pup Transport through an Ethernet

By way of example, here is how we propose to carry a pup in the Ethernet.

First will come the required 16 bits of Ethernet addressing data. A destination and source must be specified in each Ethernet packet, 8 bits for each. The source field is fixed by the gateway's host address on the transporting Ethernet. The destination field is to be derived from the pup's 48 bits of destination address; it might be the specified host number if the area/network matches that of the current transporting Ethernet, or it might be the Ethernet address of a host willing to forward the pup onward to its intended destination elsewhere.

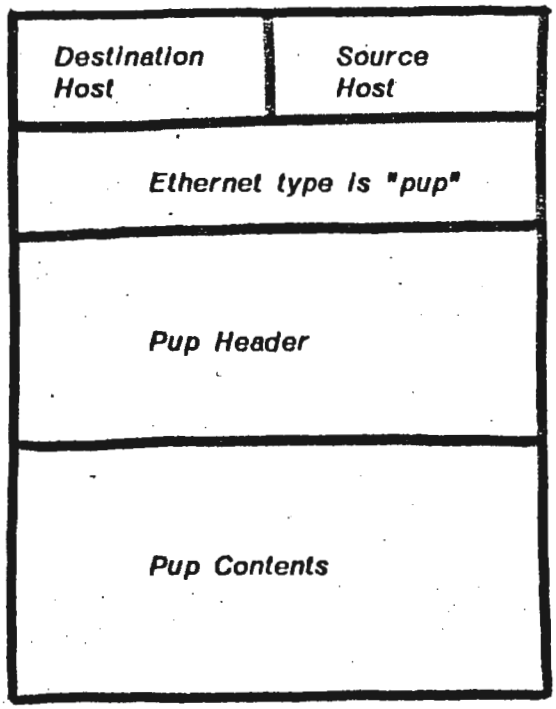
Next comes the Ethernet's own standard type word with the specific type "pup". This type is used by the receiving Ethernet's host's NCP to recognize the packet as a pup and to give it the gateway handling it deserves.

> *Note: Such handling might be (1) nothing more than handing it off to the addressed port's process, or (2) perhaps routing and reformatting for injection into an MCAnet, or Localnet, or Arpanet, or (3) perhaps*

Immediate discard for any one of a number of reasons.

And next would come the pup itself, in its entirety, transported in the required number of 16-bit words.

Figure 2. Ethernet-Encapsulated Pup



4.0 Byte Stream Protocol

And now, as an example, we offer a 2-page protocol with which two processes would transfer an error-free, flow-controlled byte-stream at the maximum rate possible, using pups.

We define the use of the following pup types. A registry of such types should be kept. They are: RTS, STR, Data, End, Abort, Cack, Sack, and Nak.

We propose that each of these types carry as pup "contents" a trailing (optional) 2-byte software checksum word. If the trailing 2 bytes are non-zero, then they contain an add-and-cycle checksum over the pup contents minus checksum.

4.1 Connecting Up. First, some distinctions. (1) Each pup has a *source* and a *destination*. (2) Each byte-stream connection has a *user* and a *server*, often called an initiator and a listener. (3) Each byte-stream has a *sender* and a *receiver*. These are orthogonal descriptors; a process may be one or the other out of the above pairs, independently.

We are given two processes wishing to transmit a byte-stream from one to the other. A process port into one of these processes, the server or listener, is known to the other, the user or initiator.

The first action taken is by the server who registers with his local gateway that he wishes to be given pups addressed to a specified publically-known port.

> *Note: This port might, of course, be registered under the server's name at some directory processes.*

The next action is taken by the user who registers 1 or 2 ports of his own with his local gateway (NCP). He sends a request for connection (RFC) in a pup from his connection port to the known server connection port.

> *Note: He may, of course, have retrieved this server port name from some directory service.*

The RFC is either an STR (sender to receiver) or RTS (receiver to sender), depending on whether the user wants the byte stream to move to or from him. The RFC carries in it the user's byte-stream port (possibly not the RFC's source) and a byte-stream name.

> *Note: The byte-stream name might be used as a file name by a file transfer server.*

Upon getting the user's RFC, the server decides to serve the byte-stream transfer, establishes a (possibly) new port for the transfer, and sends a matching RFC (STR for RTS or RTS for STR, with copied sequence number and server-assigned name) to the user's RFC's source port. When the answering RFC is received by the user, the byte-stream connection has been initiated and all exchanges afterward happen between the user and server byte-stream ports.

> *Note: This simple exchange is a packet version of the Arpanet's lumbering connection-version Initial Connection Protocol (ICP).*

4.2 Data Packets. Bytes in the byte-stream are carried from stream sender to stream receiver in pups of type "Data". The pup sequence number indicates, starting at zero, the position of the data pup's first byte in the byte stream (with wrap-around on 16 bits). All but the last 2 of the pup's content bytes are the data bytes themselves. Then comes the (optional) 2-byte checksum. That's it.

> *Note: Maybe the checksum should be in the pup header?*

4.3 End Packet. The End packet is launched by the sender to indicate where the end of stream is. The 16-bit byte sequence number in the End packet is that of the first byte after the end of stream. And then, the End's checksum. The checksum is the only information carried in the contents of a pup BSP End packet; it should always be zero. Hmmmmmm.

4.4 Abort Packet. An Abort packet can be sent by either the stream sender or stream receiver at any time. Abort messages can be completely ignored, but everything will work more smoothly if they cause the stream transfer to stop immediately. Each Abort packet will carry a 16-bit code (from a registry of codes) for program processing and a text string suitable for *huperson* consumption. (Remember the Arapanet FTP?) And then would come the checksum, as usual.

4.5 Cack Packet. A Cack is a "cumulative ack". It travels from the stream's receiver to the sender indicating that, since the beginning of the stream, all bytes through the

indicated sequence number were received correctly. Also, as for the following Sack and Nak, the Cack carries an allocation for flow control. And a checksum.

An allocation is a 3-tuple recommending (1) the number of bytes per Data pup which the receiver is prepared to receive, (2) the number of Data pups which the receiver is prepared to receive, and (3) the number of stream bytes the receiver is prepared to receive, all as of the time of departure of the Cack. And then the standard checksum.

The purpose of the allocations carried from receiver to sender is to allow the sender to participate in the optimization of byte transfer. It should be remembered that the network itself may impose further restrictions (say maximum transportable pup size), not to mention those from the sender.

Ed Fiala suggests that this allocation scheme be augmented to include provision for rate-controlled data flow, like that used on the MCA now. Further discussion is required.

4.6 Sack Packet. A Sack is a "specific ack". It reports the successful arrival of the specified number of bytes at the indicated position in the byte stream. The Sack is a completely redundant message which can be sent by a helpful receiver to indicate the arrival of a Data packet carrying data past a hole in the stream; its purpose is to cut down on retransmissions. A sender who gets ahead on his transmissions -- has several packets outstanding at a time -- can use Sacks to avoid retransmitting certain sections of stream. Completely redundant; but maybe helps performance.

4.7 Nak Packet. Nak is a "negative acknowledgement". The Nak is a completely redundant packet which can be sent by a helpful receiver to indicate the known loss of a specified number of bytes at the indicated position in the byte stream; its purpose is to hasten the retransmission of data lost at the receiver due to congestion, lack of space, or the like.

Contributions?

XEROX Inter-Office Memorandum

To:	Distributed Computing	*Date:*	February 2, 1975
From:	Bob Metcalfe	*Location:*	Coyote Hill
Subject:	Pup Again	*Organization:*	Parc CSL

The purpose of this memo is to specify a standard with which to interconnect Parc's various packet-switching networks. Ed Taft, John Shoch, and I are discussing Pup and building Pup-based software for Maxc, Smalltalk, and Alto, respectively.

This memo updates *Pup Converging* of July 16, 1974. An earlier and more discursive version, *Pup Revisited*, was distributed on April 14. Both earlier memos are hereby obsolete.

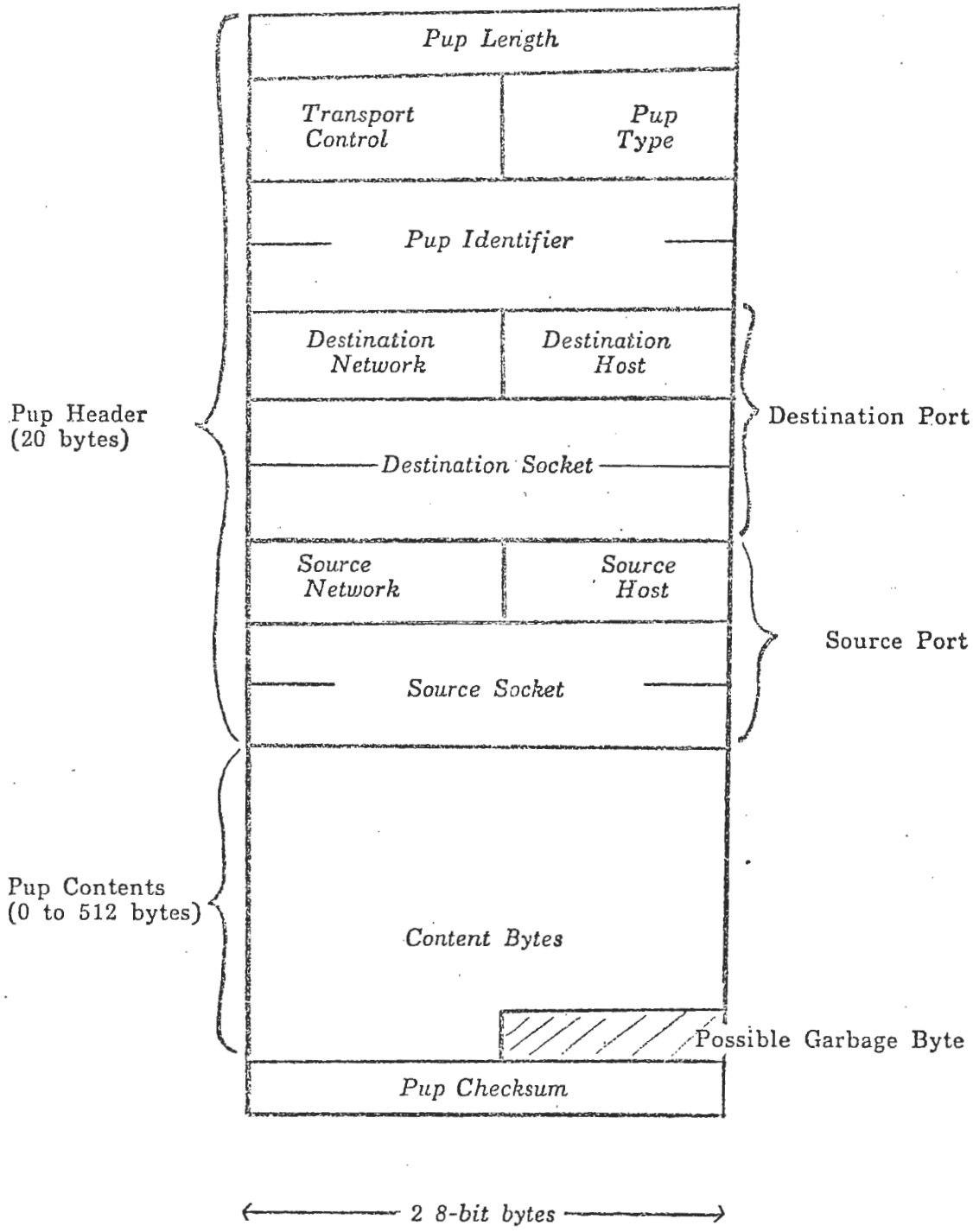
Taft's *Naming and Addressing Conventions for Pup* of January 9 and today's *Implementation of Pup in Tenex* should be read.

We propose a protocol, based on Pups (*Parc Universal Packets*), to allow processes on any of our interconnected computers to exchange packets through any of our interconnected networks. Of course, not all communications need use Pups. But, imagine being able to access the following resources from any process in Parc's diversely interconnected distributed computing system: Maxc and MaxcII for heavy computing, personal Altos for office communication, the XGP servers and Slot 3100 and Ears for printing, low-speed terminals coming in from the PT&T network for remote access, the Arpanet through various NCPs for very remote access, the distributed file system, the magnetic tape machines for backup, Polos editors and formatters, and so on.

The packet protocol being proposed consists of (1) the format of a standard (inter)network packet, the Pup, and (2) the principle that when a Pup is handed to the naked process-to-process packet transport system it can be expected to arrive undamaged at its destination *only* with high probability.

Also included in this memo are (3) how a Pup should be encapsulated for transport through an Ethernet, (4) a Pup Echo Protocol for test and diagnosis, (5) a Rendezvous Protocol, and (6) a Pup Byte Stream Protocol for carrying error- and flow-controlled byte streams between cooperating processes.

The Format of a Pup



The Pup

The *Pup Length* is the number of 8-bit bytes in the Pup, including header, contents, and checksum. There may be from 0 to 512 content bytes in a Pup so that the length will range between 22 and 534 bytes. A Pup is always carried in an integral number of 16-bit words. The number of 16-bit words is calculated by adding 1 to the length and dividing by 2. The number of content bytes is calculated by subtracting 22 from the length. When there are an odd number of content bytes in a Pup, the extra garbage byte required to fill the Pup out to an integral number of 16-bit words precedes the checksum word.

The *Transport Control* byte is for use by gateways; it should normally be zeroed at the Pup's source. Setting bit 0, the most significant bit, indicates that the Pup should be traced for debugging. Tracing might involve a gateway's recording a packet's passage or perhaps even a trace message sent to some monitoring process, perhaps the originating process itself. Bits 1-3 are used to count the number of gateways encountered during the Pup's transport. A Pup which reaches its 8th gateway will be discarded. Bits 4-7 currently have no use and should be zero.

The *Pup Type* is assigned by the source process for interpretation by the destination process; it is carried in the header for possible use by gateways. A type of 0 is illegal. Types of 128 and higher (through 255) are unregistered and indicate that gateways can't know anything about what's inside. Types 1 through 127 are registered types, such as the ones defined below, and they may get optional type-specific gateway handling. One example of type-specific gateway handling might be the virtual fragmentation of certain large Pups entering a network with a small maximum packet size. Another might be the generation of gateway trace reports on a packet of a type chosen for that purpose.

The 32 bits of the *Pup Identifier* are assigned by the source process for interpretation, relative to the Pup type, by the destination process. Pup IDs identify Pups and their contents to distinguish them from others, for purposes such as duplicate suppression and ordering.

All Pups originate at a *Source Port* and are routed to a *Destination Port*. Pup ports extend the addressing normally found in networks both for communication with distinct processes in a single host and for communication with hosts on networks other than the local one. A *Port* identifies one of 255 possible networks, one of 255 hosts in that network, and one of 2 to the 32nd minus 1 sockets in that host. No legal port has a 0 network, 0 host, or 0 socket.

The *Destination Port* is specified by the source process.

A network of 0 indicates that the Pup is addressed to a host in the current network; this is for use by processes which know that they want to communicate with a known host on their own network, but can't find out

which network they're on. We wish to avoid the situation in which processes on the same network can't communicate with one another because there is not an operating gateway at the moment. If the destination host is also 0, the process intends the Pup to be broadcast in the current network; this is for finding gateways. These conventions permit convenient communication among machines known to be on the same unidentified network and provide a mechanism for finding gateways. Processes will not be allowed to send Pups outside of their local network unless they know its identity. A process must be able to find out the identity of its local network from its local network control program which in turn must find out from a directly connected gateway. See Taft's *Naming and Addressing Conventions for Pup.*

The *Source Port* is verified enroute. The source socket is provided and/or verified by the source host's process interface. The source host and source network bytes are specified and/or verified by the first authority. Careful enroute assignment and verification of Pup source ports is the basis for (inter)network access control.

There can be from 0 to 512 *Content Bytes* in a Pup. Content bytes are carried in an integral number of 16-bit words. If the number of content bytes is odd, the last word is filled out with a garbage byte, not counted in the Pup's length.

If non-zero, the *Pup Checksum* is a 16-bit, 1's complement, add-and-cycle checksum computed over the 16-bit words in the Pup's header and contents. The sum is initialized to 0 and computed by repeated 1's complement addition and left cycle, starting with the Pup's Length word and ending with the last content word. Note that the checksum includes the garbage data byte if there is one. Checksumming is a likely candidate for implementation in microcode.

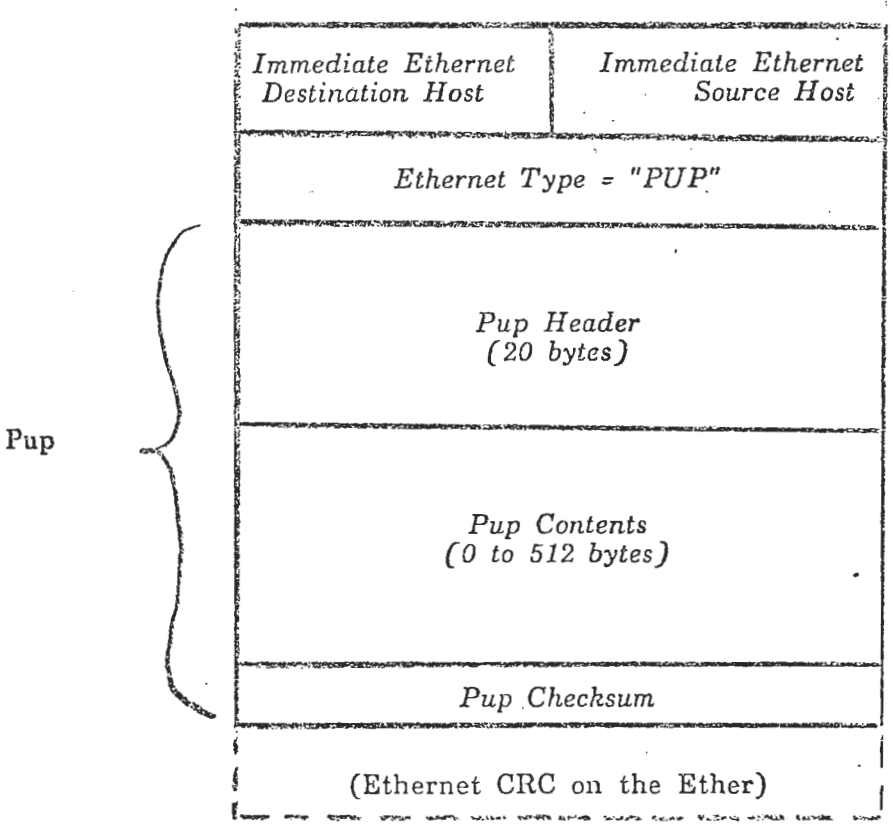
The Pup's checksum is carried with it from source to destination. If and when a Pup is altered enroute, say when the gateway hop count in the transport control field is incremented, its checksum must be recomputed. Our choice of the 1's complement add-and-cycle is intended to permit incremental checksum updating.

It can be assumed that the packet transport system will give its best efforts to the delivery of a Pup. It must be assumed, however, that Pups will be lost. If a Pup's checksum is checked enroute and found to be in error, the Pup may be thrown away without even so much as a trouble report. Pups may also be discarded in the event of a buffer shortage at any of the places it may pass. Many precautions will be taken to improve the chances of a Pup in getting to its destination, but no amount of machinery can assure trouble-free transport.

Ethernet Encapsulation

Pups are to be encapsulated to conform to the formats of a transporting network. It is unlikely but in the spirit of Pup encapsulation to rearrange various portions of a Pup for convenient handling; however, Pups must be seen by user processes and gateways as defined here. In the Ethernet, we will take the Pup as it is given to us, derive the local destination address as a function of the Pup's destination port, and carry it intact as shown.

The Format of An Ethernet-Encapsulated Pup



The *Immediate Ethernet Destination Host* byte is derived from the destination port; it will be either the destination host itself, or a gateway host through which the destination can be reached.

The *Immediate Ethernet Source Host* byte is the serial number of the host transmitting the encapsulated Pup through the Ethernet.

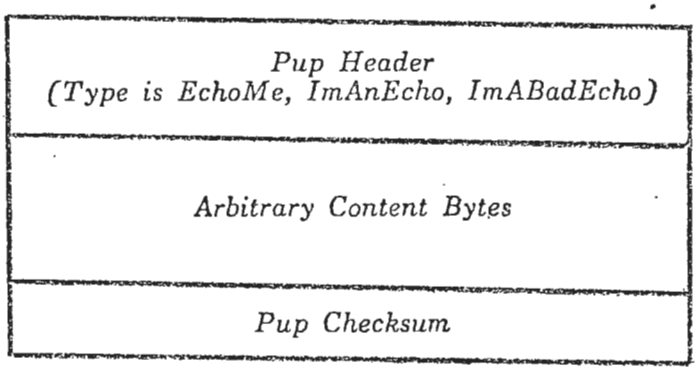
The *Ethernet Type* word identifies the packet as being a Pup so that it can be given Pup processing when it arrives at the immediate destination host.

The *Ethernet CRC* (Cyclic Redundancy Check) is shown to distinguish it from the Pup's checksum. Pups will get the Ethernet's kind of error checking while exposed to the Ethernet's kind of errors. The CRC is computed and checked totally in the bit-serial portion of the local source and destination Ethernet interfaces.

Echo Protocol

For test and diagnosis, a process receiving an EchoMe Pup may echo it. Before doing so, the receiving process should check the packet's checksum and the validity of the source port. If the packet checks out, it should be returned to its source as an ImAnEcho Pup with recomputed checksum. If it fails to check out, but there is evidence that its source is correctly identified, then the packet should be returned as an ImABadEcho Pup. Note that the source and destination ports must be switched, the gateway hop count zeroed, and the type changed; thus, the checksum must be recomputed. The process behind any port may choose to respond to EchoMe Pups according to the Echo protocol; it is expected that most hosts will have a process prepared to echo and that perhaps a certain socket on each host will be reserved for this purpose.

The Format of Echo Pups

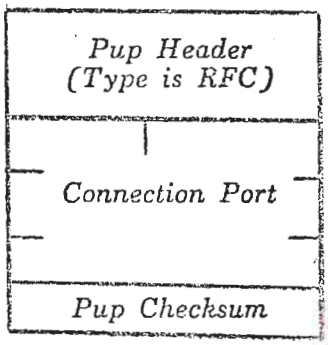


Rendezvous Protocol

Some terminology: Packets (including Pups) have a *source* and *destination.* A Rendezvous has a *listener* and *initiator.* Services have a *server* and *user.* Data, specifically byte streams, have a *sender* and *receiver.* These are independent descriptors. Usually the listener is also the server. The server may be either the sender or the receiver of a byte stream depending on the service being offered.

A Rendezvous is accomplished with an exchange of packets, each called an *RFC*, a Request For Connection.

RFC Pup Packet Format



RFCs may be exchanged to establish a connection between a user process and a server process. The ports between which RFCs are sent we call Rendezvous Ports. A user initiates by transmitting an RFC to a listening (server) rendezvous port. The listener confirms by returning an RFC with matching Pup ID. In addition to the rendezvous ports carried in the Pup header, each RFC carries the address of a port through which its source intends to maintain the newly established connection.

The *Connection Port* is separately specified so that, for example, a server can handle requests for service at a widely advertised rendezvous port and provide the service concurrently to a number of users via a number of connection ports. The connection port can be the same as the rendezvous port and we expect this will be the case for using processes.

The Pup ID on the initiating RFC should be chosen to reduce the probability of confusion among connections established near in time; the identifier in the complementing and confirming RFC must match. If connection IDs were to be generated from an appropriate real-time clock, for

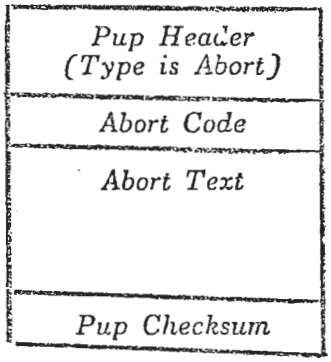
example, the probability of Pups from an extinct connection being mistaken for Pups in a new connection could be made vanishingly small.

If an initiator's RFC is lost, it will be retransmitted by the initiator after enough time has passed for a normal answer, say something like 1 or 5 seconds. Duplicate RFC's can, at best, be discarded on the basis of state information kept in the normal course of providing service; at worst, multiple servers will be generated to which no packets will be sent -- they will time out and destroy themselves.

The Abort Pup should be used to terminate a connection in the event of a detected catastrophe. A listener wishing to reject a rendezvous should try to send an Abort Pup with an explanation (e.g., "disk full" or "paper jam" or "tape busy"). Either end of a connection in progress, with its back against the wall, should try to send an Abort before self destructing, though its demise will eventually be detected anyway.

The Abort Pup carries a program interpretable code and a human readable explanation of some abnormal condition. An Abort can be sent to reject an RFC or to terminate a connection in the event of a catastrophe, say storage overflow or continuing checksum errors. An Abort must carry as its Pup ID the ID of the connection being aborted -- the ID of the connection's initiating RFC. Abort Pups need not be acknowledged because it is presumed that there would be nobody to receive the acknowledgment. Of course, receiving an Abort is something of a catastrophe and an Abort might be sent in seeming answer, just for completeness -- it would most likely arrive at an inactive port and be discarded.

BSP Abort Packet Format



The Abort code is for program interpretation and the Abort text is for human consumption. The codes should be registered and the text printable.

Byte Stream Protocol

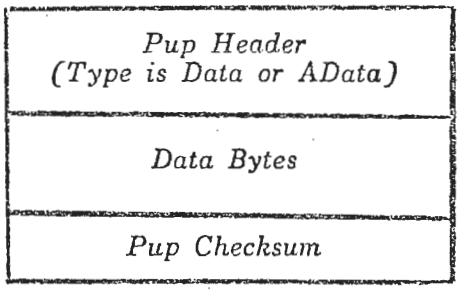
The error- and flow-controlled transfer of bytes between two processes is accomplished using a bi-directional byte stream maintained using the BSP. A byte-stream connection between two ports is established using the rendezvous protocol described above. The Pup ID used for the rendezvous will serve (1) as a connection ID for use in various connection-specific signals including Aborts (above), (2) an initial Interrupt Pup ID, and (3) an initial byte ID. To carry a byte stream from one port to another, data Pups carrying bytes identified with the increasing byte ID are transmitted from stream sender to stream receiver in return for acknowledgment packets with matching identifiers (though not neccessarily 1-to-1) which verify correct receipt and control flow. The streams of data flowing in each direction, while starting with the same initial byte ID, are independent.

Data packets should not be sent unless space has been allocated at the stream receiver. The sender is informed about receiver allocations in acknowledgment Pups returned by the stream receiver. These allocations are not additive; each one reflects the current state of the receiver's space allocation at the time of departure of its transporting acknowledgment. Acks and therefore allocations travel in both directions, independently for each direction of data flow.

In sending data Pups (below), the sender should comply with the allocation information in the most recently received Ack Pup.

The most recent allocation block indicates the maximum number of bytes per Pup that the receiver is willing to accept. Similarly, the number of Pups and number of bytes total which can safely be sent are limited. The maximum number of bytes per Pup and the number of bytes total are both expressed in terms of the number of *data* bytes, exclusive of the fixed-length Pup headers involved.

BSP Data Packet Format



There are two kinds of data Pup under the BSP, one which demands an immediate acknowledgment, called the AData Pup, and one which doesn't, called the Data Pup. All data must be positively acknowledged, but not on a strict packet-for-packet basis. It is intended that data will be transmitted in a number of Data Pups followed by an AData Pup asking for acknowledgment of receipt of all. The Pup Identifier of a BSP data packet is taken to be the ID of its first data byte.

We expect that null AData Pups (containing no data bytes) will be used to probe a receiver for an update of the allocation block and receiver byte ID. This will probably happen when a byte stream is first established and the sender has no allocation information or when the sender has been held up for some time with a zero allocation and wants to verify that the receiver is still alive.

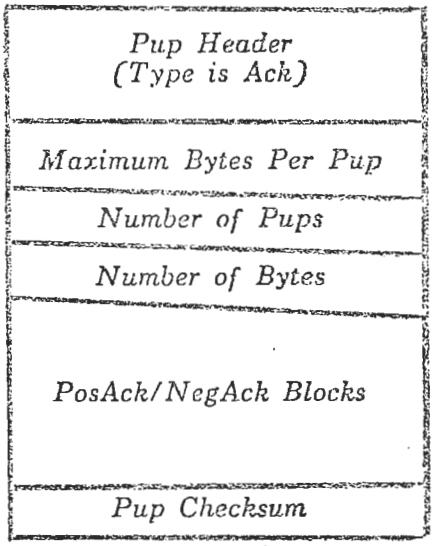
The Ack Pup indicates to the sender that all bytes previous to that identified by its Pup ID have been received correctly. Also, it carries a 3-word allocation block indicating the receiver's state as of the time the Ack was sent. The allocation should be interpreted relative to the Ack's ID, the byte ID of the first byte yet to be acknowledged. An allocation taken with the ID of its Ack specifies the ID of the last byte in the stream which the receiver is prepared to accept; this byte ID implied in Acks must be monotonically increasing so that stream senders need not take existing data Pups and break them apart. For the same reason, the maximum number of bytes per Pup cannot be decreased during the life of a BSP connection.

Data which have been transmitted but not acknowledged must be retransmitted after some timeout. If there are too many retransmissions, a stream connection may be aborted.

Optionally, an Ack Pup can carry up to 85 PosAck/NegAck Blocks. Each is a 3-word item indicating that a specified interval of bytes in the unacknowledged part of the byte stream is known by the receiver to be either received or lost. The purpose of these indications is to hasten the retransmissions of bytes known to be missing and to avoid the retransmission of bytes already received. Once a receiver indicates with a PosAck Block that an interval of bytes has been received, the sender may discard the packets which contain them knowing that they will not require retransmission. The transfer of bytes from stream sender to stream receiver should not depend on these PosAck/NegAck Blocks being used by either end, except that bytes might flow with less efficiency. A receiver may choose not to include PosAck/NegAck Blocks in its Ack Pups and a sender may choose to ignore them if present.

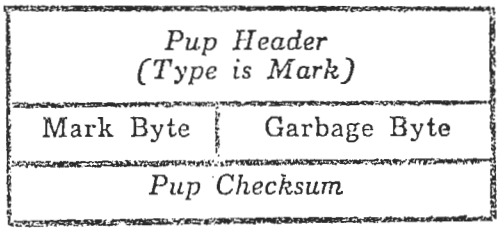
Each 3-word PosAck/NegAck Block begins with a bit indicating whether the interval is known to be missing or received -- a 1 indicates that the bytes in the interval have been received. The next 15 bits hold the number of bytes in the interval. And the next 2 words hold the byte ID of the first byte in the interval.

BSP Acknowledgment Packet Format



The Mark is a distinguished byte in the byte stream. It is analogous to the file mark found on magnetic tapes. The length of a Mark Pup is always 23. It carries one content byte which indicates which of a possible 256 types of mark is being signalled.

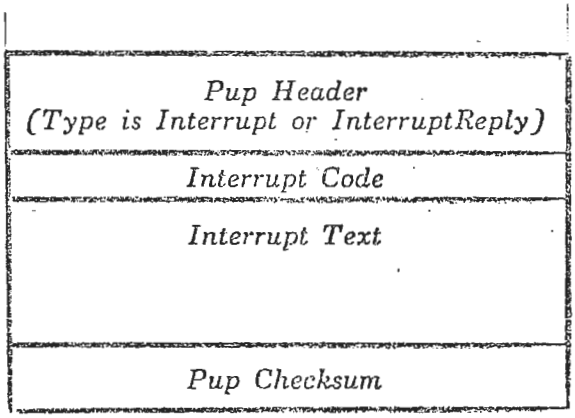
Mark Pup Format



While reading the data from a stream, a user reads up to a Mark and is then signalled in much the same way as when reading up to an end-of-file. The type of Mark should then be accessible. After clearing mark status, the user should be able to read on in the stream. Marks are acknowledged as any other byte in the stream.

An Interrupt Pup is used to signal some asynchronous event requiring immediate action by the other end of a stream. The Interrupt Pup is not subject to BSP flow control allocations and jumps over any and all buffered data. Interrupt IDs are generated from the stream's Interrupt ID which is initially the connection's ID. Successive interrupts advance the Interrupt ID. We envision using the Interrupt Pup in conjunction with the Mark Pup for flushing buffered data from a stream in response to some abnormal condition. An Interrupt Pup should not be sent until the previous Interrupt Pup has been acknowledged with an InterruptReply Pup. When receiving an Interrupt Pup, it should be acknowledged with an InterruptReply Pup only if its ID is equal to or 1 less than the current Interrupt ID. If its ID matches the current Interrupt ID, the using process should be interrupted and the Interrupt ID advanced.

BSP Interrupt Packet Format



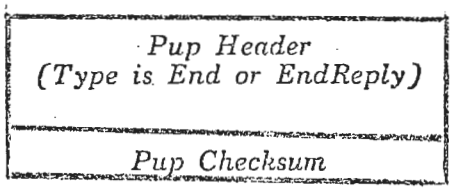
The end of a BSP connection can be initiated from either of its ports when the last of that port's outgoing data bytes has been flushed (i.e., transmitted and acknowledged). An End Pup with an ID matching that of the first byte beyond the end of the stream is sent. The End Pup must be retransmitted until an EndReply Pup is returned. The receiver of an End Pup responds, roughly, (1) by flushing any remaining outgoing data, (2) by returning an EndReply Pup, and (3) by dallying up to some time-out to re-respond to a retransmitted End Pup should its EndReply be lost. Upon receiving an EndReply Pup, the end initiator should send its own EndReply and promptly self destruct. If and when the dallying end of the stream connection receives its EndReply, it can immediately self destruct.

Upon receiving an End Pup, the using process should be notified, any remaining data flushed, and an EndReply sent. When notified that an End Pup has been received, the using process may elect to abort the connection as an expression of its dissatisfaction or proceed promptly (1) to send remaining data, (2) to retransmit data if neccessary while waiting for

acknowledgments thereof, (3) to send an EndReply in response to the initial and any subsequent End Pup, and (4) to dally until it receives an EndReply itself or times out.

Both the End Pup and the EndReply Pups carry the ID of that byte just past the end of the stream going in their respective directions. If an End or EndReply is received whose ID is not that of the next expected byte, the byte stream protocol has been violated. You will note that, in general, the ID of an EndReply Pup will not be the same as that of the End or EndReply Pup which it matches because each carries the current byte ID of the stream in its direction; this ID will not be the same except in the unlikely case that exactly the same number of bytes has been transmitted in each direction.

BSP End Packet Format

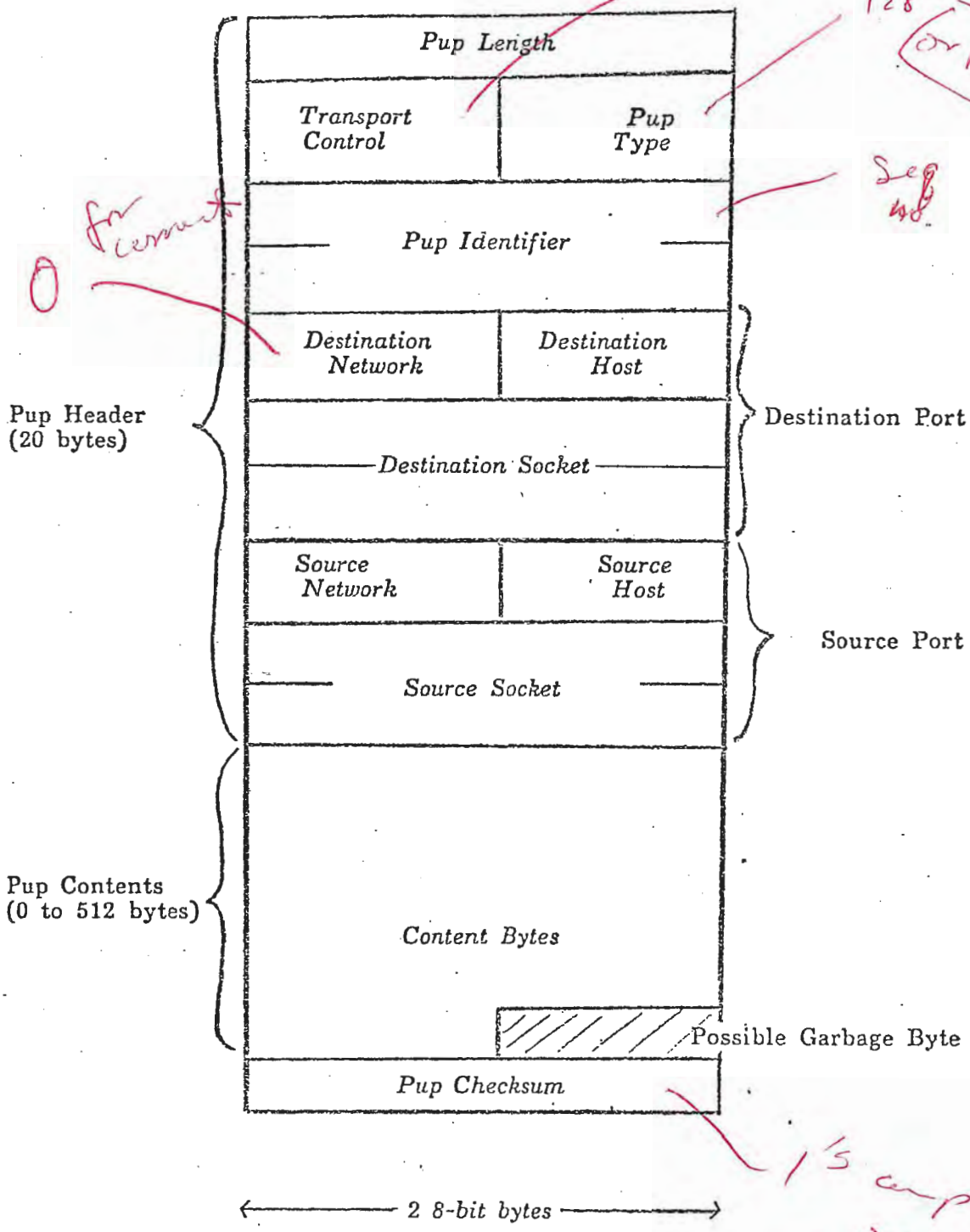


In the normal case, an End Pup and two EndReply Pups will be required to promptly close a connection. The receiver of an End Pup must dally after sending its EndReply just in case the EndReply is lost and the End is retransmitted. The longer the dallying period, the higher the probability that both ends of a stream connection will be able to agree on its normal termination. The purpose of the second EndReply, the one sent by the end initiator, is to attempt to notify the dallying end that it need not dally longer. Thus, in the normal case, three Pups and it's over; in a slightly less normal case, the dallying end will dally for its timeout wasting resources; in the arbitrarily unlikely case that the dallying end self destructs before an EndReply has been successfully received by the end-initiating port, the initiator will feel that the stream was terminated abnormally while the dallying end will feel everything went AOK. Stream connections can be normally terminated with certain agreement if, by convention, the ends agree to mark the stream in each direction immediately before closing it. In this case the Marks form the basis for agreement on clean termination and the end sequence is just a formality.

It may happen that both ends choose to send End Pups concurrently. Upon receiving an End Pup in seeming answer to an End Pup of its own, a port should (1) commit to dallying, (2) send an EndReply, (3) persist in retransmitting its End Pup until an EndReply is received in answer, and (4) then promptly self destruct.

For the record, here are the Pup type assignments: EchoMe=1, ImAnEcho=2, ImABadEcho=3, RFC=10, Abort=11, Data=20, AData=21, Ack=22, Mark=23, Interrupt=24, InterruptReply=25, End=26, and EndReply=27.

The Format of a Pup



Inter-Office Memorandum

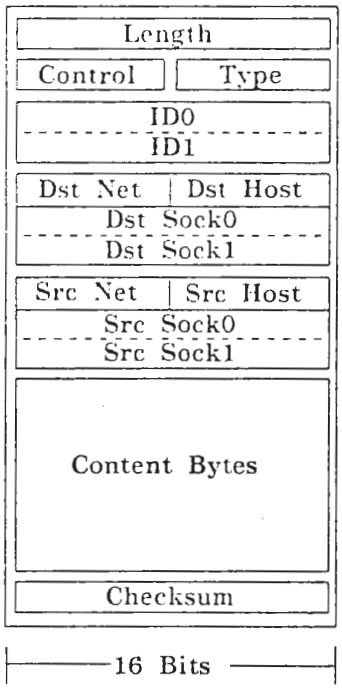
Bill Duvall

To	Distributed Computing	Date	15 January 1975
From	Bob Metcalfe	Location	Coyote Hill
Subject	Pup Format	Organization	Parc CSL

XEROX

Here is an update on the format of a Pup. The only change since my memo of July 16, 1974 is that the Pup Identifier is now 32 bits, up from 16. Pup overhead is now 22 bytes per packet, including the Pup Checksum. See Ed Taft's memo of January 9, 1975 for elaboration on Pup address fields. At present, we are intending that the Pup Checksum (if non-zero) be a 1's complement add-and-cycle over the entire Pup. I have started a registry of Pup types in <metcalfe>pupsyms.bc. A Pup's Length should be between 22 and 534 bytes.

Pup Format



Duvall

XEROX PALO ALTO RESEARCH CENTER
Learning Research Group

June 15, 1975

To: Distributed Computing

From: John Shoch

Subject: **A File Transfer Protocol Using the BSP**

This document represents an attempt to define a simple protocol for the movement of files among the many different computers currently connected to the PARC communications networks.

We expect that the ideas described in this pre-implementation draft will evolve in the light of experience gained when the protocol is implemented in several different environments (for MAXC, Alto/BCPL, Alto/Smalltalk, and Nova). The current version has benefited greatly from comments and ideas provided by Bob Metcalfe, Ed Taft, David Boggs, Bill Duvall, Bob Sproull, and others. Further comments and criticisms are welcome.

Assumptions and Basic Principles

---If you are reading this document, it is assumed that you are familiar with the contents of Bob Metcalfe's memo *PUP Again*, dated February 2, 1975, and Ed Taft's memo *Implementation of Pup in Tenex*, dated January 19, 1975.

---We are talking about files transmitted from one logical process to another logical process, and not necessarily from one file system to another file system. It is possible, for example, to generate a file 'on the fly' with one process and then have it consumed by another process, without it ever residing on a physical disk.

---We are not discussing particular programs which might chose to use the File Transfer Protocol, nor are we concerned here with implementation questions. Programs such as a File Transfer Program, a BCPL remote-stream facility, mail systems, or an automatic disk update program will, in all probability, conform to the File Transfer Protocol; but that is a matter of convention.

---It should be possible to construct a simple file tranfer protocol as a thin veneer placed over the Byte Stream Protocol (BSP). (Other, more sophisticated, features may emerge as part of the Distributed File System.)

---While the mix of machines and networks is very heterogeneous, these differences are subsumed by the definition of PUPs and the Byte Stream Protocol. The BSP is designed to provide "...error- and flow-controlled byte streams between cooperating processes." Thus, the only parts of the 'outside world' which the FTP will ever see -- or use -- are the facilities provided by the BSP.

---The File Transfer Protocol only comes into play after a BSP stream has been established. That is, actual programs will use the Rendevous Protocol (or some other means) to establish a connection between two ports. The FTP does not define this mechanism.

---One BSP connection should be sufficient for transmitting a file. Unlike the Arpanet protocol, both command information and data will pass through the same bi-directional BSP stream.

On the Nature of Files

It is common to say that a file is merely a collection of bits, but that is, or course, over simplified. A file is a collection of bits viewed in a particular way within a specified context. Much of the work involved in transferring files from one machine to another over some number of different networks is the result of attempting to resolve conflicting contextual representations of a file: 7, 8, or 36 bit, ASCII or binary, naming conventions, CR or EOL, *ad infinitum.*

In addition, such contextual differences make it hard to add a new machine or different network to a multi-network system without redoing the resolution matrix; and it is also important to avoid reformatting data to conform to local conventions at every step through a collection of networks.

To simplify the situation, this contextual information may be viewed as an adjunct part of a file, which might migrate with the file, as necessary. This information, or *file property list*, may be used to describe different kinds of "files", including those:

--residing on a physical disk;

--in transit, on a BSP connection;

--being sought by a particular user; or,

--being generated by a server.

The properties of a file are, in general, contained either explicitly or implicitly within an operating system on a particular machine. An actual property list may contain information provided by the user, from the file system, or from specialized information about the environment. Consider a typical scenario: moving a text file from MAXC to an Alto for editing with Bravo:

1. MAXC can generate a property list including the file name, date last written, author, protection, byte size, etc.

2. This information, along with the file, is transmitted to the Alto.

3. The Alto process does its best to map the property list into its environment: the file name is used, perhaps without the version number; protection information is discarded; the end of line indication may be changed, if necessary; etc.

There is no guarantee that a receiving process will be able to properly encode all properties.

A file could, however, attempt to preserve some of its original context by including as one of its properties an extraordinary request to maintain the

property list. For example, an Alto might be willing to save a property list on an auxiliary file, and use some of that information to regenerate a property list when the file is returned to MAXC. Thus, it is possible to preserve the protection information of a MAXC file which is temporarily used on an Alto.

A property list may be generated by any process which desires to transmit a file. Similarly, any process receiving a file and its property list can use that information to interpret the contents of the file. (Remember that all data moved under the BSP are sent as 8 bit bytes, and the property list will indicate any unusual packing of data.)

The property list might also be used to merely provide information about one or more files (where they are, when last written, etc.). In addition, a file property list can provide a partial description for a file which you are trying to find, or which you would like to create.

Typical Properties of a File	**Typical Values**
The minimal set of file properties	
Name	
Bytesize	7, 8, 16, 36
Format	Packed, Right-justified
Other potentially useful properties	
Size	
Original-author	
Current-owner	
Username	<for interacting with MAXC, etc.>
Userpassword	
Useraccount	
Protection	read-only, password-access
Type	ASCII-text, binary, Press-format
End-of-line-convention	CR, CRLF, EOL
Date-first-created	
Date-first-written-here	<if created elsewhere>
Date-last-modified-here	
Date-last-read-here	

Date-last-copied-from-here	
Location-of-original-source	(Ethernet)Palo, (ARPANET)Maxc
Location-where-copied-from	(Ethernet)Archive-machine
Location-history	
Last-writer	
Last-reader	
Last-copier	
Starting-position	<for partial read/write of a file>
Number-of-bytes	
Partial-write-mode	replace, insert, append
Data-compression-used	<data compression technique used>
Data-encryption-used	Lucifer, yes
Disposition	
Save-the-property-list	
etc.	

Syntax of a File Property List

A file property list consists of a string of printing ASCII characters, beginning with a left parenthesis and ending with a matching right parenthesis. Within that list, each property is represented similarly as a parenthesized list. For example:

((Name TESTFILE.7) (Bytesize 36) (Format Right-justified) ...)

This scheme has the advantage of being human readable, although it will require some form of scanner or interpreter.

[See Appendix I for a complete description of the property lists.]

The Protocol

In every FTP connection, one party is the User and the other party is the Server: the User initiates commands, while the Server provides replies. In general, there will be some form of reply generated in answer to every command.

Every transaction in the FTP will consist of a BSP Mark Byte, followed by a piece of data, and terminated at the next BSP Mark Byte encountered. The data thus enclosed might be one or more file property lists, a text

string, or a file. A command might simply be terminated by the arrival of the next Mark Byte command, or by an End-of-Command Mark Byte (essentially a no-op). All of the Mark Byte codes used in the FTP are listed at the end of this document.

For example, a simple retrieval of a file would include:

1. User sends the Server a command requesting the retrieval, with a file property list describing the desired file.

2. Server responds with an appropriate message.

3. If the Server said Yes, then the file is placed into the byte stream.

If a process replies with a "No" the first byte of the following data is a numeric code for possible machine processing, and the remaining bytes are a comment for the user.

This initial version of the protocol requires that while the file property list need not be complete, it must be sufficient to uniquely name a file (there is currently one exception: the Directory command). In a later version, it should be possible to view a property list sent with any command as a template, specifing a whole set of files to be transferred.

As a matter of convention, a user who wants to begin an FTP transaction will send a command indicating the version of the protocol which is implemented. The text string which follows might include the version of the program being used, for display to the user.

FTP Command Messages ([...] represents a Mark Byte)

The basic set of commands

[Retreive] <file property list> [End-of-command]

[Store] <file property list> [EOC]

[Yes] <reply code byte> <text string> [EOC]

[No] <reply code byte> <text string> [EOC]

[Here-is-file] <file data> [EOC]

[Reset] <text string> [EOC]

[Reset-reply] <text string> [EOC]

[Quit] <text string> [EOC]

[Comment] <text string> [EOC]

[I-am-version-n-who-are-you] <version byte> <text string> [EOC]

[I-am-version-n] <version byte> <text string> [EOC]

Additional commands

[Directory] <file property list> [EOC]

[Here-comes-the-full-property-list(s)]<file property list>...[EOC]

[Username] <text string> [EOC]

[Userpassword] <text string> [EOC]

[Useraccount] <text string> [EOC]

[Connect-to-directory] <text string> [EOC]

[Connection-password] <text string> [EOC]

Commands to another file system (these are not technically part of the protocol, but will probably be very useful, and thus are defined here)

[Create] <file property list> [EOC]

[Delete] <file property list> [EOC]

[Append] <file property list> <file property list> [EOC]

[Reset-properties] <file property list> <file property list> [EOC]

[Copy] <file property list> <file property list> [EOC]

File formats and conversion

When a file property list is sent to a server as part of an FTP command, it describes the file as the user would like to have it moved. The user may be willing to do some subsequent conversions, or may ask the server to perform the conversion. For example, if an Alto wants to retreive a text file from MAXC, but have MAXC do the conversion of CRLF to CR, then the Alto process might send the command:

[Retreive]((Name FOO.SR)(Bytesize 8)(End-of-line-convention CR))

Maxc might respond with a [Yes], indicating willingness to provide the file in that form; or Maxc might refuse to do the conversion.

It is expected that all implementations of the FTP should at least be able to store and retreive ASCII text files according to one standard convention:

(...(Bytesize 8)(End-of-line-convention CR)(Type ASCII)..)

(This convention is subject to further dissent.....)

Interrupts, Restarts, and Aborts

Either process may, if necessary, use the BSP convention for interrupting a byte stream: place a [Reset] Mark Byte in the stream at the current

position, and then signal a BSP Interrupt. This will clear out all pending data, and re-establish control at the top level of the protocol.

A process receiving an interrupt can then look for the corresponding Mark Byte in the incoming byte stream, but must also indicate the reset position in its outgoing byte stream, so that the two may regain synchronization. Thus, upon receiving a BSP Interrupt a process must signal a corresponding BSP Interrupt-reply, and drop a [Reset-reply] Mark Byte into the stream.

The protocol relies upon the BSP for establishing, terminating, or aborting all connections.

Other Remarks

1. Note that most implementations of the BSP provide buffering of the outgoing bytes, so it may be necessary to poke the Byte Stream in an appropriate manner in order to send a complete command.

2. All BSP connections are full-duplex, and it may be important for a process to monitor its input stream even while transmitting.

3. With the FTP, a user process may request a transfer of only part of a file: ((Name BACKUP.1) (Starting-position 2000) (Number-of-bytes 512) ...). The first byte of a file is always byte 0. Issues such as positioning backwards, leaving files open for efficiency, etc. are left to the program.

4. In general, third party transfers are not supported. There are several situations which we believe can be handled adequately within the current protocol:

a. A User asks a Server for a file, but the Server knows that the file is actually backed up on an archive facility. The Server can then 1) invisibly fetch the file from the archive, and then transmit it to the User; or, 2) the Server can send a [No] to the user indicating that the file lives elsewhere, allowing the User to ask for the full property list (including the file's actual location) and then establish a new connection to the archive. In either case, the Server will not try to magically force a connection between the User and the archive.

b. A User wants to take a file from the Server and print it on a different printing facility. The User can then 1) fetch the file from the Server, and then send it to the printer; or, 2) ask the Server to send the file to the printer in an appropriate way (this is especially simple on MAXC, where it need only be copied to another directory).

Ideas not implemented in this version of the protocol

1. An inquiry command might be useful, which would allow a process to ask another about its current capabilites, what's implemented, etc. The version number command in the current protocol is a small step in this direction.

2. "Parsing the property lists may be a burden on small machines, and it may require a significant amount of functionally identical code in different

machines and programs. Perhaps we should consider having a server somewhere which would parse a property list, and translate it into a machine-oriented form. Then a system not wishing to incur the parsing overhead could simply send off the property list to a server to be parsed, and receive a ready-to-use translation in return." (WSD)

Appendix I: Syntax of File Property Lists

A file property list consists of a string of printing ASCII characters, beginning with a left parenthesis and ending with a matching right parenthesis. Within that list, each property is represented similarly as a parenthesized list. For example:

((Name TESTFILE.7) (Bytesize 36) (Format Right-justified) ...)

This scheme has the advantage of being human readable, although it will require some form of scanner or interpreter.

When finally resolved, this section will define formats for:

delimiters

multiple attribute values [...(Owner Bob Hymie Icabod)...]

standard dates and time

numbers, decimal and otherwise

etc.

Appendix II: Registry of FTP Mark Byte Commands

Command	*Value*
[Retreive]	
[Store]	
[Yes]	
[No]	
[Here-is-the-file]	
[Reset]	
[Reset-reply]	
[Quit]	
[End-of-Command]	
[Comment]	
[I-am-version-n-who-are-you]	
[I-am-version-n]	
[Directory]	
[Here-comes-the-full-property-list(s)]	
[Username]	
[Userpassword]	
[Useraccount]	
[Connect-to-directory]	
[Connection-password]	
[Create]	
[Delete]	
[Append]	
[Reset-properties]	
[Copy]	

Appendix III: Registry of FTP Interrupt Codes

Interrupt and Interrupt-reply

Appendix IV: Commands, Replies, & Error Codes

Command	*Replies expected*
[Retreive]	[Yes]....[Here-is-the-file]....[]
	[No].....[]
[Store]	[Yes]....[]
	[No].....[]
[Here-is-the-file]	<no reply expected>
[Reset]	[Reset-reply]
[Reset-reply]	<no reply expected>
[Quit]	[Yes]....[] or no reply
[End-of-command]	<no reply expected>
[Comment]	<no reply expected>
[I-am-version-n-who-are-you]	[I-am-version-n]
[I-am-version-n]	<no reply expected>
[Directory]	[Here-is-the-full-property-list]....[]
[Username]	[Yes]....[] or [No]....[]
[Userpassword]	[Yes]....[] or [No]....[]
[Useracount]	[Yes]....[] or [No]....[]
[Connect-to-directory]	[Yes]....[] or [No]....[]
[Connection-password]	[Yes]....[] or [No]....[]
[Create]	[Yes]....[] or [No]....[]
[Delete]	[Yes]....[] or [No]....[]
[Append]	[Yes]....[] or [No]....[]
[Reset-properties]	[Yes]....[] or [No]....[]
[Copy]	[Yes]....[] or [No]....[]

Reply codes within [Yes] and [No]

[No]< >No such file[]

[No]< >Protection violation[]

[No]< >Read access not allowed[]

[No]< >Write access not allowed[]

[No]< >Bad format of file name[]

[No]< >Invalid Username[]

[No]< >Invalid Useraccount[]

[No]< >Invalid Userpassword[]

[No]< >Invalid Directory Name[]

[No]< > []

[No]< > []

[No]< > []

(The full set of error codes will evolve with the first implementations of the FTP)

XEROX PALO ALTO RESEARCH CENTER
Computer Science Laboratory

January 19, 1975

To: Distributed Computing

From: Ed Taft

Subject: **Implementation of Pup in Tenex**

This memo is a first attempt at a specification for implementation of Pup in Tenex. For now, we will limit ourselves to covering the implementation at a fairly cursory level and defer low-level specifications such as bit assignments and error codes to a later revision of this document.

Familiarity with Pup itself is assumed. Your feedback is welcome.

User Program Interface

Tenex user programs deal with Pup transmissions through the standard file system interface, in a manner similar to the Tenex Arpanet interface. That is, GTJFN is used to establish association between a JFN and a "network filename" (which evaluates to the local and foreign ports to be used). OPENF causes the local port to be created and (optionally) communication to be established with the foreign port. Data transfer operations come in two flavors: "raw packet" (DUMPI, DUMPO), and "byte stream" (BIN, BOUT, SIN, SOUT, etc). CLOSF causes the port to be released. And MTOPR is used for the remaining special device-dependent functions that don't seem to fit in anywhere else.

Network Filenames

A network filename (for GTJFN) is in the form:

PUP: [<local port> [#]] . [<foreign port>] [;T]

where "<" and ">" are meta-brackets and "[...]" denotes optional presence.

The *local* and *foreign port* specifications are as described in the recent memo "Naming and Addressing Conventions for Pup". They each evaluate to an 8-bit *network number*, an 8-bit *host number*, and a 32-bit *socket number*. Any or all of these elements may be zero or unspecified, which causes the port to be *wildcard* and subject to special handling as described below.

The *local port* specification must evaluate to a <network #, host #, socket #> triple that satisfies the following conditions:

1) The network and host numbers, if both specified, must correspond to an actual Maxc connection to some network. Ordinarily, user programs will leave these elements unspecified.

2) The socket number must be one accessible to the executing process.

Tenex divides the set of all possible 32-bit local socket numbers into three ranges, distinguished by the high-order 17 bits of the socket number, in a manner almost identical to the way in which Arpanet socket numbers are partitioned.

Sockets whose high-order 17 bits are zero are designated *system sockets*, and are accessible only to processes with wheel or operator capability enabled. These are intended for use by system server processes that listen on known, advertised sockets.

Sockets whose high-order 17 bits are in the range 1 to 99999 (decimal) are designated *user-relative sockets*. These are accessible to processes whose *connected directory number* is equal to those high-order 17 bits.

Sockets whose high-order 17 bits are 100000 (decimal) or greater are designated *job-relative sockets*. These are accessible to processes whose *job number* is equal to those high-order 17 bits minus 100000 (decimal).

The socket number in the local port specification, in conjunction with the "#" and ";T" attributes (if present), determine the actual local socket number to be used, in the following manner:

1) If the "#" attribute is present, the 32-bit socket number is completely determined by the socket number field of the local port specification. The use of "#" is required to refer to a system socket, and may also be used to refer directly to other sockets that are accessible to the executing process.

2) If the "#" attribute is not present, Tenex ignores the high-order 17 bits of the socket number given in the local port specification, and instead sets these bits as follows:

a) If the ";T" attribute is not present, the high-order 17 bits are set equal to the process's connected directory number, hence making a user-relative socket.

b) If the ";T" attribute is present, the high-order 17 bits are set equal to the process's job number plus 100000 (decimal), hence making a job-relative socket.

3) If the socket number in the local port specification is zero or unspecified, the low-order 15 bits are set to 8 times the JFN being assigned by GTJFN.

GTJFN, OPENF

The function of GTJFN for a network filename (as is the case for all other filenames) is simply to establish an association between that filename and a JFN. GTJFN therefore checks only to see that the network filename is well-formed. Operations such as verifying access to the local socket, creating the local port, and exchanging messages with foreign hosts are performed by OPENF.

At OPENF time, the following operations are performed:

1) Tenex checks that the specified local socket number is legally accessible by the executing process, and also ensures that it is currently unused. An error will result if the local socket number is equal to any other port's local socket number and the two ports have matching local network and host numbers. By "matching", we mean either that they are equal or that one or both of them are wildcard.

2) If the foreign port specification evaluates to more than one possible foreign port address (see the memo "Naming and Addressing Conventions for Pup"), a choice is made at this time by Tenex. Such a choice will be made on the basis of relevant parameters such as fewest number of intervening gateways between the local and foreign hosts.

A network file may be opened in two modes: *raw packet* mode (17 octal) and *byte stream* mode (0 or 6). We shall describe raw packet mode first since it is simpler.

Raw Packet Mode

In raw packet mode, entire Pups are transferred to and from the user address space, including Pup headers. Except for a small amount of checking and defaulting of fields in the Pup header, Tenex does no processing of the Pup. Acknowledgments, timeouts, duplicate message detection, flow control, and similar functions are left entirely up to the user program to perform. Tenex will discard both incoming and outgoing Pups that exceed resource constraints or produce other anomalies, with no error indication to the user process.

Executing OPENF in raw packet mode merely causes the specified local port to be created. No messages are transmitted by OPENF.

The DUMPI operation takes a command list (see JSYS Manual). For each command in the list, it waits (if necessary) for a Pup to be received whose "destination port" record matches the local port, and whose "source port" record matches the foreign port specification for this network file (with fields corresponding to wildcard elements ignored). The Pup is then simply dropped into the block specified by the user. (An error will occur if the Pup is too big to fit in the block).

Similarly, for each command in the list passed to DUMPO, a Pup is transmitted (after possibly waiting for Tenex to find some buffer space). In the case of DUMPO, Tenex first performs some minimal processing of the Pup header, as follows: First, the "Pup length" is checked for consistency with the actual size of the block given in the DUMPO command. Next, for any fields in the "source port" and "destination port" that are zero, Tenex substitutes the corresponding elements from the local and foreign port specifications. Nonzero fields of the "source port" and "destination port" are not touched. An error will occur if the resulting "source port" or "destination port" records still contain any zero elements or are inconsistent with the local and foreign port specifications.

Both DUMPI and DUMPO treat the "Pup checksum" as data. The user program is responsible for generating and/or checking the "Pup checksum" if desired.

Finally, CLOSF first waits for any buffered outgoing messages to actually be transmitted, then merely deletes the local port. Buffered incoming messages are lost, and any further messages received by Tenex for that port will be discarded without error indication.

This exhausts the primitives required for raw packet mode communication. Now we come to the interesting part.

Byte Stream Mode

Byte stream mode operates in accordance with the Byte Stream Protocol (BSP) described in Bob Metcalfe's most recent Pup specification, as amended by various unrecorded conversations Bob and I have had. Recent indications are that BSP is likely to change a lot in the near future.

The general idea of byte stream mode is that the user program simply reads and writes streams of pure data bytes, treating the network file as an ordinary sequential I/O medium. All BSP-specific operations (such as acknowledgments and flow control) are performed entirely by Tenex.

[Note: The question of whether BSP connections should be simplex or duplex has not yet been decided. In the following descriptions, we assume that a BSP connection supports only a single byte stream (one direction). A point to be considered is that Tenex supports only one sequential input or output stream per JFN.]

The operations performed by OPENF depend on two factors: (1) whether or not the foreign port is fully specified, and (2) whether the mode is 0 (normal) or 6 (immediate return).

If the foreign port is fully specified (i.e. contains no wildcard elements) and the mode is 0 (normal), OPENF first creates the local port (as for raw packet mode), then sends the appropriate RFC (Request For Connection) and waits for the answering RFC (if an Abort is received, OPENF gives an error return). Depending on whether the OPENF specifies reading or writing, the request sent is RTS or STR. The state of the port is then set to "Open" and control returns to the user program. If the local network and host numbers are unspecified by the network filename, Tenex assigns appropriate values (based on the destination port) before sending the first RFC.

In "immediate return" mode (6), the same operations are performed up through sending the RFC. At that point, the state of the port is set to "RFC outstanding" and control returns immediately to the user program without further waiting. Receipt of the answering RFC or Abort causes the port state to be changed to "Open" or "Abort" respectively. The user program may determine that this has happened by either polling the port state or arming the state-change interrupt.

Note that "normal" and "immediate return" modes differ only in whether or not the executing process waits while the protocol exchanges are taking place. In terms of internal events and messages exchanged with the foreign host, there is no difference between the two modes. Tenex takes care of properly acknowledging all messages received whether or not the user process is waiting for such acknowledgments.

If the foreign port is not fully specified but rather contains one or more wildcard elements, executing OPENF in either normal or immediate return mode does *not* send an RFC but rather puts the socket in a "Listening" state. When an RFC is received whose "source port" and "destination port" records match the foreign and local port specifications for this listening socket (with wildcard elements ignored), Tenex substitutes the actual values of all address fields in the Pup (both source and destination) for corresponding wildcard fields in the foreign and local port specifications, sends the answering RFC, and proceeds as in the normal case.

To summarize OPENF, there are four cases:

1) Normal mode, no wildcard: Send RFC, wait for reply.

2) Immediate return mode, no wildcard: Send RFC, return immediately.

3) Normal mode, wildcard: Put port in "Listening" state, wait for RFC, send answering RFC.

4) Immediate return mode, wildcard: Put port in "Listening" state, return immediately.

Once a port has been successfully opened, all sequential I/O operations are possible (BIN, BOUT, SIN, SOUT, etc).

Sequential output operations will wait only if necessary to obey flow control and buffering considerations. Normally, output will be buffered and blocked into maximum-length Pups where possible, but MTOPR function 21 may be used to force buffered data to be transmitted immediately (see JSYS Manual).

Sequential input operations will wait if necessary for data to arrive. An end-of-file condition will be signalled (generating an EOF pseudo-interrupt if it is enabled) upon attempting to read past a "Mark" or "End" packet. In the former case, SDSTS may be used to clear the end-of-file condition so as to allow reading the next logical file in the byte stream.

CLOSF for an output port will cause any buffered data to be transmitted, followed by a BSP End packet. It then waits for the acknowledging End, deletes the port, and returns. For an input port, if no End has been received an Abort will be sent; otherwise, the connection is just flushed with no messages sent.

Miscellaneous JSYSes

CVHST is extended to deal with PARC network addresses. CVHST takes a network number in LH 2, a host number in RH 2, and a socket number in 3, and produces a string of the form "(network)host-socket", with names substituted for numbers wherever possible. If the host number is zero, the output is "network", and if the socket number is zero, the output is "(network)host". If B0 of 2 is set, Tenex attempts to eliminate fields from the output wherever possible, by using identifiers that specify the values of more than one field.

MTOPR function 3 (write end-of-file), given for a port open for output in byte stream mode, will cause any buffered data to be transmitted immediately, followed by a "Mark" packet. By this means, one may separate multiple logical files transmitted sequentially over a single byte stream.

MTOPR functions 21 (force out all buffered data), 22 (send INS/INR), and 24 (assign interrupt channels) are implemented precisely as described in the JSYS Manual for Pup ports opened in byte stream mode.

MTOPR function 23 causes Tenex thereafter to discard occasional packets for the specified port (both incoming and outgoing) at random intervals. This facility is provided for the purpose of testing lost-message-detection and flow control algorithms.

GDSTS may be used to obtain certain device-dependent information about Pup ports. The port state and various status flags are returned in 2, including the "Mark" and "End" flags, which are set when the respective Pup types are encountered. The foreign network number is returned in LH 3, the foreign host number in RH 3, and the foreign socket number in 4.

Selected status bits may be changed by use of SDSTS. For example, the "Mark" flag may be turned off to clear an end-of-file condition and allow further input.

SIBE, SOBE, SOBF, DIBE, and DOBE are all implemented as described in the JSYS Manual. SIBE and SOBE return the number of buffered bytes for input and output respectively. SOBE skips and DOBE wakes up only when all data previously output has been successfully transmitted and acknowledged.

Port States

In the process of being opened and closed, ports go through several states. State changes are triggered by various events such as the issuing of JSYSes or the receipt of control Pups. A background Tenex process implements a finite state machine, by means of which all state changes take place.

User programs may monitor the state of a port either by polling it (via GDSTS) or by arming the state-change interrupt. Note that state changes triggered by events other than the execution of JSYSes may occur asynchronously with respect to operations performed by the user process.

The states currently defined are as follows:

Closed: No connection exists.

RFC Out: An RFC has been sent and a reply is awaited.

Listening: The port is listening for an RFC.

Open: RFCs have been exchanged and the connection is open.

Abort: An Abort has been received and the connection has been flushed abnormally.

End Out: An End has been sent but not yet acknowledged (send connections only).

End In: An End has been received and acknowledged, but the byte stream has not been completely read by the user process (receive connections only).

Sequential I/O is possible only when the state is "Open" or "End In". Attempts at sequential I/O at other times will result in an I/O data error pseudo-interrupt.

Telnet Protocol

Telnet is a higher-level protocol, entirely based upon the Byte Stream Protocol. We discuss it in this document only because server Telnet (network terminal) handling must be performed inside Tenex.

A Telnet connection is ordinarily established when a user process performs an ICP to the Telnet server on Maxc, which listens to socket 1 on all networks. After byte streams have been opened in both directions [possibly requiring two connections], they are "attached" to a network terminal by means of the ATPTY JSYS, described in the JSYS Manual. Thereafter, handling of both byte streams is performed entirely by Tenex.

When either byte stream is terminated, Tenex closes both streams, destroys the assigned port, releases the network terminal, and detaches the controlled job (if any).

For want of better suggestions, we adopt a subset of the Arpanet Telnet protocol for control of terminal connections, omitting most of its provisions (particularly the more complicated ones) for which there is no forseeable need within the PARC environment.

All Telnet control commands appear as data in the byte stream, preceded by the escape character IAC ("Interpret As Command", octal 377). All bytes that are not part of such a control sequence are treated as ordinary data bytes. Typically, Telnet data will consist only of ASCII characters in the range 0-177 (octal), but we leave the door open to the possibility of 8-bit binary communication with terminals, or to extended character sets. The data byte 377 is represented by two IACs in a row.

The byte following an IAC determines the control function to be performed. Arguments, if any, are given in succeeding bytes. The control functions to be implemented initially are as follows:

1) Data Mark (DM), in conjunction with the BSP "Interrupt" signal (INS/INR), is used to perform the "Synch" operation. A Telnet sender (user or server) desiring to generate a "Synch" signal inserts a DM in the byte stream and simultaneously transmits an INS. The INS, not being subject to BSP flow control, reaches the receiver even if the byte stream is otherwise clogged up due to allocation or buffering considerations. Upon receipt of the INS, the receiver immediately discards already processed data

as appropriate (e.g. clearing terminal buffers), and then reads the byte stream until the DM is encountered. While doing so, the receiver discards all data *except* Telnet control commands and any other "interesting" signals it may choose to act upon.

In Telnet terminal communication, the "Synch" signal is used for two similar purposes. First, it may be generated by the Telnet server when it is called upon the "clear the terminal output buffer". For example, the Tenex CFOBF JSYS, when executed for a network terminal, generates a "Synch" and thereby causes the user's terminal output buffer to be cleared and all data already in the byte stream to be flushed. Second, it may be generated by the Telnet user upon discovering a "terminal input buffer full" condition that is even preventing "abort" signals (e.g. control-C) from reaching the server because the byte stream is blocked due to lack of allocation.

2) Terminal Characteristics (TC) is sent by the user to inform the server of interesting characteristics of the user's terminal. Such characteristics might include page length, line width, and video or hard-copy. Upon receiving a TC control command, the Tenex server Telnet sets the corresponding local parameters for that network terminal. [More detailed design needed here.]

Arpanet Telnet functions not to be implemented (at least initially) include the following:

1) Network standard representations for common operations, such as EL (Erase Line), IP (Interrupt Process), etc. The justification for this is that within the PARC environment, we do not mind adopting the conventions of whatever server host we happen to be connected to. If users turn out to be unhappy about this, these functions are easy to add.

2) Negotiation about server or user echoing. Since we don't have any half-duplex terminals and always operate over short distances, it seems reasonable to adopt server echoing as the PARC standard.

3) The negotiation mechanism itself. The Arpanet Telnet "option negotiation" protocol is quite involved, and experience has shown it to be very difficult to implement correctly. When we find something to negotiate about, we should perhaps try to design something simpler.

4) More elaborate interactions such as the Telnet Reconnection protocol and the Remote-Controlled Echoing and Transmission protocol (again, due to lack of any forseeable need for these mechanisms).

Implementation

This section of the document is only an overview of the proposed implementation. More detail will be included in future revisions.

Pup Encapsulation

Pups are encapsulated in the Ethernet as described in Bob Metcalfe's memo. That is, the Ethernet destination and source hosts are in the first word (required by hardware); the second word contains 1000 (octal) as a signal that this message contains a Pup, and the remainder of the message is the Pup itself. (The Ethernet's hardware CRC word at the end is not part of the Pup).

In the MCA, Pups are preceded by a single word which is 100000 (octal) plus the length of the Pup in words. This convention allows messages containing Pups to be distinguished from messages conforming to the old MCA protocol. Pups may be split across physical messages in the same manner as is done currently for "logical" MCA messages. Refer to IOS-13 for relevant information.

In the Arpanet, Pups are encapsulated in standard IMP-Host protocol data messages on experimental links whose numbers are yet to be assigned. The 32-bit leader is immediately followed by the Pup.

The maximum length Pup that Tenex will send or receive on any network contains 512 data bytes.

Tenex/NVIO Communication

Tenex performs I/O to all networks through NVIO. We propose a protocol for Tenex/NVIO communication that minimizes the amount of development and maintenance that must be done for NVIO. This is for two reasons:

1) NVIO is already difficult to maintain. It is written in Nova assembly language, and its debugging and monitoring facilities are inferior to those available for Tenex development.

2) The Nova is eventually to be replaced by an Alto (in Maxc-II or possibly sooner), and whatever is added now will have to be recoded for the Alto.

This protocol requires no knowledge of Pup format on the part of NVIO. NVIO's role is limited to passing packets between Maxc and the various networks, and providing a moderate amount of buffering. Even forwarding of packets to other hosts (Maxc's gateway function) will be performed by Tenex.

For Pup input, Tenex passes NVIO the address of a maximum-length buffer in Maxc main memory. Upon receipt of a packet from any network (except the Arpanet), NVIO transfers that packet from its own buffers to the Maxc buffer. It then stores data in an overhead word in the buffer informing Tenex of (1) the number of 16-bit words in the packet, and (2) the physical source address of the packet (network and host). Finally, NVIO generates a Maxc interrupt, and it is done with this packet.

For Pup output, Tenex passes NVIO the address of a buffer containing a packet to be transmitted (with overhead word already set up). If NVIO has sufficient room to buffer the packet, it copies it and notifies Tenex that it

is prepared to accept further packets. Later, when NVIO has actually transmitted the packet or otherwise disposed of it, it passes the address of the Maxc buffer back to Tenex and generates a completion interrupt, after setting a field in the overhead word to indicate completion or error. NVIO will discard any packets that it cannot transmit within a short timeout (e.g. due to repeated collisions on the Ethernet or an unresponsive destination host on the MCA).

Pups to be sent over the Arpanet are encapsulated by Tenex and given to the Arpanet driver in Tenex for transmission over the existing Tenex/NVIO Arpanet interface. This is done to spare NVIO the necessity of multiplexing Pup and non-Pup Arpanet communication.

Maxc's Role as a Gateway

[Yet to be written.]

Backward Compatibility for MCA

It is expected that there will be a transition period during which Maxc will support Pup but Pogos (specifically Squid) will not. For the sake of making this transition as smooth as possible, a module will *temporarily* be included in Tenex supporting the old MCA protocol. This program will intercept non-Pup packets received from the MCA, and will provide a user interface to the MCA compatible with the existing one.

Specifically, the following will be supported:

1) Terminal connections to the MCA, as used by Squid and NEWMCA.

2) Sequential I/O to device MCA:, as used by Minx and the Pogos file server.

The present "Ether-MCA" protocol for the Ethernet will *not* be supported. This should not cause any inconvenience, however, since it is expected that Altos will support Pup before Maxc does.

Inter-Office Memorandum

To:	Distributed Computing	*Date:*	July 16, 1974
From:	Bob Metcalfe	*Location:*	Coyote Hill
Subject:	Pup Converging	*Organization:*	Parc CSL

XEROX

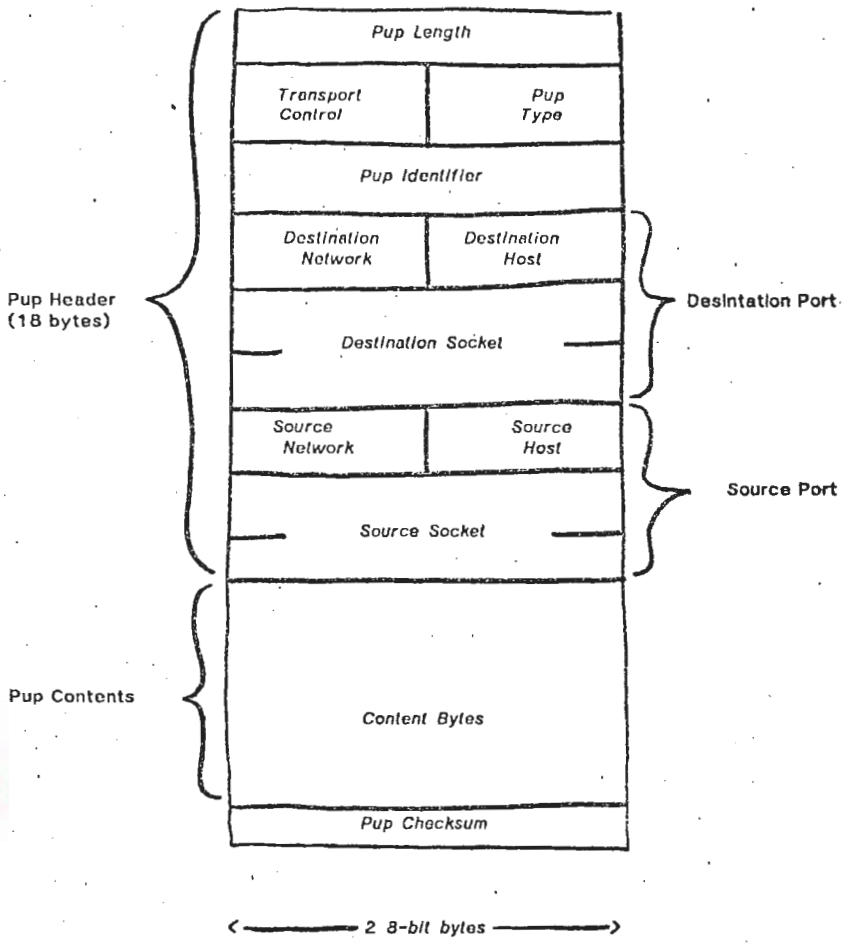
The purpose of this memo is to specify a standard with which to interconnect Parc's various packet-switching networks. The standard has been discussed and revised among a number of CSL and SSL people who have (1) a general interest in computer communication and/or (2) a pressing need for a satisfactory interconnection strategy for distributed systems now under construction. We hope that our assorted understandings of the so-called Parc Universal Packet (i.e., pup) are converging and that this latest, streamlined version of the specification is almost if not already suitable for use. Serious fleshing out and debugging are about to begin, subscriptions are being solicited, and suggestions are welcome.

A more discursive version of this memo, *Pup Revisited*, was distributed on April 14. Its predecessor, *A Proposed Pup*, was distributed on March 19. And a directly related memo, *Experimental Ethernet File Transfer Protocol*, was distributed on February 5. These earlier memos, though possibly useful in understanding pup's evolution, are hereby obsolete.

In short, we propose that a standard packet protocol, based on pups, be adopted to allow processes on any of our interconnected computers to exchange packets through any of our interconnected networks. Of course, not all communications need use pups. But, imagine being able to access the following resources from any process in Parc's diversely interconnected distributed computing system: personal computers for office communication, the XGP machines and Ears for printing, low-speed terminals coming in from the PT&T network for remote access, the Arpanet through various NCPs for very remote access, a Parc (distributed) file system, the magnetic tape machines for backup, Palos editors and formatters, and so on.

The standard packet protocol being proposed consists of (1) the format of a standard (inter)network packet, the pup, and (2) the principle that, when a pup is handed to the raw process-to-process packet transport system, it can be expected to arrive undamaged at its destination *only* with high probability, say 99 out of 100 times. Also included in this memo are (3) how a pup should be encapsulated for transport through an Ethernet, for example, (4) a pup Echo Protocol for testing purposes, and (5) a pup Byte Stream Protocol for carrying error- and flow-controlled byte streams between cooperating processes.

The Format of a Pup



The Pup

The *Pup Length* is the number of bytes in the pup, including header, contents, and checksum. The length must therefore be greater than 19 and less than 65,536. The number of 16-bit words occupied by a pup is calculated by adding 1 to the length and dividing by 2. The number of content bytes is calculated by subtracting 20 from the length.

The *Transport Control* byte is for use by gateways. Setting bit 0, the most significant bit, indicates that the pup should be traced for debugging. Tracing might involve a gateway's recording a packet's passage or perhaps even a trace message sent to some monitoring process, perhaps the originating process itself. Bits 1-3 are used to count the number of gateways encountered during the pup's transport. A pup which reaches its 8th gateway is to be discarded. Bits 4-7 currently have no use.

The *Pup Type* is assigned by the source process for interpretation by the destination process; it is carried in the header for possible use by gateways. A type of 0 is illegal, 1 means gatewnys can't know anything about what's inside, and the rest are registered types as, for example, the 13 types defined below.

The *Pup Identifier* is assigned by the source process for interpretation, relative to the pup type, by the destination process, in error and flow control. It identifies the packet to distinguish it from others, for duplicate suppression and ordering, say. It is anticipated that gateways will use pup identifiers, for example, in tracing and trouble reporting.

All pups originate at a *Source Port* and are routed to a *Destination Port*. A *port* identifies one of 256 possible networks, one of 256 hosts in that network, and then a socket in that host. There is no network 0; 1 is the Upstairs MCA; 2 is the Downstairs MCA; 3 is the Coyote Hill Ethernet; 4 is the Arpanet; and the others will be defined as built. No legal port has a 0 network, 0 host, or 0 socket.

The *Destination Port* is specified by the source process.

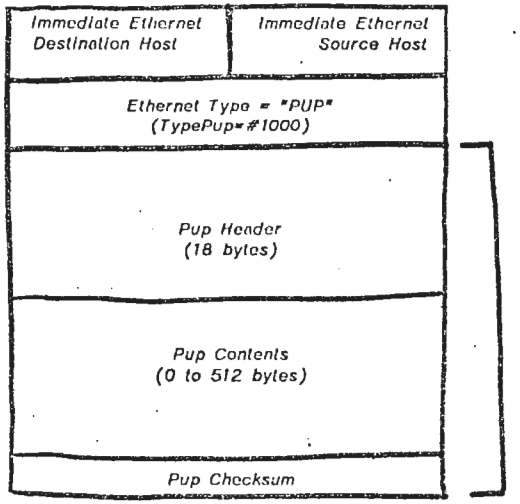
The *Source Port* is verified enroute. The source socket is provided by the source host's process interface. The source host and source network bytes are specified (verified) by the first authority. Careful enroute assignment and verification of pup source ports is the basis for (inter)network access control.

There can be from 0 to 65,515 *Content Bytes* in a pup. These bytes are carried in an integral number of 16-bit words. If the number of bytes is odd, the last word is filled out with a zero byte.

Not all, if any, networks will carry pups anywhere near as long as 65,515 bytes. We suggest that 512 bytes will be a useful maximum. The problem of fragmenting large pups for transport through networks which can only handle small ones is not well understood; it may be that processes desiring to use large pups will have to deduce the maximum size for their transmisssions at transfer time.

If non-zero, the *Pup Checksum* is a 16-bit add-and-cycle checksum computed over the pup's header and contents. We propose that this add-and-cycle checksum be microcoded for the standard Alto. It will prove useful for reliable software in general, pup in particular. A Nova can compute the add-and-cycle checksum in software at about 7 memory references per word. An Alto, in microcode, can do it in about 1, perhaps nearly 1/2.

The Format of An Ethernet-Encapsulated Pup



Ethernet Encapsulation

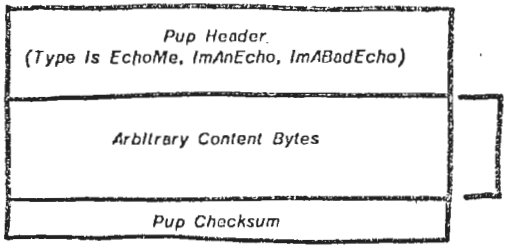
The *Immediate Ethernet Destination Host* byte is derived from the destination port; it will be either the destination host itself, or a gateway host through which the destination can be reached.

The *Immediate Ethernet Source Host* byte is the serial number of the host transmitting the pup through the Ethernet.

The *Ethernet Type* word identifies the packet as being a pup so that it can be given gateway processing when it arrives at the immediate destination host.

The Ethernet, until further notice, will compute the add-and-cycle checksum on all packets, including pups, as a part of standard operating procedure. This checksum will serve to detect problems in the otherwise unchecked parallel portions of transmission hardware and to measure the error control provided by the Ethernet's hardware CRC.

The Format of Echo Pups



Echo Protocol

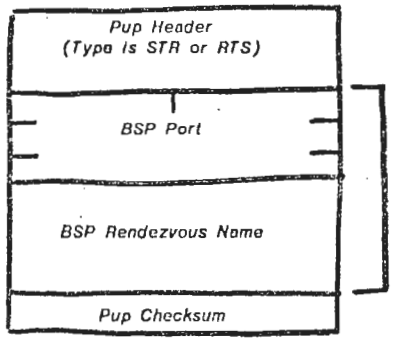
For test purposes, a process receiving an EchoMe (i.e., Pup Type 2) may echo it. To do so it should check the packet completely. If the packet checks out, it should be returned to its source as an ImAnEcho (i.e., Pup Type 3) with recomputed checksum. If it fails to check out, but there is evidence that its source is correctly identified, then the packet should be returned with recomputed checksum as an ImABadEcho (i.e., Pup Type 4).

Byte Stream Protocol

Some terminology. Packets (including pups) have a *source* and *destination*. Rendezvous have a *listener* and *initiator*. Services have a *server* and *user*. Byte streams have a *sender* and *receiver*. These are independent descriptors. Usually the listener is also the server. The server may be either the sender or the receiver of a byte stream depending on the flavor of the initiating RFC and the service being offered.

To carry a byte stream from one process to another, pups sequentially identified from 0 are transmitted from stream sender to receiver in return for packets with matching identifiers (though not 1-to-1) which acknowledge receipt and control flow.

BSP STR/RTS Packet Format



RFCs (Requests For Connection), an RTS (Receiver-To-Sender, Pup Type 5) and an STR (Sender-To-Receiver, Pup Type 6), must be exchanged to establish a byte stream under the BSP. A process initiates by transmitting either an STR or RTS to a listening port. The listener confirms by returning the complementary RFC. Each RFC identifies a port through which the byte stream is to flow and an optional (printable) rendezvous name.

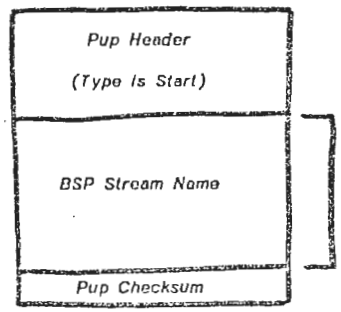
The *BSP Port* is specified so that, for example, a server can handle requests for service at a widely advertised port and provide the service concurrently to a number of users via utility ports.

The *BSP Rendezvous Name* can be used, for example, as a file or device name.

The *Pup Identifer* on the initiating RFC can be chosen at random; the identifier in the complementing and confirming RFC must match.

A polite listener wishing to reject an RFC should send an Abort (see below) with, perhaps, an explanation (e.g., "disk full" or "paper jam" or "tape busy"). Its pup identifier should match that of the RFC it's rejecting.

BSP Start Packet Format



The BSP Start (Pup Type 7) is the first packet to be transmitted from either of the BSP ports connected through the BSP rendezvous. It serves to establish a name for that part of the byte stream to follow and to get the receiver to return an acknowledgment with its allocation block.

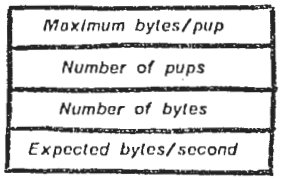
The *BSP Stream Name* might be used as a file name in a BSP file server, say.

Many Starts can be sent over the life of a byte stream; each starts a new piece of the byte stream, gives it a name, and causes the receiver to reestablish its allocation block.

A receiver may send a BSP Start in response to a BSP End so as to extend the byte stream. A Start from the stream receiver must be matched by a Start; from the stream sender, it must be Cacked.

The first Start to be sent must have a Pup Identifier of 0. All subsequent packets from the sender must have sequentially increasing identifiers.

BSP Allocation Block Format



Data packets should not be sent unless space has been allocated at the stream receiver. The sender is informed about receiver allocations in acknowledgment pups sourced at the stream receiver. These allocations are not additive; each one reflects the current state of the receiver's allocation at the time of departure of its transporting acknowledgment.

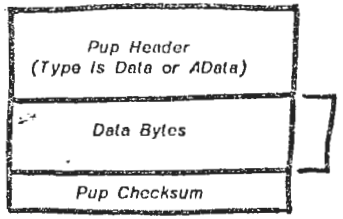
All three kinds of BSP acknowledgment (see below) carry an *Allocation Block*.

Each time a data pup is sent, the sender should satisfy and update the allocation constraints carried in the most recent acknowledgment of any type.

The most recent allocation block indicates the maximum number of bytes per pup that the receiver is willing to accept. Similarly, the number of pups and number of bytes total which can safely be sent are limited. These must be adjusted on the basis of pups thought to be in transit.

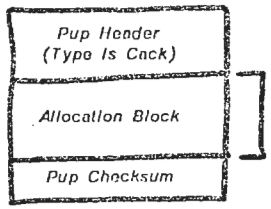
The allocation block may also contain the receiver's estimate of how quickly it is processing the byte stream so that the sender can project the current allocation from the most recent. A zero in the rate entry tells the sender that it must wait for allocations and not project allocations over time.

BSP Data Packet Format



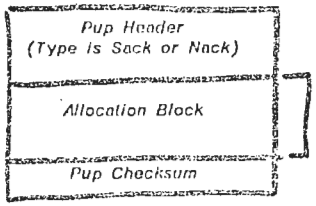
There are two kinds of data pup under the BSP, one which demands an acknowledgment (i.e., AData, Pup Type 8) and one which doesn't (i.e., Data, Pup Type 9).

BSP Cumulative Acknowledgment Packet Format



The Cack (Cumulative Acknowledgment, Pup Type 10) indicates to the sender that all packets, starting from the beginning of the stream, have been received correctly. The Cack's identifier is the identifier of the last packet of the received set. This includes Starts, ADatas, Datas, and Ends.

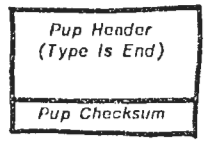
BSP Special Acknowledgments Packet Format



The Sack (Specific Ack, Pup Type 11) indicates that the packet identified in the pup header was received correctly; its purpose is to eliminate the need to retransmit the indicated packet when an earlier one is lost.

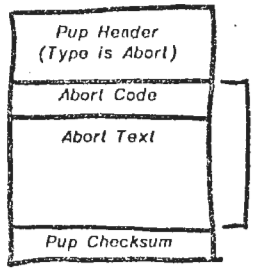
The Nack (Negative Ack, Pup Type 12) indicates that the packet identified in its pup header has been discarded by the receiver and needs to be retransmitted; its purpose is to hasten the retransmission of a packet known by the receiver to be lost, say because there was insufficient buffer space when it arrived.

BSP End Packet Format



The BSP End (Pup Type 13) calls for a normal termination of the byte stream. It is transmitted by the sender when there is no more data to send. The receiver can respond with a matching End which terminates the stream or with a Start which continues it (and must be answered with a Start which must be acknowledged).

BSP Abort Packet Format



The Abort (Pup Type 14) carries a program interpretable code and a human readable explanation of some abnormal condition. An abort can be sent to reject an RFC or to terminate a byte stream in the event of a catastrophe, say storage overflow or continuing checksum errors.

Comments?

DISTRIBUTION / DISTRIBUTED COMPUTING

BAUDELAIRE, Patrick	MCDANIEL, Gene
BORROW, Dan	MELVIN, John
BOGGS, David	METCALFE, Bob
DEUTSCH, Peter	MITCHELL, Jim
DUVALL, Bill	OSTRON, Eric
FLEIND, Jerry	PARISH, Vicki*
ENGLISH, Bill	RULIFSON, Jeff
FIALA, Ed	SHOUP, Dick
GUIBAS, Leo	SIMONYI, Charles
HOLT, Bayles	SPROULL, Bob
JEROME, Suzan	STURGIS, Howard
KAHRS, Mark	SWINEHART, Dan
KAY, Alan	TAYLOR, Bob
KIMBALL, Ralph	THACKER, Chuck
KUIPERS, Ben	WARREN, Wayne
LAMPSON, Butler	WEGBREIT, Ben
LAWS, Ben	WHITE, George

XEROX PARC CSL MEMORANDUM

January 9, 1975

To: Distributed Computing

From: Ed Taft

Subject: **Naming and Addressing Conventions for Pup**

This memo started out as a section in a forthcoming document describing a proposed implementation of Pup in Tenex. In reviewing the design, Bob Metcalfe and I came up with some unresolved internetworking protocol problems that are independent of any particular implementation.

The questions to be touched in this memo are:

1) How are networks, hosts, and processes to be *named*? That is, what form of identification is to be used by someone attempting to establish contact with some service or resource? It is desirable that the naming convention we adopt require little or no user knowledge of host or process numbering conventions or (inter)network topology.

2) How are hosts and processes to be *addressed*? At the lowest level, a *port address* is merely a <network #, host #, socket #> triple (a 48-bit number). However, a single host may be connected to more than one network or to the same network more than once, and a single process (or multiple identical instances of a process) may be associated with multiple ports on the same or different networks, possibly even on different hosts.

3) How does a host discover what network it is on and what hosts are gateways? How do we handle the degenerate cases in which a host does not know what network it is on and/or there are no accessible gateways? (These questions are of particular importance in the PARC environment because it is impractical to attempt to store the necessary information permanently in each host).

This memo attempts to discuss these issues in some detail and come up with a rough design. Your suggestions and comments are solicited.

Host Names

Most internetworking protocols, if they discuss naming conventions at all, specify a simple hierarchical scheme in which each network has its own name space, independent of the name spaces of other networks. In order to reference an internetwork process address, it is necessary to specify (1) the network name, (2) the target host name (whatever it is called on that network), and (3) the name (or number) of the process, port, or whatever on that host. For example, the internetwork name of the FTP server socket on Maxc might be "(Arpanet)Parc-Maxc-3".

In the case of the PARC networks, it seems desirable to adopt a somewhat different approach: that *a single name space encompasses all networks.* This really says two things:

1) A given host or process name refers to one and only one host or process (or possibly to a set of identical instances of a host or process).

2) For any given host or process that has more than one network address, all such addresses are associated with a *single name.*

Such a scheme has several benefits. First, the single name space allows users to specify a target host or process by means of a simple identifier rather than by a full pathname that assumes user knowledge of (inter)network topology. For example, to refer to the Ears printing service, it is necessary to specify only a single name (e.g. "Ears") rather than a full pathname such as "(Ethernet)Palo-1234".

Second, even if a host resides on more than one network, or a process listens on several ports on the same or different networks, it is unnecessary for the user to make a decision (specified though a full pathname) distinguishing between these choices. Since this is nothing more than a special case of message routing decisions that hosts and gateways must make anyway, it seems quite reasonable to leave such a choice completely in the hands of host and gateway software (unless there is a special reason the user wants to specify a full pathname).

Finally, in the case where there exist more than one identical instance of a particular host or process, it might be advantageous to give all such instances the same name. For example, if several hosts provide a "telephone directory" service, or several hosts are prepared to accept error statistics from Alto memory diagnostics, there is no reason not to give all instances of such a service the same name. One must take care in application of the "identicalness" criterion, however; e.g. it would be poor form if Ears output for a resident of Building 34 were actually sent to an "Ears" server in Building 31.

With these considerations in mind, we propose the following naming convention. The most complete possible port specification is in the form

(<network>) <host> - <socket>

where *network, host*, and *socket* may be either octal numbers or identifiers that evaluate to the appropriate quantities. Elements may be omitted if their values are implicitly determined by the identifiers given for the other element(s), as will be described shortly.

Association between *names* and *addresses* is performed by means of a *network directory.* This directory is maintained on Maxc and on other hosts that may wish to provide a directory lookup service.

Three types of names may appear in the network directory:

1) *Network names*, which each have a single value which is the corresponding network number. For example, the value of "Ethernet" is 3.

2) *Host names*, which each have a list of values consisting of <network #, host #> pairs. For example, the value of "Maxc" is the list (<1,1>, <2,1>, <3,200>, <4,40>) because Maxc's address on the upstairs MCA is 1, on the downstairs MCA 1, on the Ethernet 200, and on the Arpanet 40.

3) *Process names*, which each have a list of values consisting of <network #, host #, socket #> triples. For example, if an information service were implemented by processes listening to socket 1234 of Ethernet host 33 and socket 2345 of Ethernet host 55, the value of "Info" would be (<3,33,1234>, <3,55,2345>).

Additional information about each name and/or value might also be maintained in the network directory. Such attributes should be organized in the form of a property list, and might contain information such as the name of the owner of an Alto, or the physical location of a server such as Ears.

To illustrate, let us assume that the network directory contains the following entries:

Upstairs-MCA	1
Downstairs-MCA	2
Ethernet	3
Arpanet	4
Maxc	(<1,1>, <2,1>, <3,200>, <4,40>)
Dingleberry	(<3,33>)
XGP	(<1,5>, <3,2>)
Info	(<3,33,1234>, <3,55,2345>)

The following examples demonstrate how some typical port names might be handled:

(Ethernet)XGP-1234
Evaluates to (3)2-1234, since the foreign port is completely specified.

XGP-1234
In this case, there is a choice between (1)5-1234 (the upstairs MCA address of XGP) and (3)2-1234 (the Ethernet address). Both addresses are returned by the network directory lookup server, and it is up to local software (e.g. the NCP) to make an intelligent choice on the basis of criteria such as fewest number of intervening gateways.

(Arpanet)XGP-1234
Causes an error since XGP has no Arpanet address.

Info
Evaluates to a choice between (3)33-1234 and (3)55-2345. The choice could be made arbitrarily or on the basis of some relevant attribute returned by the network directory lookup service.

Dingleberry-Info or (Ethernet)Dingleberry-Info
Forces evaluation to (3)33-1234.

Port Addresses and Packet Destinations

Given a naming convention such as that just described, one might argue that a Pup should be allowed to have a *name* as a destination, or (an equivalent and more practical alternative) a *list* of addresses rather than a single address. Hosts and gateways would be free to route packets to any

one of the listed destinations, with the choice being made dynamically as network load or topology changed over time.

Such a scheme is unfortunately too cumbersome to be practical. Among other things, it would require the already over-long Pup header to be further extended. Furthermore, anomalies could arise in the case where a name referred, not to multiple addresses for the same process, but rather to multiple instances of a process.

It is the intent of this proposal, therefore, that the choice between alternative addresses be made at the time of the *initial connection* or *rendezvous* and remain fixed for the life of a "connection" (whatever we mean by that; e.g. as characterized by the Byte Stream Protocol). All packets are marked as arising from a single, specified source port and are delivered to a single, specified destination port.

Of course, the foregoing statement does not prohibit hosts from implementing ports (process connections) that will accept Pups from any of several sources or directed to any of several destinations. In fact, it is our intention to implement such a "wildcard" capability in Tenex.

Nor do we restrict the manner in which Pups may be *routed* between hosts and gateways. For example, if two hosts are connected to each other by multiple paths (as are Maxc and XGP), it is perfectly reasonable for XGP to actually transmit over the MCA a Pup whose destination address is (Ethernet)Maxc. In such circumstances, Maxc will simply invoke a special case of its gateway capabilities, namely to "route" the Pup from the MCA to its destination Ethernet port. The fact that the gateway and destination hosts are one and the same means that such routing takes place internal to the host, but in all other respects there is no difference between this and ordinary routing functions. This capability permits communications to continue without interruption should one of the networks go down.

Degenerate Cases

An important consideration in the PARC environment is that no hosts besides gateways should need initially to know what network they are on. Given the portability of Nova and Alto disk packs and even of entire machines, there is no practical way to permanently build into each host the network number(s) of the connected network(s).

This consideration leads to two requirements:

1) Intra-network communication should be possible without any of the hosts involved knowing what network they are connected to (though they must somehow know that they are connected to the *same* network). Such a situation might arise if the network were in fact not connected to any other network, or if all available gateways were down.

2) For the purposes of initiating inter-network communication, it should be possible for a host to (a) discover what network it is on and (b) discover what hosts on that network are gateways, and to what other networks.

To meet these requirements, we first set down the convention that no

network may have a network number of zero. A zero value in the source network number field of a Pup header is defined to mean that the sending host does not know what network it is connected to.

A user or user process desiring to initiate intra-network communication may transmit Pups to a specific host on that network, setting both host and destination network number fields to zero. In this case, the assumption is that somehow the initiator of this communication *knows* that the two hosts are on the same network. Such "knowledge" might be obtained in various ways. For example, the (human) user might simply direct his Alto user process to establish contact with Ethernet host 123, which is a specific machine which he *knows* is on the same Ethernet as his own machine.

Note that the network directory described earlier is (strictly speaking) not usable by a host that does not know what network it is on, even if it is able to access that data base. That is, having looked up "XGP" and discovered that its possible addresses are (1)5 and (3)2, we cannot do anything with this information since we don't know whether we are connected to network 1 or network 3 (or neither), and hence have no way of deciding whether to transmit to host 5 or host 2 on our "own" network, or even whether we can get there from here at all.

While this problem cannot be solved in general, there are various ways in which we can contrive to make life easier. For example, a host might maintain a local data base of names of hosts on its own network. (This local data base, in the case of Alto, could be saved from one session to the next by being written on a disk pack, and would also be useful in case the network directory lookup service is not accessible). Such a data base might be manually entered by the user, or might be automatically compiled by the host software from the network directory at times when the host *does* know what network it is on. Of course, this data base could be rendered invalid by changes in network topology, or by the user shoving his Alto disk pack into another machine on a different Ethernet.

All these problems go away if the host somehow discovers what network(s) it is connected to. Additionally, to perform inter-network communication, it is necessary for a host to know both what network it is on and what hosts on that network are gateways.

To this end, we first observe that all gateways *must* know what networks they are connected to. We then provide each gateway host with an *inquiry service* on a standard socket. A host desiring to find out its local network number and gateways then broadcasts an "Are you a gateway?" message to this socket at all (or a selected subset of) the hosts on its network. All the operating gateways on that network then reply with an "I am a gateway" message that includes the correct source host and network numbers in the Pup header. The body of the message might include routing information such as what other networks are accessible via that gateway, and by how many hops. With this information, the host now knows its own network number and may perform both intra- and inter-network communication in its full generality.

Bill Duvall

XEROX PALO ALTO RESEARCH CENTER
Computer Science Laboratory

February 2, 1975

To: Distributed Computing

From: Ed Taft

Subject: Implementation of Pup in Tenex

This is the promised revision of a memo with the same title dated January 19, 1975. Changes have been made to bring the proposed implementation in line with the revised Byte Stream Protocol, and both the user interface and the implementation are covered in much greater detail than before.

This memo assumes familiarity with the following documents which have been distributed by Bob Metcalfe and myself in the past month:

"Pup Again", February 2, 1975.
"Naming and Addressing Conventions for Pup", January 9, 1975.

Since I am about to begin implementation, I would appreciate receiving any comments or suggestions you care to make as soon as possible.

User Program Interface

Tenex user programs deal with Pup transmissions through the standard filo system interface, in a manner similar to the Tenex Arpanet interface. That is, GTJFN is used to establish association between a JFN and a "network filename" (which evaluates to the local and foreign ports to be used). OPENF causes the local port to be created and (optionally) communication to be established with the foreign port. Data transfer operations come in two flavors: "raw packet" (new JSYSes called PUPI and PUPO), and "byte stream" (BIN, BOUT, SIN, SOUT, etc). CLOSF causes the port to be released. And MTOPR is used for the remaining special device-dependent functions that don't seem to fit in anywhere else.

Network Filenames

A network filename (for GTJFN) is in the form:

PUP: [<local port> [#]] . [<foreign port>] [;T]

where "<" and ">" are meta-brackets and "[...]" denotes optional presence.

The *local* and *foreign port* specifications are as described in the memo "Naming and Addressing Conventions for Pup". They each evaluate to an 8-bit *network number*, an 8-bit *host number*, and a 32-bit *socket number*. Any or all of these elements may be zero or unspecified, which causes the port to be *wildcard* and subject to special handling as described below.

The *local port* specification must evaluate to a <network #, host #, socket #> triple that satisfies the following conditions:

1) The network and host numbers, if both specified, must correspond to an actual Maxc connection to some network. Ordinarily, user programs will leave these elements unspecified.

2) The socket number must be one accessible to the executing process.

Tenex divides the set of all possible 32-bit local socket numbers into three ranges, distinguished by the high-order 17 bits of the socket number, in a manner almost identical to the way in which Arpanet socket numbers are partitioned.

Sockets whose high-order 17 bits are zero are designated *system sockets*, and are accessible only to processes with wheel or operator capability enabled. These are intended for use by system server processes that listen on known, advertised sockets.

Sockets whose high-order 17 bits are in the range 1 to 49999 (decimal) are designated *user-relative sockets*. These are accessible to processes whose *connected directory number* is equal to those high-order 17 bits.

Sockets whose high-order 17 bits are in the range 50000 to 99999 (decimal) are available on a first-come-first-served basis to all processes.

Sockets whose high-order 17 bits are 100000 (decimal) or greater are designated *job-relative sockets*. These are accessible to processes whose *job number* is equal to those high-order 17 bits minus 100000 (decimal).

The socket number in the local port specification, in conjunction with the "#" and ";T" attributes (if present), determine the actual local socket number to be used, in the following manner:

1) If the "#" attribute is present, the 32-bit socket number is completely determined by the socket number field of the local port specification. The use of "#" is required to refer to a system socket, and may also be used to refer directly to other sockets that are accessible to the executing process.

2) If the "#" attribute is not present, Tenex ignores the high-order 17 bits of the socket number given in the local port specification, and instead sets these bits as follows:

a) If the ";T" attribute is not present, the high-order 17 bits are set equal to the process's connected directory number, hence making a user-relative socket.

b) If the ";T" attribute is present, the high-order 17 bits are set equal to the process's job number plus 100000 (decimal), hence making a job-relative socket.

3) If the socket number in the local port specification is zero or unspecified, the low-order 15 bits are set to 8 times the JFN being assigned by GTJFN.

The *foreign port* specification evaluates to a <network #, host #, socket #> triple in which any or all elements (including socket number) may be zero (wildcard). Tenex's handling of local socket numbers is not performed on foreign socket numbers; i.e. all 32 bits of the foreign socket number are always determined by the user's foreign port specification.

GTJFN, OPENF

The function of GTJFN for a network filename (as is the case for all other filenames) is simply to establish an association between that filename and a JFN. GTJFN therefore checks only to see that the network filename is well-formed. Operations such as verifying access to the local socket, creating the local port, and exchanging messages with foreign hosts are performed by OPENF.

At OPENF time, the following operations are performed for all data modes:

1) Tenex checks that the specified local socket number is legally accessible to the executing process (error: OPNX13, "Illegal access"), and also ensures that it is currently unused. An error (OPNX9, "File busy") will result if the local socket number is equal to any other port's local socket number and the two ports have matching local network and host numbers. By "matching", we mean either that they are equal or that one or both of them are wildcard.

2) If the foreign port specification evaluates to more than one possible foreign port address (see the memo "Naming and Addressing Conventions for Pup"), a choice is made at this time by Tenex. Such a choice will be made on the basis of relevant parameters such as fewest number of intervening gateways between the local and foreign hosts.

A network file may be opened in either of two principal modes: *raw packet mode* (17 octal) and *byte stream* mode (0 through 4). We shall describe raw packet mode first since it is simpler.

Raw Packet Mode

In raw packet mode, entire Pups are transferred to and from the user address space, including Pup headers. Except for a small amount of checking and defaulting of fields in the Pup header, Tenex does no processing of the Pup. Acknowledgments, timeouts, duplicate message detection, flow control, and similar functions are left entirely up to the user program to perform. Tenex will discard both incoming and outgoing Pups that exceed resource constraints or produce other anomalies, with no error indication to the user process.

Executing OPENF in raw packet mode merely causes the specified local port to be created. No messages are transmitted by OPENF. The OPENF may specify reading, writing, or both; typically, programs will open ports for both reading and writing.

Data transfers to and from the user address space are performed by means

of new JSYSes called PUPI and PUPO,[1] which transfer one complete Pup per command. Packets are stored in user memory in standard PDP-10 form, which is to say two 16-bit bytes (or four 8-bit bytes) per word, left justified. Unused bits in PDP-10 words are unspecified on input and ignored on output.

Both PUPI and PUPO treat the "Pup Checksum" as data. The user program is responsible for generating and/or checking the "Pup Checksum" if desired.

The calling sequence for PUPI and PUPO is as follows:

Accepts in	1:	B0: Never dismiss for I/O, give error return instead. RH: JFN
	2:	LH: Length of Pup in 36-bit words. RH: Address of first word of Pup.

PUPI or PUPO

Returns	+1:	Unsuccessful, error # in 1.
	+2:	Successful

The PUPI operation first waits for a Pup to be received whose "Destination Port" matches the local port, and whose "Source Port" matches the foreign port specification for this network file (with fields corresponding to wildcard elements ignored). The Pup is then simply dropped into the block specified by the user. If the Pup is too big to fit in the block, an error (PUPX1, "PUPI/O size error") will be generated and the Pup will not be lost.

Similarly, PUPO first waits for Tenex to find some buffer space, then transmits the Pup. In the case of PUPO, Tenex first performs some minimal processing of the Pup header, as follows: First, the "Pup Length" is checked for consistency with the actual size of the block given in the PUPO command (error: PUPX1, "PUPI/O size error"). Next, for any fields in the "Source Port" and "Destination Port" that are zero, Tenex substitutes the corresponding elements from the local and foreign port specifications. Nonzero fields of the "Source Port" and "Destination Port" are not touched. An error will occur (PUPX2, "Pup address error") if the resulting "Source Port" or "Destination Port" still contain any zero elements or are inconsistent with the local and foreign port specifications.

The substitutions just described are performed as the Pup is copied from the user block to internal buffers; i.e. PUPO does not modify user memory. It should also be noted that these substitutions will invalidate a nonzero "Pup Checksum". A program supplying a nonzero "Pup Checksum" must therefore fully specify the "Source Port" and "Destination Port" (thereby preventing any substitution by Tenex), and use these in computing the checksum.

---

[1] At one time we contemplated using DUMPI and DUMPO for raw packet I/O, but those JSYSes were found to be unsuitable for this operation due to excessive overhead and non-interruptability.

For either PUPI or PUPO, if B0 of 1 is set, an immediate error return will be given if attempting the operation would cause the process to block for I/O (error: PUPX3, "PUPI/O not possible now"). For PUPI, this can be due to there being no buffered input Pups to read, while for PUPO it is due to there already being too many Pups buffered for output but not yet transmitted by Tenex.

Finally, CLOSF first waits for any buffered outgoing packets to actually be transmitted, then merely deletes the local port. Buffered incoming packets are lost, and any further Pups received by Tenex for that port will be discarded without error indication.

This exhausts the primitives required for raw packet mode communication. Now we come to the interesting part.

Byte Stream Mode

Byte stream mode operates in accordance with the Byte Stream Protocol (BSP) described in Bob Metcalfe's Pup specification, "Pup Again" (also released today).

The general idea of byte stream mode is that the user program simply reads and writes streams of pure data bytes, treating the network file as an ordinary sequential I/O medium. All BSP-specific operations (such as acknowledgments and flow control) are performed entirely by Tenex.

The byte stream primitives provided by Tenex are as follows:

1) OPENF causes the establishment and initialization of a byte stream connection, and optionally performs a rendezvous.

2) BIN, BOUT, and higher-level sequential I/O operations cause pure, error-free byte streams to be read and written.

3) CLOSF causes the connection to be terminated in an orderly manner.

4) A collection of MTOPR functions are implemented for performing special operations peculiar to Pup and BSP.

An important property of BSP connections is that they are bidirectional. Unfortunately, Tenex is not capable of performing both sequential input and output independently over a single JFN. A program desiring to do bidirectional sequential I/O through a single port must obtain two JFNs for that port and open one for reading and the other for writing. The first OPENF performs the functions described below with respect to performing the rendezvous (if any) and establishing the connection, and the second OPENF is effectively a no-op (but is nevertheless required before the second JFN may be used for I/O operations). Similarly, two CLOSFs are required to disassociate the two JFNs from the port; the first performs all the operations necessary to close the connection, while the second does nothing.

The byte size specified by OPENF must be either 7 or 8 (error: SFBSX2, "Illegal byte size"). The 7-bit byte size is provided for convenience in

dealing with the usual Tenex 7-bit ASCII files; conversion between 7-bit characters and 8-bit Pup bytes is performed simply by adding a zero high-order bit on output and stripping off the high-order bit on input.

The OPENF used to establish the byte stream connection may specify any of five modes, numbered 0 through 4. These modes determine the operations to be performed by the OPENF and are indistinguishable thereafter. Of these five modes, four (modes 0 through 3) automatically implement special cases of the Rendezvous Protocol, while the fifth (mode 4) does not. In more detail, the various modes of OPENF operate as follows.

Executing OPENF in mode 0 or 1 performs a rendezvous in which the local port takes on the role of *initiator*. That is, after creating the local port (as for raw packet mode), OPENF transmits a Request for Connection (RFC) to the foreign port, designating the same local port (the *rendezvous port*) as being the *connection port* as well. In mode 0 ("normal"), OPENF then waits until the answering RFC arrives, changes the foreign port address for this connection to be the "Connection Port" designated by the answering RFC, sets the connection state to "Open", and returns control to the user program. In mode 1 ("immediate return"), the same operations are performed by Tenex as for mode 0, but control returns to the user program immediately after the initiating RFC is transmitted. The connection state is "RFC Outstanding" until the answering RFC is received, at which point the connection becomes "Open". The user program may determine that this has happened by either polling the connection state or arming the state-change interrupt.

In modes 2 and 3, the local port takes on the role of *listener*. Executing OPENF does *not* send an RFC but rather puts the local port into a "Listening" state. In mode 2 ("normal"), OPENF then waits, while in mode 3 ("immediate return"), it returns immediately to the user program. When an RFC is received whose "Source Port" and "Destination Port" match the foreign and local port specifications for this listening port (which may have wildcard elements), Tenex substitutes the actual values of the "Destination Port" and "Connection Port" fields (in the RFC) for the local and foreign port specifications of this port (respectively), sends the answering RFC to the foreign rendezvous port (from which the incoming RFC arose), and sets the state to "Open" as in the normal case.

Note that "normal" and "immediate return" modes differ only in whether or not the executing process waits while the protocol exchanges are taking place. In terms of internal events and messages exchanged with the foreign host, there is no difference between the two modes. Tenex takes care of properly acknowledging all Pups received whether or not the user process is waiting for such acknowledgments.

Wildcard elements in the *foreign port* specification are permitted only when "Listening" (modes 2 and 3); their presence in other modes will result in an error (PUPX2, "Pup address error"). On the other hand, the *local port* specification may (and usually will) have wildcard network and host elements. For modes 0, 1, and 4, OPENF automatically sets these elements to appropriate values (usually based on the foreign port specification). In modes 2 and 3, these fields are set from the "Destination Port" of the incoming RFC, as explained above. In any event, no port is permitted to

reach the "Open" state unless the local and foreign port addresses are completely determined.

For all four modes just described, the rendezvous performed is a special case of the Rendezvous Protocol in which the local *rendezvous port* and *BSP connection port* are the same. To permit other forms of rendezvous, OPENF mode 4 is provided which creates a BSP connection port without performing *any* rendezvous. When this option is used, the user program must supply the 32-bit *Connection Identifier*, right-justified in AC3. The local port is at once set to the "Open" state, and OPENF returns immediately.

When using mode 4, it is the responsibility of the user program to perform the rendezvous itself. It is intended that this be done by server programs (such as Telnet and FTP servers) that listen on a single, widely-advertised *rendezvous port*, and pass each separate request for service off to a different *connection port*. This is most easily done in the following manner:

1) Open the rendezvous port for I/O in raw packet mode, with the foreign port specified as completely wildcard.

2) When an RFC arrives, open a new BSP connection port (using OPENF mode 4), with the following parameters:

a) Local network and host equal to the "Destination Network" and "Destination Host" fields from the RFC;

b) Local socket selected to be unique;

c) Foreign port equal to the "Connection Port" in the RFC;

d) Connection Identifier equal to the "Pup Identifier" of the RFC.

3) Transmit (over the rendezvous port) an answering RFC containing the address of the BSP connection port just created.

Once a port has been opened successfully, all sequential I/O operations are possible (BIN, BOUT, SIN, SOUT, etc).

Sequential output operations will wait only if necessary to obey flow control and buffering considerations. Normally, output will be buffered and blocked into maximum-length Pups where possible, but MTOPR function 21 may be used to force buffered data to be transmitted immediately (see JSYS Manual).

Sequential input operations will wait if necessary for data to arrive. An end-of-file condition will be signalled (generating an EOF pseudo-interrupt if it is enabled) upon attempting to read past a Mark or End packet. In the former case, SDSTS may be used to clear the end-of-file condition so as to allow reading the next logical file in the byte stream.

CLOSF will first wait for any buffered output data to be completely transmitted and acknowledged. It then transmits an End (or an End Reply if an End has already been received) and waits until the 3-way "End" handshake has completed before returning to the user program. The port's state becomes "Closed" (meaning that the port itself has ceased to exist).

If at any time an Abort is received, the connection will be terminated abnormally. If the user program is executing an OPENF or CLOSF at the time, the port is simply placed in the "Closed" state and the JSYS error return is taken (error: OPNX21, "Rejected by foreign host"). At any other time, the port is placed in the "Abort" state, which is signalled to the user process by a state-change interrupt (if armed), or by an I/O Data Error interrupt upon the next attempt to perform sequential I/O. A subsequent CLOSF will reset the state to "Closed" without further network activity.

If, while a BSP connection is "Open", the user process is dismissed for I/O over that port and no activity occurs for 15 seconds, Tenex will probe the foreign port (by means of an AData, which demands an acknowledgment) to make sure it is still alive. This will be repeated every 15 seconds if necessary. If no response is received after 5 minutes of inactivity, the "Timeout" flag will be set in the port status word, generating an I/O Data Error interrupt. The user process may choose to ignore this condition by enabling the interrupt and clearing the "Timeout" flag (using SDSTS) during the interrupt routine. Ordinarily, however, a timeout will cause the process to terminate, since I/O Data Error is a "panic" interrupt.

A CLOSF executed while the "Timeout" flag is set will cause an Abort to be sent rather than an End, and will return immediately.

Operations initiated by CLOSF are also subject to being timed out. If the port remains in the same (non-"Closed") state for more than 30 seconds, an Abort will be sent and the error return will be taken (PUPX4, "Timeout during End handshake").

Ports in the "RFC Out" and "Listening" states are not timed out.

Miscellaneous JSYSes

CVHST is extended to deal with PARC network addresses. CVHST takes a network number in LH 2, a host number in RH 2, and a socket number in 3, and produces a string of the form "(network)host-socket", with names substituted for numbers wherever possible. If the host number is zero, the output is "network", and if the socket number is zero, the output is "(network)host". If B0 of 2 is set, Tenex attempts to eliminate fields from the output wherever possible, by using identifiers that specify the values of more than one field.

CVSKT is similarly extended to take a Pup JFN in 1 and return the local port parameters (network number in LH 2, host number in RH 2, and socket number in 3). The action of CVSKT depends on whether or not the port is open. If not, CVSKT merely parses the network filename string to yield a local port address. If the port is open, however, CVSKT returns the actual address of the local port, which may have some formerly wildcard fields (network or host) replaced by specific values.

MTOPR function 3 (write end-of-file), given for a port open for output in byte stream mode, will cause any buffered data to be transmitted immediately, followed by a Mark packet. The value of the Mark byte should be supplied right-justified in AC3. By this means, one may separate multiple logical files transmitted sequentially over a single byte stream.

MTOPR function 21 causes any partially filled output Pup to be transmitted immediately, rather than being retained until enough sequential output has accumulated to occupy the largest possible Pup (consistent with BSP allocation), as is done normally. This MTOPR does not, however, wait for all previous output to be acknowledged; that function is performed by DOBE (see below).

MTOPR function 22 causes an Interrupt packet to be generated.

MTOPR function 23 causes Tenex thereafter to discard occasional packets for the specified port (both incoming and outgoing) at random intervals. This facility is provided for the purpose of testing lost-message-detection and flow control algorithms.

MTOPR function 24 is used to arm or disarm various interrupt-causing conditions and to assign PSI channels to them. The interrupt channel assignments are passed in AC3, with a value of 0 to 35 (decimal) causing the condition to be assigned to the specified channel, and 36 or greater causing the condition to be disarmed.

B0-5 PSI channel for interrupt generated upon receipt of an Interrupt Pup (byte stream mode only).
B6-11 PSI channel for interrupt generated upon receipt of any packet for a port opened in raw packet mode. (This is handy for use in server processes that listen on several rendezvous ports at once.)
B12-17 State change PSI channel (byte stream mode only).

Note that the indicated channel and the rest of the PSI system must also be initialized and enabled and activated in the usual manner (see JSYS Manual). All conditions are initially disarmed when a port is created.

MTOPR function 25 generates an Abort and causes the port to be placed in the "Abort" state. If AC3 is nonzero, its contents are used as the "Abort Code" and AC4 is used as a string pointer to supply the "Abort Text".

MTOPR function 26 (legal only for a port in the "Abort" state) returns the "Abort Code" in AC3 and uses AC4 as a string pointer to store the "Abort Text".

GDSTS may be used to obtain certain device-dependent information about Pup ports. The port state and various status flags are returned in 2 (see below). The foreign network number is returned in LH 3, the foreign host number in RH 3, and the foreign socket number in 4.

Selected status bits may be changed by use of SDSTS. For example, the "Mark" flag may be turned off to clear an end-of-file condition and allow further input.

The port status word (defined only for ports open in byte stream mode) has the following format:

B0-3 Port state (see next section).
B4 Mark encountered. Generates EOF interrupt (if armed). Must be cleared by SDSTS to permit further input.
B5 End encountered. Generates EOF interrupt (if armed).

B6 Timeout (no activity for 5 minutes). Generates I/O Data Error interrupt. May be cleared by SDSTS.
B7 If set, generation and checking of Pup Checksums is suppressed. Initially clear, this bit may be set or cleared by SDSTS.
B10-17 Content byte of most recently encountered Mark.

SIBE, SOBE, SOBF, DIBE, and DOBE are all implemented as described in the JSYS Manual. SIBE and SOBE return the number of buffered bytes for input and output respectively. The number of "buffered input bytes" (for SIBE and DIBE) is defined to be the number of bytes that can be read without causing the process to block. Hence, it does not include bytes contained in any later Pups that may have arrived out of sequence. The number of "buffered output bytes" (for SOBE and DOBE) is similarly defined to be the number of bytes that have been transmitted (or are potentially transmittable) but not yet cumulatively acknowledged. Bytes contained in packets that have been specifically acknowledged out of sequence are therefore not considered to have been transmitted when making this count. Note also that the count does not include bytes contained in the packet currently being assembled; hence, MTOPR 21 (or perhaps MTOPR 3) should be executed before issuing a DOBE, if it is desired to dismiss until all bytes output up to that point have been successfully received and acknowledged.

Port States

In the process of being opened and closed, ports go through several states. State changes are triggered by various events such as the issuing of JSYSes or the receipt of control Pups. A background Tenex process implements a finite state machine, by means of which all state changes take place.

User programs may monitor the state of a port either by polling it (via GDSTS) or by arming the state-change interrupt. Note that state changes triggered by events other than the execution of JSYSes may occur asynchronously with respect to operations performed by the user process.

The following table shows all the defined states and their associated events, actions, and transitions. The "Timeout" event mentioned is a long-term error timeout. Most states also have a short-term timeout for the purpose of retransmitting possibly lost packets; such timeouts never give rise to state changes or other actions.

State	Event	Action	Successor
Closed (0)	OPENF (0,1)	Send RFC	RFCOut
	OPENF (2,3)		Listening
	OPENF (4)		Open
RFCOut (1)	RFC rec'd		Open
	Abort rec'd		Abort
	CLOSF	Send Abort	Closed
Listening (2)	RFC rec'd	Send RFC	Open
	CLOSF		Closed

Open (3)	Timeout		Timeout
	End rec'd		EndIn
	CLOSF	Send End	EndOut
	Abort rec'd		Abort
Timeout (4)	Any activity		Open
	CLOSF	Send Abort	Closed
EndIn (5)	CLOSF	Send EndReply	Dally
	Abort rec'd		Abort
Dally (6)	EndReply rec'd		Closed
	End rec'd	Send EndReply	Dally
	Abort rec'd		Closed
	Timeout		Closed
EndOut (7)	EndReply rec'd	Send EndReply	Closed
	End rec'd	Send EndReply	EndOutDally
	Abort rec'd		Closed
	Timeout	Send Abort	Closed
EndOutDally (10)	EndReply rec'd	Send EndReply	Closed
	End rec'd	Send EndReply	EndOutDally
	Abort rec'd		Closed
	Timeout	Send Abort	Closed
Abort (11)	CLOSF		Closed

Sequential I/O is possible only when the state is "Open" or "End In". Attempts at sequential I/O at other times will result in an I/O Data Error pseudo-interrupt.

Telnet Protocol

Telnet is a higher-level protocol, entirely based upon the Byte Stream Protocol. We discuss it in this document only because server Telnet (network terminal) handling must be performed inside Tenex.

A Telnet connection is ordinarily established when a user process performs a rendezvous to the Telnet server on Maxc, which listens to socket 1 on all networks. After a BSP connection has been opened, it may be "attached" to a network terminal by means of the ATPTY JSYS, described in the JSYS Manual. Thereafter, handling of both byte streams is performed entirely by Tenex.

When the connection is terminated (either normally or abnormally), Tenex destroys the assigned port, releases the network terminal, and detaches the controlled job (if any).

For want of better suggestions, we adopt conventions somewhat like the Arpanet Telnet protocol for control of terminal connections, omitting most

of its provisions (particularly the more complicated ones) for which there is no forseeable need within the PARC environment.

All Telnet control commands consist of a BSP Mark byte (of which there are 256 possible types) followed by zero or more arguments given in data bytes immediately following the Mark.

All bytes that are not part of such a control sequence are treated as ordinary data bytes. Typically, Telnet data will consist only of ASCII characters in the range 0-177 (octal), but we leave the door open to the possibility of 8-bit binary terminal I/O for such purposes as extended character sets or graphics.

The control functions to be implemented initially are as follows:

1) Data Mark (DM, Mark type 1), in conjunction with the BSP Interrupt signal, is used to perform the "Synch" operation. A Telnet sender (user or server) desiring to generate a "Synch" signal inserts a DM in the byte stream and simultaneously transmits an Interrupt. The Interrupt, not being subject to BSP flow control, reaches the receiver even if the byte stream is otherwise clogged up due to allocation or buffering considerations. Upon receipt of the Interrupt, the receiver immediately discards already processed data as appropriate (e.g. clearing terminal buffers), and then reads the byte stream until the DM is encountered. While doing so, the receiver discards all data *except* Telnet control commands and any other "interesting" signals it may choose to act upon.

In Telnet terminal communication, the "Synch" signal is used for two similar purposes. First, it may be generated by the Telnet server when it is called upon to "clear the terminal output buffer". For example, the Tenex CFOBF JSYS, when executed for a network terminal, generates a "Synch" and thereby causes the user's terminal output buffer to be cleared and all data already in the byte stream to be flushed. Second, it may be generated by the Telnet user upon discovering a "terminal input buffer full" condition that is even preventing panic signals (e.g. control-C) from reaching the server because the byte stream is blocked due to lack of allocation.

2) Terminal parameters, such as Line Width (Mark type 2), Page Length (Mark type 3), and Terminal Type (Mark type 4). Each of these is followed by an argument byte specifying the value of the parameter. For Line Width and Page Length, this value is simply the maximum number of characters per line and lines per page, with the value zero defined to mean "unknown" or "inapplicable" (e.g. due to variable width characters). For Terminal Type, the argument is a Tenex terminal type number,[1] which at present may be any one of the following:

---

[1] It is conceded that this aspect of our Telnet protocol is rather strongly oriented toward Tenex conventions. The Arpanet Telnet experience has demonstrated that it is impractical to attempt to do Telnet terminal handling in a "completely general" fashion. We propose simply putting off designing a more general-purpose terminal type specification protocol until such time as we require the added generality.

0	Model 33 TTY (no Tab, Formfeed, or lower case).
1	Model 35 TTY (Tab and Formfeed, no lower case).
2	Model 37 TTY (Tab, Formfeed, lower case).
3	TI (lower case, funny padding, no Tab or Formfeed).
7	NVT (Tab, Formfeed, lower case, no padding whatsoever).
10	LA30, GE Terminet-300 (special padding).
11	TI-733 (slower CR than normal TI).
12	Scope (pause at end of page).
20	PARC NVT (lower case, no Tab or Formfeed, no padding).

Upon receiving a terminal parameter command, the Tenex server Telnet sets the corresponding local parameter for that network terminal.

Arpanet Telnet functions not to be implemented (at least initially) include the following:

1) Network standard representations for common operations, such as EL (Erase Line), IP (Interrupt Process), etc. The justification for this is that within the PARC environment, we do not mind adopting the conventions of whatever server host we happen to be connected to. If users turn out to be unhappy about this, these functions are easy to add.

2) Negotiation about server or user echoing. Since we don't have any half-duplex terminals and always operate over short distances, it seems reasonable to adopt server echoing as the PARC standard.

3) The negotiation mechanism itself. The Arpanet Telnet "option negotiation" protocol is quite involved, and experience has shown it to be very difficult to implement correctly. When we find something to negotiate about, we should perhaps try to design something simpler.

4) More elaborate interactions such as the Telnet Reconnection protocol and the Remote-Controlled Transmission and Echoing protocol (again, due to lack of any forseeable need for these mechanisms).

Backward Compatibility for MCA

It is expected that there will be a transition period during which Maxc will support Pup but Pogos (specifically Squid) will not. For the sake of making this transition as smooth as possible, a module will *temporarily* be included in Tenex supporting the old MCA protocol. This program will intercept non-Pup packets received from the MCA, and will provide a user interface to the MCA compatible with the existing one.

Specifically, the following will be supported:

1) Terminal connections to the MCA, as used by Squid and NEWMCA.

2) Sequential I/O to device MCA:, as used by Minx and the Pogos file server.

The present "Ether-MCA" protocol for the Ethernet will also be supported for as long as is necessary to allow Alto users to obtain the new Pup-based successors to the current EtherMCA and Maxc programs.

This compatibility will be provided in a manner intended to minimize the programming time required. The current MCA driver will be modified to interface with the new Tenex/NVIO communication protocol at the lowest level. No attempt will be made to achieve efficient sharing of code or data structures, since we intend to entirely remove this module as soon as possible.

Implementation

Processing of Pups is performed by Tenex at three major levels. At the lowest level (input and output *interrupt level*), Pups received from NVIO are merely placed on the input queues for the appropriate ports, and Pups to be output are removed from the port output queues and passed to NVIO.

At the second level lies the *byte stream processor*, which performs the work required to transform a queue of raw input Pups into a queue of buffers containing a pure byte stream, and the reverse on output. The byte stream processor is implemented by an independent background Tenex process which runs on demand and which also performs various other tasks such as executing the Rendezvous Protocol and handling I/O for network terminals.

The highest level consists of the *user program interface* and related routines that are sequentially executed in response to JSYS calls. For ports open in byte stream mode, this level communicates with the byte stream processor, whereas for ports open in raw packet mode, communication is directly with the interrupt-level Pup driver.

Pup Encapsulation

Pups are encapsulated in the Ethernet as described in Bob Metcalfe's memo. That is, the Ethernet destination and source hosts are in the first word (required by hardware); the second word contains 1000 (octal) as a signal that this message contains a Pup, and the remainder of the message is the Pup itself.

In the MCA, Pups are preceded by a single word which is 100000 (octal) plus the length of the Pup in words. This convention allows messages containing Pups to be distinguished from messages conforming to the old MCA protocol. Pups may be split across physical messages in the same manner as is done currently for "logical" MCA messages. Refer to IOS-13 for relevant information.

In the Arpanet, Pups are encapsulated in standard IMP-Host protocol data messages on experimental links whose numbers are yet to be assigned. The 32-bit leader is immediately followed by the Pup.

The maximum length Pup that Tenex will send or receive on any network contains 512 data bytes.

Tenex/NVIO Communication

Tenex performs I/O to all networks through NVIO. We propose a protocol for Tenex/NVIO communication that minimizes the amount of development and maintenance that must be done for NVIO. This is for two reasons:

1) NVIO is already difficult to maintain. It is written in Nova assembly language, and its debugging and monitoring facilities are inferior to those available for Tenex development.

2) The Nova is eventually to be replaced by an Alto (in Maxc-II or possibly sooner), and whatever is added now will have to be recoded for the Alto.

This protocol requires no knowledge of Pup format on the part of NVIO. NVIO's role is limited to passing packets between Maxc and the various networks, and providing a moderate amount of buffering. Even forwarding of packets to other hosts (Maxc's gateway function) will be performed by Tenex.

For Pup input, Tenex passes NVIO the address of a maximum-length buffer in Maxc main memory. Upon receipt of a packet from any network, NVIO transfers that packet from its own buffers to the Maxc buffer. It then stores data in an overhead word in the buffer informing Tenex of (1) the number of 16-bit words in the packet, and (2) the physical source address of the packet (network and host). Finally, NVIO generates a Maxc interrupt, and it is done with this packet.

For Pup output, Tenex passes NVIO the address of a buffer containing a packet to be transmitted (with overhead word already set up). If NVIO has sufficient room to buffer the packet, it copies it and notifies Tenex that it is prepared to accept further packets. Later, when NVIO has actually transmitted the packet or otherwise disposed of it, it passes the address of the Maxc buffer back to Tenex and generates a completion interrupt, after setting a field in the overhead word to indicate completion or error. NVIO will discard any packets that it cannot transmit within a short timeout (e.g. due to repeated collisions on the Ethernet or an unresponsive destination host on the MCA).

With the introduction of Pup and packet-at-a-time transfers between Tenex and NVIO, we will take the opportunity to do away with the current word-at-a-time protocol used for Arpanet communication. NVIO can distinguish between Pup, Host-Host Protocol, and other types of packets by looking at the link number of both incoming and outgoing Arpanet messages. In the case of Host-Host Protocol messages, NVIO can also take advantage of the variable byte size capability of the Nova-Imp interface to perform reformatting that is now done at interrupt level by Tenex.

One unresolved question: Should we arrange for Pup storage buffers in Tenex to be shared with non-Pup Arpanet storage, or should we have independent storage regions? The former seems cleaner, but may require

some care to ensure that Pup and Arpanet buffer management strategies are compatible, and we would like to minimize changes to be made in the present Tenex Arpanet driver. If we choose independent storage regions, then we need either to provide a separate communication path between Tenex and NVIO for non-Pup Arpanet messages, or else to block-transfer data between Pup storage and Arpanet storage in Maxc.

Interrupt-Level Pup Processor

Pup input is initially enabled by the Pup background process, which allocates and locks into core a queue of free, maximum-length Pup input buffers, and is responsible for maintaining a sufficient number of input buffers in this queue so that it never becomes empty (unless an excessive number of input Pups have been buffered but not yet taken by user processes). The address of the first buffer is passed to NVIO to be filled with a Pup.

When a Pup has been received and NVIO has triggered the Pup input completion interrupt in Maxc, the following operations are performed:

1) If the packet is not an encapsulated Pup (as determined by the format of the enclosing packet, which is specific to the network on which the packet was received), special routines are invoked to dispose of the packet. These will not be described here.

2) If the Pup Destination is not Maxc, the Pup is passed to the *gateway processor* (to be described later) for immediate forwarding to another network/host.

3) The Destination Socket number (only) is used to compute a hash probe into the local socket table. The correct port is found by comparing the Pup Destination with the local port specification and the Pup Source with the foreign port specification of each port checked, ignoring wildcard elements. If no match is found, the Pup is discarded.

4) If too many unprocessed Pups are already queued for the input port, the Pup is discarded. Otherwise, the buffer is linked to the end of the *input buffer queue* for that port, and is unlocked from core.

5) If the "Pup Received" interrupt is enabled for that port, a PSI request is generated for the owning process.

6) The next free input buffer (if any) is taken from the free input buffer queue and passed to NVIO, and the interrupt is dismissed.

Output Pups generated either from user level or by the byte stream processor are locked into core and linked to the end of the port's *output buffer queue*. If Pup output is idle, an output interrupt is then manually initiated to start things moving.

When the output interrupt routine is called, it first disposes of the buffer just completed (if any), as will be described shortly. It then sequences through the output port table in round-robin fashion until it finds a port with a non-empty output buffer queue. A buffer is removed from the front of the queue, its address is passed to NVIO, and the interrupt is dismissed.

Note that NVIO will generate an output interrupt for either (or both) of two reasons: (a) it has buffered an output packet in Nova memory and is prepared to take more, or (b) it has disposed of an output packet (either transmitted it or declared it untransmittable). In the latter case, NVIO stores a completion or error code in the buffer header and passes the address of the buffer back to Tenex.

Pups generated by the byte stream processor may also be on a second queue called the *BSP output queue* (threaded through a separate cell in the buffer header). When Tenex receives an interrupt signalling that NVIO has finished with such a packet, the interrupt routine does nothing with that buffer, but rather leaves it for disposal or possible retransmission by the byte stream processor. However, if the buffer is not on the BSP output queue, it is immediately unlocked and placed on a queue to be deallocated by the Pup background process. (The interrupt routine cannot perform the deallocation itself because the storage allocation routines may reference nonresident storage).

Pup Background Process

A number of tasks are performed by the Pup background process, which is an asynchronous, independent Tenex process running as a fork under job zero. Included in these tasks are the following:

1) Global storage management, such as allocating and locking free input buffers, deallocating used output buffers, and occasionally garbage-collecting the socket number hash table.

2) Network terminal handling. All network terminals are periodically scanned for service. Input data is retrieved via byte stream processor subroutines and packed into terminal input buffers, and terminal output is unpacked and given to the byte stream processor to transmit.

3) Byte Stream and Rendezvous Protocol processing. Each port open in byte stream mode (including network terminal ports) has associated with it sufficient state to permit the (single) Pup background process to scan through all ports and perform any needed services for each one. Hence we may describe these operations as if there were a separate process handling each byte stream.

Input processing is performed whenever the port's input buffer queue is found to be non-empty. The byte stream processor dispatches on the Pup Type field of each Pup, disposes of it as appropriate, and continues until the input buffer queue is empty.

When Data (and AData) Pups are processed, they are placed on a queue called the *BSP input queue*, ordered by the value of the Pup ID. Duplicate packets are discarded during this step. The BSP input queue is divided into two segments: a "complete" segment at the front and an "incomplete" segment at the back. The "complete" portion constitutes an uninterrupted byte stream, which may immediately be read by a JSYS call or transferred to terminal input buffers. The "incomplete" portion (which, by definition, begins at the first gap in the byte stream) consists of all later packets that may have been received out of sequence.

The byte stream processor maintains a pointer to the boundary between the "complete" and "incomplete" portions of the BSP input queue, and updates it as gaps are filled in by newly-arriving data. It also maintains a "left window edge" which is the byte ID of the first packet in the queue, and a "right window edge" which is equal to the left window edge plus a constant determining the maximum number of bytes that may be buffered for a single port. The byte stream processor will discard received data packets whose Pup IDs lie outside this window.

After processing all packets in the input buffer queue, the byte stream processor decides whether or not to send an acknowledgment. If an AData was encountered, an acknowledgment is always sent; otherwise, the decision is a function of (a) the number of bytes in the "complete" portion of the BSP input queue, and (b) the time elapsed since the last acknowledgment was sent.

The acknowledgment packet is generated as follows: The Pup ID is set equal to the byte sequence number corresponding to the first gap (i.e. the beginning of the "incomplete" segment of the BSP input queue), hence cumulatively acknowledging all the "complete" data. The Number of Bytes field is set to the difference between that and the right window edge. All Pups in the "incomplete" portion of the queue are then specifically acknowledged in the PosAck/NegAck Blocks field of the Ack packet.

Output is handled in a similar fashion. Data generated by the outputting process is delivered to the byte stream processor as a queue of Pups which we call the *BSP output queue*. This queue is threaded through a different cell in the buffer header than is used for the output buffer queue referenced at interrupt level.

The byte stream processor also maintains a "window" of Pup IDs that may be transmitted, based on information in received Ack packets.

To transmit a packet, the byte stream processor stamps it with the current time and places it in the output buffer queue, but *without* removing it from the BSP output queue. Each time the byte stream processor services an outputting port, it (re)queues for output all packets lying within the window that have (a) not been transmitted previously, (b) timed out, or (c) had negative acknowledgments received for them.

When an Ack packet is received, all packets being positively acknowledged (either cumulatively or specifically) are removed from the BSP output queue, and also from the output buffer queue if they are already queued for retransmission. Buffers for which negative acknowledgments are received are marked for immediate retransmission. The window parameters are then updated with the new information in the Ack, possibly permitting the user process to generate further output.

Received Pups other than Data, AData, and Ack are passed to a subroutine implementing the finite state machine whose transition table was given in a previous section of this memo.

JSYS Call Processor

The JSYSes for the most part are very straightforward and may be implemented by an uncomplicated mapping from the user program interface specifications given earlier.

The PUPI and PUPO operations for raw packet I/O simply remove packets from the input buffer queue and append packets to the output buffer queue, respectively. No action by the Pup background process is required.

For sequential input and output, packing and unpacking of the data is performed at JSYS level. The sequential input routine sets up a byte pointer to the first buffer in the BSP input queue and a byte count equal to the number of bytes in the Pup. The data thereafter is removed by the device-independent handlers for BIN, SIN, etc. When the packet is exhausted, the byte stream processor is awoken to do any necessary servicing (such as updating the window parameters and sending a new Ack), and the sequential input routine immediately proceeds to the next packet in the BSP input queue. If the "complete" portion of the queue becomes exhausted, the process blocks until it again becomes non-empty.

The sequential output routine allocates a buffer capable of holding a maximum-length Pup, then simply sets up the appropriate byte pointer and count. Actual packing of the data is done by the device-independent BOUT and SOUT handlers. When the buffer is full or must be forced out (by CLOSF, MTOPR 21, etc.), it is placed on the end of the BSP output queue and the byte stream processor is awoken. The process then blocks if the current byte sequence number is beyond the right window edge by more than a certain amount.

It might be advantageous (for reasons of reduced process switching overhead and possibly increased bandwidth) to allow some of the byte stream processor code to be executed at JSYS level at those points where we specified "waking up the byte stream processor" in the above descriptions. This requires fairly strict interlocking, however, to prevent anomalies due to race conditions. It is probably sufficient to provide one lock per port, serving as a global interlock for BSP-related processing for that port.

Gateway Processor

In order to minimize the overhead required to perform forwarding of packets from one network to another, the gateway processor is immediately called at input interrupt level upon receipt of a packet whose destination is not Maxc. Forwarding is accomplished by the following sequence of operations:

1) The hop count in the Transport Control field is incremented, and the Pup is discarded if it reaches the value 8.

2) If the Pup Checksum is nonzero, it is updated to account for the Transport Control field's having been changed. This change is made incrementally, so the updated checksum will be correct only if the original checksum was correct. The gateway processor neither generates nor checks Pup Checksums.

3) The Destination Network is used to index into a routing table that determines the network/host address to which the packet is to be forwarded. Special entries in the table denote networks to which Maxc is connected directly, and other networks which are known to be unreachable. In the former case, the Pup is transmitted directly to the network/host address given in the Pup header, while in the latter case the Pup is discarded.

4) The Pup is linked to the end of a special output queue that is periodically checked by the output interrupt routine.

It is anticipated that the routing table will be fixed in the initial implementation. In later implementations, it might be periodically updated by the Pup background process on the basis of dynamic determination of network load and/or connectivity.

Remaining Topics

The following subjects are among those that remain to be considered in future documents:

1) Structure and contents of network directory (for name/address translation). Protocol for making the directory lookup function available as a service for other hosts.

2) Standard "NCP Port" (by means of which host operating systems may communicate information such as host status, which is not related to individual user ports).

3) Inquiry service to be implemented by gateway hosts (see the memo "Naming and Addressing Conventions for Pup").

4) File Transfer Protocol (FTP).

Z80 Cissem.

More Z80 ASM NOTES 1/6/82
cos Duvall

① option flags: /H = Don't allow Hex numbers not prefixed by Numeric
/S = Don't turn off display
/T = Line by Line listing (for debugging only)
/Z = Suppress 0 at end of strings

② Macro Facility

DEFINE Macroname ARGLIST = macro body

Macroname ::= symbol

ARGLIST ::= empty | ARG ARGLISTTAIL
ARGLISTTAIL ::= empty | , ARG ARGLISTTAIL
ARG ::= SYMBOL | delim STRING delim

Macrobody ::= Text ARGREF macrobody | empty

ARGREF ::= empty | { ARG PARTIALOP }

PARTIALOP ::= empty | :D:D

E.G. DEFINE LSHIFT REG = {SLA {REG|1:1}
RL {REG|0:1}}

Then: LSHIFT DE expands to:

```
SLA  E
RL   D
```

A Few Notes on the 8080 compatible version of z80asm.

W. S. Duvall

Dec 31, 1981

In order to run the z80 assembler in the 8080 compatibility mode (which allows you to assemble 8080 assembler files with minimal modification), you need to retrieve the files "[IVY]<DUVALL>Z80ASM>8080OPS" and "[IVY]<DUVALL>Z80ASM>ASM8080". Copy 8080ops to the file named "z80ops" on your disk, and run "z80asm/b", which will produce an initialized boot file "Z80.BOOT" for the 8080 mnemonics. You can rename this file to be "8080.boot" or something if you wish to keep both a z80 and 8080 assembler around.

To Prepare Your Source File:

(1) Place the statement "GET ASM8080" as the first line in the 8080 assembler source file.

(2) Comment out any ORG statements (these will be done in the load control file)

(3) The following constructs are illegal, and must be removed:

3a. IF
3b. ENDIF
3c. SET
3d. $

(4) Reserved words may not be re-used as program symbols.

(5) Hexidecimal numbers must end with the letter "H", but neet NOT begin with a number. This means that "FAH" is a legitimate number.

(6) The string delimiter is double quote.

The string escape character is ", which may be Combined with:

"": literal "
"': literal '
"↑: literal ↑
"N: Return (0DH)
"L: Linefeed (0AH)
"T: TAB (09H)
"B: Backspace (08H)
"NNN: A character with the octal value NNN

The character '↑' produces a CR LF sequence.

(7) Character constants are preceded by a single quote, e.g. '

Inter-Office Memorandum

To Distributed Computing

Date 15 January 1975

From Bob Metcalfe

Location Coyote Hill

Subject Pup Format

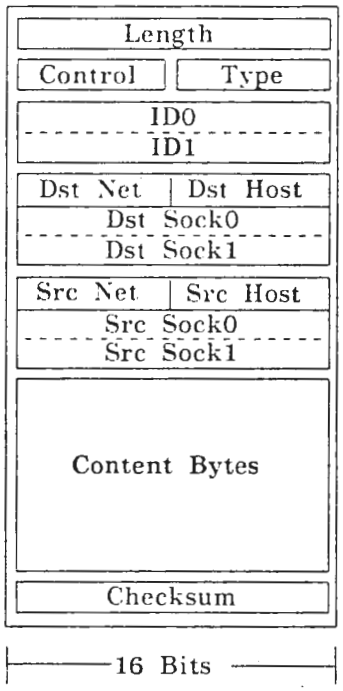
Organization Parc CSL

XEROX

DRAFT
COMMENTS PLEASE

Here is an update on the format of a Pup. The only change since my memo of July 16, 1974 is that the Pup Identifier is now 32 bits, up from 16. Pup overhead is now 22 bytes per packet, including the Pup Checksum. See Ed Taft's memo of January 9, 1975 for elaboration on Pup address fields. At present, we are intending that the Pup Checksum (if non-zero) be a 1's complement add-and-cycle over the entire Pup. I have started a registry of Pup types in <metcalfe>pupsyms.bc. A Pup's Length should be between 22 and 534 bytes.

Pup Format



```
; ASM8080 -- DEFS FILE FOR ASSEMBLING 8080 CODE

; MUST USE 8080OPS AS OPCODE FILE

DEFINE PSW = AF #
DEFINE M = @HL #
DEFINE DB = BYTE #
DEFINE DW = XWORD #
DEFINE DS = BLOCK #
DEFINE EQU = = #
DEFINE END = #
```

```
; 8080 COMPATIBLE MNEMONICS -- USING FILE MUST GET "ASM8080"
; W. S. DUVALL Dec 31, 1981 9:02 AM


OPCODE MOV


  ; Store Register 8-bit

        A B C D E H L
    @HL: 77 70 71 72 73 74 75

  ; Load Register 8-bit

    A B C D E H L @HL

    A: 7F 78 79 7A 7B 7C 7D 7E
    B: 47 40 41 42 43 44 45 46
    C: 4F 48 49 4A 4B 4C 4D 4E
    D: 57 50 51 52 53 54 55 56
    E: 5F 58 59 5A 5B 5C 5D 5E
    H: 67 60 61 62 63 64 65 66
    L: 6F 68 69 6A 6B 6C 6D 6E

OPCODE LDAX

  B: 02
  D: 12

OPCODE STAX

  B: 0A
  D: 1A

OPCODE LHLD

  m: 2A[1.R][1.L]


OPCODE SHLD

  m: 22[1.R][1.L]


OPCODE LDA

  m: 3A[1.R][1.L]


OPCODE STA

  m: 32[1.R][1.L]

OPCODE MVI
  ; Load Register 8-bit Immediate

          m
    A: 3E[1.R]
    B: 06[1.R]
    C: 0E[1.R]
    D: 16[1.R]
    E: 1E[1.R]
    H: 26[1.R]
    L: 2E[1.R]
    @HL: 36[1.R]

OPCODE LXI
  ; Load Register 16-bit Immediate

          m
    B: 01[1.R][1.L]
    D: 11[1.R][1.L]
    H: 21[1.R][1.L]
    SP: 31[1.R][1.L]

OPCODE PUSH ; Push operand: (SP) ← REG PAIR, (SP) ← (SP) + 2
```

```
  @HL: 9E

OPCODE SBI;
  m: DE[1.R];

OPCODE ANA; A = A &

  A: A7
  B: A0
  C: A1
  D: A2
  E: A3
  H: A4
  L: A5
  @HL: A6

OPCODE ANI;
  m: E6[1.R];

OPCODE XRA; A = A Xor

  A: AF
  B: A9
  C: A9
  D: AA
  E: AB
  H: AC
  L: AD
  @HL: AE

OPCODE XRI;
  m: EE[1.R];

OPCODE ORA; A = A or

  A: B7
  B: B0
  C: B1
  D: B2
  E: B3
  H: B4
  L: B5
  @HL: B6

OPCODE ORI;
  m: F6[1.R];

OPCODE CMP; A - op, no load (set flags)

  A: BF
  B: B8
  C: B9
  D: BA
  E: BB
  H: BC
  L: BD
  @HL: BE

OPCODE CPI;
  m: FE[1.R];

OPCODE DAA 27; DECIMAL ADJUST ACCUMULATOR
OPCODE CMA 2F; COMPLEMENT ACCUMULATOR
OPCODE STC 37; SET CARRY
OPCODE CMC 3F; COMPLEMENT CARRY

OPCODE INR; REG = REG + 1

  A: 3C
  B: 04
  C: 0C
  D: 14
  E: 1C
  H: 24
  L: 2C
  @HL: 34
```

```
OPCODE INX; REG = REG + 1
  ; 16 BIT GROUP

  B: 03
  D: 13
  H: 23
  SP: 33


OPCODE DCR; REG = REG + 1

  A: 3D
  B: 05
  C: 0D
  D: 15
  E: 1D
  H: 25
  L: 2D
  @HL: 35

OPCODE DCX; 16 BIT GROUP

  B: 0B
  D: 1B
  H: 2B
  SP: 3B

OPCODE RLC 07; Rotate Left Circular

OPCODE RRC 0F; Rotate Right Circular

OPCODE RAL 17; Rotate Left (includes carry)

OPCODE RAR 1F; Rotate Right (includes carry)

; Jumps

  ; Jumps to Absolute Addresses

    OPCODE JMP; Jump Unconditional

      m: C3[1.R][1.L]

    OPCODE JC
      m: DA[1.R][1.L]; Jump on Carry
    OPCODE JNC
      m: D2[1.R][1.L]; Jump on not Carry
    OPCODE JZ
      m: CA[1.R][1.L]; Jump on Zero
    OPCODE JNZ
      m: C2[1.R][1.L]; Jump on not Zero
    OPCODE JPE
      m: EA[1.R][1.L]; Jump on Parity Even
    OPCODE JPO
      m: E2[1.R][1.L]; Jump on Parity odd
    OPCODE JM
      m: FA[1.R][1.L]; Jump on Neg
    OPCODE JP
      m: F2[1.R][1.L]; Jump on Pos

  ; Call to subroutine push pc + 3 on stack

    OPCODE CALL
      m: CD[1.R][1.L]; Unconditional
    OPCODE CC
      m: DC[1.R][1.L]; Call on Carry
    OPCODE CNC
      m: D4[1.R][1.L]; Call on not Carry
    OPCODE CZ
      m: CC[1.R][1.L]; Call on Zero
    OPCODE CNZ
      m: C4[1.R][1.L]; Call on not Zero
    OPCODE CPE
      m: EC[1.R][1.L]; Call on Parity Even
    OPCODE CPO
```

```
      m: E4[1.R][1.L]; Call on Parity odd
    OPCODE CM
      m: FC[1.R][1.L]; Call on Neg
    OPCODE CP
      m: F4[1.R][1.L]; Call on Pos

  ; Return from subroutine

    OPCODE RET C9; return from subroutine
    OPCODE RC D8; return on Carry
    OPCODE RNC D0; return on not Carry
    OPCODE RZ C8; return on Zero
    OPCODE RNZ C0; return on not Zero
    OPCODE RPE E8; return on Parity Even
    OPCODE RPO E0; return on Parity odd
    OPCODE RM F8; return on Neg
    OPCODE RP F0; return on positive

; Input/Output

  OPCODE IN ; input from IO port

    m: DB[1.R]; Input HL = memaddr, B = byte count, A←char

  OPCODE OUT ; Output to IO port

    m: D3[1.R]; Output HL = memaddr, B = byte count, A = byte


OPCODE NOP 00
OPCODE HLT 76 ; Wait for interrupt
OPCODE DI F3 ; Disable interrupts
OPCODE EI FB ; Disable interrupts
```

```
OPCODE = MOV
  MOV @HL, A     ;77   Lit = 77
  MOV @HL, B     ;70   Lit = 70
  MOV @HL, C     ;71   Lit = 71
  MOV @HL, D     ;72   Lit = 72
  MOV @HL, E     ;73   Lit = 73
  MOV @HL, H     ;74   Lit = 74
  MOV @HL, L     ;75   Lit = 75
  MOV L, A      ;6F   Lit = 6F
  MOV L, B      ;68   Lit = 68
  MOV L, C      ;69   Lit = 69
  MOV L, D      ;6A   Lit = 6A
  MOV L, E      ;6B   Lit = 6B
  MOV L, H      ;6C   Lit = 6C
  MOV L, L      ;6D   Lit = 6D
  MOV L, @HL     ;6E   Lit = 6E
  MOV H, A      ;67   Lit = 67
  MOV H, B      ;60   Lit = 60
  MOV H, C      ;61   Lit = 61
  MOV H, D      ;62   Lit = 62
  MOV H, E      ;63   Lit = 63
  MOV H, H      ;64   Lit = 64
  MOV H, L      ;65   Lit = 65
  MOV H, @HL     ;66   Lit = 66
  MOV E, A      ;5F   Lit = 5F
  MOV E, B      ;58   Lit = 58
  MOV E, C      ;59   Lit = 59
  MOV E, D      ;5A   Lit = 5A
  MOV E, E      ;5B   Lit = 5B
  MOV E, H      ;5C   Lit = 5C
  MOV E, L      ;5D   Lit = 5D
  MOV E, @HL     ;5E   Lit = 5E
  MOV D, A      ;57   Lit = 57
  MOV D, B      ;50   Lit = 50
  MOV D, C      ;51   Lit = 51
  MOV D, D      ;52   Lit = 52
  MOV D, E      ;53   Lit = 53
  MOV D, H      ;54   Lit = 54
  MOV D, L      ;55   Lit = 55
  MOV D, @HL     ;56   Lit = 56
  MOV C, A      ;4F   Lit = 4F
  MOV C, B      ;48   Lit = 48
  MOV C, C      ;49   Lit = 49
  MOV C, D      ;4A   Lit = 4A
  MOV C, E      ;4B   Lit = 4B
  MOV C, H      ;4C   Lit = 4C
  MOV C, L      ;4D   Lit = 4D
  MOV C, @HL     ;4E   Lit = 4E
  MOV B, A      ;47   Lit = 47
  MOV B, B      ;40   Lit = 40
  MOV B, C      ;41   Lit = 41
  MOV B, D      ;42   Lit = 42
  MOV B, E      ;43   Lit = 43
  MOV B, H      ;44   Lit = 44
  MOV B, L      ;45   Lit = 45
  MOV B, @HL     ;46   Lit = 46
  MOV A, A      ;7F   Lit = 7F
  MOV A, B      ;78   Lit = 78
  MOV A, C      ;79   Lit = 79
  MOV A, D      ;7A   Lit = 7A
  MOV A, E      ;7B   Lit = 7B
  MOV A, H      ;7C   Lit = 7C
  MOV A, L      ;7D   Lit = 7D
  MOV A, @HL     ;7E   Lit = 7E

OPCODE = LDAX
  LDAX D      ;12   Lit = 12
  LDAX B      ;02   Lit = 02

OPCODE = STAX
  STAX D      ;1A   Lit = 1A
  STAX B      ;0A   Lit = 0A

OPCODE = LHLD
```

```
  LHLD m      ; 2Axx   Lit = 2A, Field = 1.R, Field = 1.L

OPCODE = SHLD
  SHLD m      ; 22xx   Lit = 22, Field = 1.R, Field = 1.L

OPCODE = LDA
  LDA m       ; 3Axx   Lit = 3A, Field = 1.R, Field = 1.L

OPCODE = STA
  STA m       ; 32xx   Lit = 32, Field = 1.R, Field = 1.L

OPCODE = MVI
  MVI @HL, m    ; 36x   Lit = 36, Field = 1.R
  MVI L, m      ; 2Ex   Lit = 2E, Field = 1.R
  MVI H, m      ; 26x   Lit = 26, Field = 1.R
  MVI E, m      ; 1Ex   Lit = 1E, Field = 1.R
  MVI D, m      ; 16x   Lit = 16, Field = 1.R
  MVI C, m      ; 0Ex   Lit = 0E, Field = 1.R
  MVI B, m      ; 06x   Lit = 06, Field = 1.R
  MVI A, m      ; 3Ex   Lit = 3E, Field = 1.R

OPCODE = LXI
  LXI SP, m     ; 31xx   Lit = 31, Field = 1.R, Field = 1.L
  LXI H, m      ; 21xx   Lit = 21, Field = 1.R, Field = 1.L
  LXI D, m      ; 11xx   Lit = 11, Field = 1.R, Field = 1.L
  LXI B, m      ; 01xx   Lit = 01, Field = 1.R, Field = 1.L

OPCODE = PUSH
  PUSH AF     ; F5   Lit = F5
  PUSH H      ; E5   Lit = E5
  PUSH D      ; D5   Lit = D5
  PUSH B      ; C5   Lit = C5

OPCODE = POP
  POP AF      ; F1   Lit = F1
  POP H       ; E1   Lit = E1
  POP D       ; D1   Lit = D1
  POP B       ; C1   Lit = C1

OPCODE = XCHG
  XCHG        ; EB   Lit = EB

OPCODE = XTHL
  XTHL        ; E3   Lit = E3

OPCODE = PCHL
  PCHL        ; E9   Lit = E9

OPCODE = SPHL
  SPHL        ; F9   Lit = F9

OPCODE = ADD
  ADD @HL     ; 86   Lit = 86
  ADD L       ; 85   Lit = 85
  ADD H       ; 84   Lit = 84
  ADD E       ; 83   Lit = 83
  ADD D       ; 82   Lit = 82
  ADD C       ; 81   Lit = 81
  ADD B       ; 80   Lit = 80
  ADD A       ; 87   Lit = 87

OPCODE = DAD
  DAD SP      ; 39   Lit = 39
  DAD H       ; 29   Lit = 29
  DAD D       ; 19   Lit = 19
  DAD B       ; 09   Lit = 09

OPCODE = ADI
  ADI m       ; C6x   Lit = C6, Field = 1.R

OPCODE = ADC
  ADC @HL     ; 8E   Lit = 8E
  ADC L       ; 8D   Lit = 8D
  ADC H       ; 8C   Lit = 8C
  ADC E       ; 8B   Lit = 8B
  ADC D       ; 8A   Lit = 8A
```

```
  ADC C      ; 89   Lit = 89
  ADC B      ; 88   Lit = 88
  ADC A      ; 8F   Lit = 8F

OPCODE = ACI
  ACI m      ; CEx   Lit = CE, Field = 1.R

OPCODE = SUB
  SUB @HL     ; 96   Lit = 96
  SUB L      ; 95   Lit = 95
  SUB H      ; 94   Lit = 94
  SUB E      ; 93   Lit = 93
  SUB D      ; 92   Lit = 92
  SUB C      ; 91   Lit = 91
  SUB B      ; 90   Lit = 90
  SUB A      ; 97   Lit = 97

OPCODE = SUI
  SUI m      ; D6x   Lit = D6, Field = 1.R

OPCODE = SBB
  SBB @HL     ; 9E   Lit = 9E
  SBB L      ; 9D   Lit = 9D
  SBB H      ; 9C   Lit = 9C
  SBB E      ; 9B   Lit = 9B
  SBB D      ; 9A   Lit = 9A
  SBB C      ; 99   Lit = 99
  SBB B      ; 98   Lit = 98
  SBB A      ; 9F   Lit = 9F

OPCODE = SBI
  SBI m      ; DEx   Lit = DE, Field = 1.R

OPCODE = ANA
  ANA @HL     ; A6   Lit = A6
  ANA L      ; A5   Lit = A5
  ANA H      ; A4   Lit = A4
  ANA E      ; A3   Lit = A3
  ANA D      ; A2   Lit = A2
  ANA C      ; A1   Lit = A1
  ANA B      ; A0   Lit = A0
  ANA A      ; A7   Lit = A7

OPCODE = ANI
  ANI m      ; E6x   Lit = E6, Field = 1.R

OPCODE = XRA
  XRA @HL     ; AE   Lit = AE
  XRA L      ; AD   Lit = AD
  XRA H      ; AC   Lit = AC
  XRA E      ; AB   Lit = AB
  XRA D      ; AA   Lit = AA
  XRA C      ; A9   Lit = A9
  XRA B      ; A9   Lit = A9
  XRA A      ; AF   Lit = AF

OPCODE = XRI
  XRI m      ; EEx   Lit = EE, Field = 1.R

OPCODE = ORA
  ORA @HL     ; B6   Lit = B6
  ORA L      ; B5   Lit = B5
  ORA H      ; B4   Lit = B4
  ORA E      ; B3   Lit = B3
  ORA D      ; B2   Lit = B2
  ORA C      ; B1   Lit = B1
  ORA B      ; B0   Lit = B0
  ORA A      ; B7   Lit = B7

OPCODE = ORI
  ORI m      ; F6x   Lit = F6, Field = 1.R

OPCODE = CMP
  CMP @HL     ; BE   Lit = BE
  CMP L      ; BD   Lit = BD
  CMP H      ; BC   Lit = BC
```

```
  CMP E      ;BB   Lit = BB
  CMP D      ;BA   Lit = BA
  CMP C      ;B9   Lit = B9
  CMP B      ;B8   Lit = B8
  CMP A      ;BF   Lit = BF

OPCODE = CPI
  CPI m      ;FEx   Lit = FE, Field = 1.R

OPCODE = DAA
  DAA        ;27   Lit = 27

OPCODE = CMA
  CMA        ;2F   Lit = 2F

OPCODE = STC
  STC        ;37   Lit = 37

OPCODE = CMC
  CMC        ;3F   Lit = 3F

OPCODE = INR
  INR @HL     ;34   Lit = 34
  INR L      ;2C   Lit = 2C
  INR H      ;24   Lit = 24
  INR E      ;1C   Lit = 1C
  INR D      ;14   Lit = 14
  INR C      ;0C   Lit = 0C
  INR B      ;04   Lit = 04
  INR A      ;3C   Lit = 3C

OPCODE = INX
  INX SP     ;33   Lit = 33
  INX H      ;23   Lit = 23
  INX D      ;13   Lit = 13
  INX B      ;03   Lit = 03

OPCODE = DCR
  DCR @HL     ;35   Lit = 35
  DCR L      ;2D   Lit = 2D
  DCR H      ;25   Lit = 25
  DCR E      ;1D   Lit = 1D
  DCR D      ;15   Lit = 15
  DCR C      ;0D   Lit = 0D
  DCR B      ;05   Lit = 05
  DCR A      ;3D   Lit = 3D

OPCODE = DCX
  DCX SP     ;3B   Lit = 3B
  DCX H      ;2B   Lit = 2B
  DCX D      ;1B   Lit = 1B
  DCX B      ;0B   Lit = 0B

OPCODE = RLC
  RLC        ;07   Lit = 07

OPCODE = RRC
  RRC        ;0F   Lit = 0F

OPCODE = RAL
  RAL        ;17   Lit = 17

OPCODE = RAR
  RAR        ;1F   Lit = 1F

OPCODE = JMP
  JMP m      ;C3xx   Lit = C3, Field = 1.R, Field = 1.L

OPCODE = JC
  JC m       ;DAxx   Lit = DA, Field = 1.R, Field = 1.L

OPCODE = JNC
  JNC m      ;D2xx   Lit = D2, Field = 1.R, Field = 1.L

OPCODE = JZ
  JZ m       ;CAxx   Lit = CA, Field = 1.R, Field = 1.L
```

```
OPCODE = JNZ
  JNZ m       ; C2xx   Lit = C2, Field = 1.R, Field = 1.L

OPCODE = JPE
  JPE m       ; EAxx   Lit = EA, Field = 1.R, Field = 1.L

OPCODE = JPO
  JPO m       ; E2xx   Lit = E2, Field = 1.R, Field = 1.L

OPCODE = JM
  JM m        ; FAxx   Lit = FA, Field = 1.R, Field = 1.L

OPCODE = JP
  JP m        ; F2xx   Lit = F2, Field = 1.R, Field = 1.L

OPCODE = CALL
  CALL m      ; CDxx   Lit = CD, Field = 1.R, Field = 1.L

OPCODE = CC
  CC m        ; DCxx   Lit = DC, Field = 1.R, Field = 1.L

OPCODE = CNC
  CNC m       ; D4xx   Lit = D4, Field = 1.R, Field = 1.L

OPCODE = CZ
  CZ m        ; CCxx   Lit = CC, Field = 1.R, Field = 1.L

OPCODE = CNZ
  CNZ m       ; C4xx   Lit = C4, Field = 1.R, Field = 1.L

OPCODE = CPE
  CPE m       ; ECxx   Lit = EC, Field = 1.R, Field = 1.L

OPCODE = CPO
  CPO m       ; E4xx   Lit = E4, Field = 1.R, Field = 1.L

OPCODE = CM
  CM m        ; FCxx   Lit = FC, Field = 1.R, Field = 1.L

OPCODE = CP
  CP m        ; F4xx   Lit = F4, Field = 1.R, Field = 1.L

OPCODE = RET
  RET         ; C9   Lit = C9

OPCODE = RC
  RC          ; D8   Lit = D8

OPCODE = RNC
  RNC         ; D0   Lit = D0

OPCODE = RZ
  RZ          ; C8   Lit = C8

OPCODE = RNZ
  RNZ         ; C0   Lit = C0

OPCODE = RPE
  RPE         ; E8   Lit = E8

OPCODE = RPO
  RPO         ; E0   Lit = E0

OPCODE = RM
  RM          ; F8   Lit = F8

OPCODE = RP
  RP          ; F0   Lit = F0

OPCODE = IN
  IN m        ; DBx   Lit = DB, Field = 1.R

OPCODE = OUT
  OUT m       ; D3x   Lit = D3, Field = 1.R
```

```
OPCODE = NOP

OPCODE = HLT
  HLT       ;76   Lit = 76

OPCODE = DI
  DI        ;F3   Lit = F3

OPCODE = EI
  EI        ;FB   Lit = FB
```