

Inter-Office Memorandum

To	Communication Protocols	Date	June 30, 1978
From	Ed Taft and Bob Metcalfe	Location	Palo Alto
Subject	Pup Specifications	Organization	PARC/CSL

XEROX

Filed on: [Maxcl]<Pup>PupSpec.press

The purpose of this memo is to specify a standard with which to interconnect Parc's various packet-switching networks.

This memo updates *Pup Specifications* of October 20, 1975. A number of corrections and editorial changes have been incorporated. The primary motivation for this update has been to produce a version of the specifications in Bravo and Press format.

We have designed 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. However, nearly all resources presently accessible via the Xerox research network communicate using Pups and Pup-based protocols, and, in particular, the Gateway hosts that interconnect our networks will forward Pups only.

At the most basic level, this standard introduces three fundamental principles.

1. All Pup-based communication makes use of a standard inter-network packet format (the *Pup*), specified in this document. All transporting networks are assumed to be able to carry Pups transparently.
2. Pup communication is end-to-end at the packet level. The inter-network is required only to be able to transport independently-addressed Pups from source to destination. Use of higher levels of protocol is entirely the responsibility of the communicating end processes.
3. When Pups are handed to the packet transport mechanism, they may be expected to arrive only with high probability and not necessarily in the order they were transmitted. It is the responsibility of the communicating end processes to agree upon higher-level protocols that provide reliable communication of the quality they themselves require.

Also included in this memo are:

- how a Pup should be encapsulated for transport through various existing networks;
- a Pup Echo Protocol for test and diagnosis;
- a Rendezvous/Termination Protocol for connection establishment and synchronization; and
- a Pup Byte Stream Protocol for carrying error- and flow-controlled byte streams between

cooperating processes.

It should be noted that all of these protocols have remained essentially unchanged since 1975. Experience has demonstrated the strengths and weaknesses of these protocols, and a number of other higher-level Pup-based protocols have been designed and implemented. However, this bulk of all traffic is still carried by the protocols described in this document, and these protocols are unlikely to undergo any further change.

A word of caution is in order to anyone contemplating a new implementation of the Pup protocols. While these specifications are reasonably complete in describing the syntax and semantics of the protocols, they contain very little discussion of strategies for implementing them. Carefully-chosen strategies turn out to be crucial to good performance. Unfortunately, very little documentation on implementation strategies presently exists.

All numbers are decimal unless followed by 'B'.

The Pup

The standard Pup format is illustrated in Figure 1.

The *Pup Length* is the number of 8-bit bytes in the Pup, including header, contents, and checksum. There may be from 0 to 532 content bytes in a Pup so that the length will range between 22 and 554 bytes. The Pup specification does not expressly prohibit Pups longer than 554 bytes, but by convention hosts are not expected to be able to transport Pups larger than this. 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. To date, no such tracing mechanisms have been implemented. Bits 0-3 are used to count the number of Gateways encountered during the Pup's transport. A Pup which reaches its 16th 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 1 through 127 are registered types, such as the ones defined in this document. They may receive optional type-specific handling by intermediate agents such as Gateways. Registered types are assigned a single interpretation that is either used in only one protocol or that is applicable in all protocols; they may not have different interpretations in different protocols.

Types of 128 through 255 are unregistered and their interpretation is strictly a matter of agreement between the source and destination processes. A given type may be assigned independent meanings in different protocols so long as such protocols need not be *compatible* (in the sense of being used simultaneously between a given pair of ports).

In practice, the distinction between registered and unregistered Pup Types has not turned out to be particularly useful, except in one case. Pups of type *Error* may be generated by the packet transport mechanisms without knowledge of the higher-level protocols being used by the end processes.

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. The specific interpretation of the Pup ID is not defined at the Pup level but is rather a matter of convention established at higher levels of protocol (e.g., the Byte Stream Protocol presented later in this document).

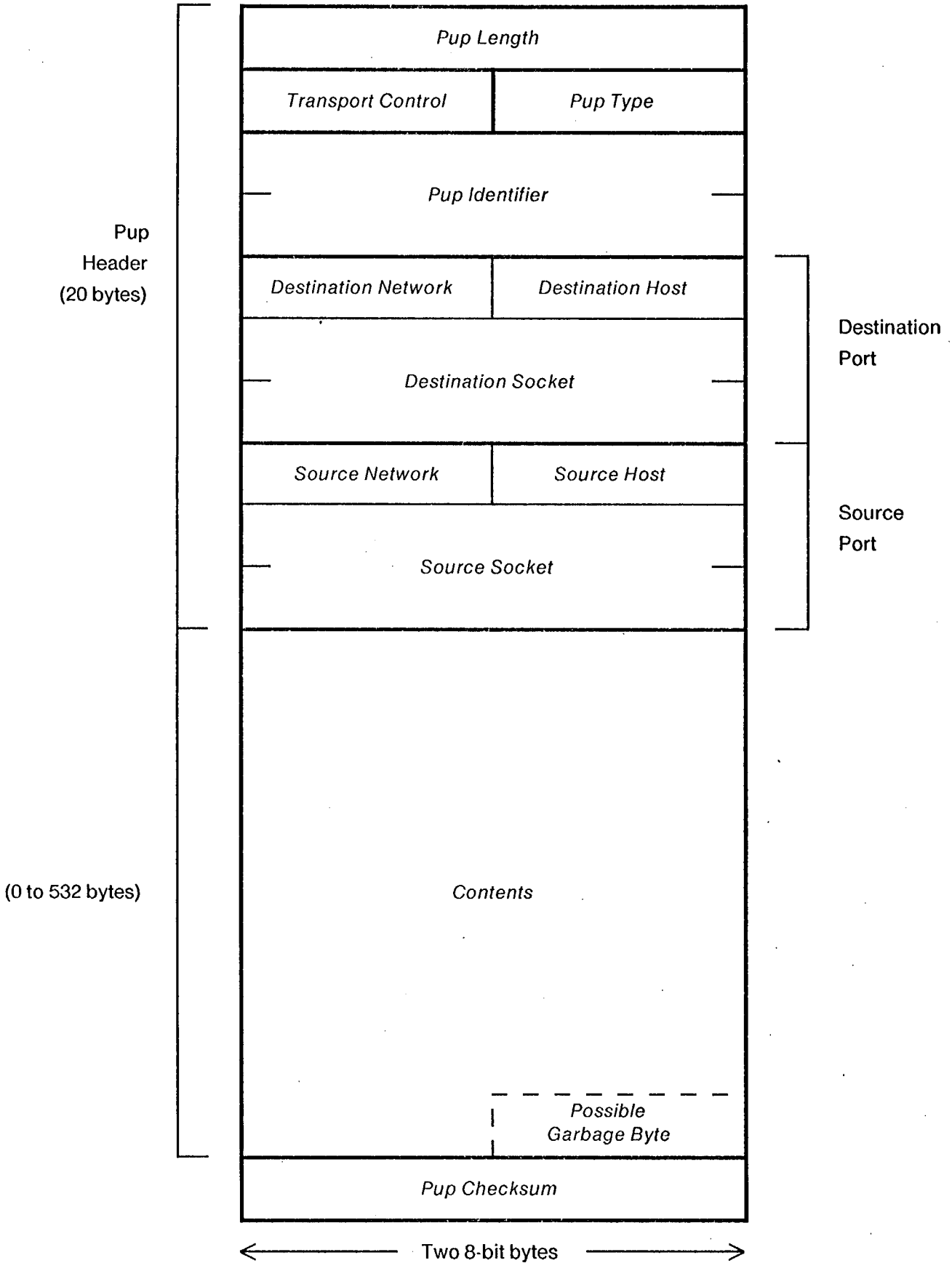


Figure 1. The Format of a Pup

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 $2^{32}-1$ sockets in that host.

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

A source network of zero indicates that the source process does not know the identity of the network to which its host is connected. A destination network of zero indicates that the Pup is addressed to a host in the current network, i.e., the one over which the Pup is physically transmitted by the source process. This convention 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 zero, the process intends the Pup to be *broadcast* in the destination network (if such a broadcast capability is implemented in that network). The Pup is received by active ports at the given destination socket in all hosts on the specified network. By convention, we do not presently permit a Pup source to broadcast to any except a directly connected network; that is, broadcasts never propagate through 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 the memo *Naming and Addressing Conventions for Pup* for an elaboration on these topics.

There may be from 0 to 532 *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.

The *Pup Checksum* is an optional 16-bit, one's complement, add-and-cycle checksum computed over the 16-bit words in the Pup's header and contents. It is intended as an end-to-end reassurance of correct transport by intermediate hardware and software components, and is not associated with or a replacement for any network's existing error checking mechanisms (which are usually specialized to detect the specific sorts of errors commonly encountered on that network).

The checksum is initialized to 0 and computed by repeated one'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. If the result is the ones-complement value "minus zero" (177777B), it should be converted to "plus zero". 177777B is specifically defined to mean that the Pup carries no checksum.

The Pup's checksum is carried with it from source to destination. If and when a Pup is altered enroute, say when the 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. The algorithm for updating the checksum after changing a single word of the Pup is as follows (one's complement arithmetic used throughout):

1. If the Pup Checksum is 177777B, do nothing.
2. Subtract the old contents of the changed word from the new.
3. Left-cycle this difference ($n \bmod 16$) bits, where n is the distance (in words) from the changed word to the Pup Checksum word.

4. Add the result to the existing Pup Checksum.

The foregoing procedure produces a correct Pup Checksum if and only if the original Pup Checksum was correct.

It may 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 sometimes be lost* (even when they are "known" to traverse only networks that are believed to be "perfect"). 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 or other resource limitation at any of the places through which it may pass. An optional Pup Error Protocol exists by means of which a Pup's source may be notified of the packet's demise, but no process should depend on receiving such negative acknowledgments for all (or indeed any) lost packets. 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.

Pup Encapsulation

Pups are to be *encapsulated* to conform to the conventions and 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 previously.

Pup encapsulation consists of two steps (sometimes accomplished in a single operation). First, the Pup is transformed in whatever fashion is necessary to permit the entire Pup (including Header and Checksum) to pass through the transporting network as "data". This generally involves simply adding network-dependent headers and/or trailers, but could also include encoding of data in various ways (for transmission over phone lines, for example). Second, an *immediate destination host* is derived from the Pup Destination Port. This will be either the *final* destination host (if that host is directly connected to the network over which the Pup is about to be transported) or a *Gateway* host through which it is believed (by some sort of routing function) the final destination may be reached.

When a Pup is received at its final destination port, it is *decapsulated* (by applying the inverse of the encapsulation transformation for that network) before being passed to the destination process. When a Pup is received by a Gateway, it is (1) decapsulated, (2) routed to another network, and (3) re-encapsulated according to the conventions of this new network.

Ethernet Encapsulation

Refer to Figure 2. The *Destination* byte (*Immediate Ethernet Destination Host*) 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 *Source* byte (*Immediate Ethernet Source Host*) is the hardware address of the host transmitting the encapsulated Pup through the Ethernet.

The *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 Type of a Pup is 512. * (1000 0100)

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.

IP = 513 (1001 0010)

<i>Destination</i>	<i>Source</i>
<i>Type = "Pup"</i>	
<i>Ethernet CRC</i>	

Ethernet Encapsulation

Encapsulated Pup

<i>Type</i>	<i>Host</i>
<i>Link</i>	<i>0</i>

Arpanet Encapsulation

Figure 2. Pup Encapsulation

Arpanet Encapsulation

At present, Pups are encapsulated within Arpanet packets using the "old" Imp-Host leader format. The *Type* byte is zero, to denote that the packet is an Arpanet "Regular message".

The *Host* byte (*Immediate Arpanet Destination Host*) is derived from the Pup Destination Port as described previously. Note that when the message arrives at its immediate destination, this byte will have been changed by the Imps to identify the message's sender.

The *Link* identifies the message as being a Pup. It should be 152.

BiSync Encapsulation

Pups are encapsulated for transmission on low-speed synchronous lines using a data frame format that is a subset of the BiSync protocol. Synchronous line drivers also implement a network-dependent (non-Pup) line control protocol that enables maintenance of sub-network connectivity and routing information. That protocol is not described in this document.

Echo Protocol

For test and diagnosis purposes, a process receiving an EchoMe Pup may echo it (Figure 3). 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 exchanged, the Gateway hop count zeroed, and the type changed; thus, the checksum must be recomputed. If the receiving process accepts broadcast Pups, it (or the host's Pup handling software) must substitute the local host's actual address for the (zero) Pup Destination Host.

The process controlling 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. Well-known socket 5 is presently assigned to Echo servers.

Rendezvous/Termination 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.

The Rendezvous/Termination Protocol is a convention by means of which a *connection* between two ports may be established and later broken. The manner in which the processes communicate over an existing connection is the subject of other protocols (for example, the Byte Stream Protocol, to be described later).

Rendezvous

A Rendezvous is accomplished with an exchange of packets, each called an *RFC*, a *Request For Connection* (Figure 4).

RFCs may be exchanged to establish a connection between a user process and a server process. The ports between which the 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 *Connection Port* through which the RFC's source intends to maintain the newly

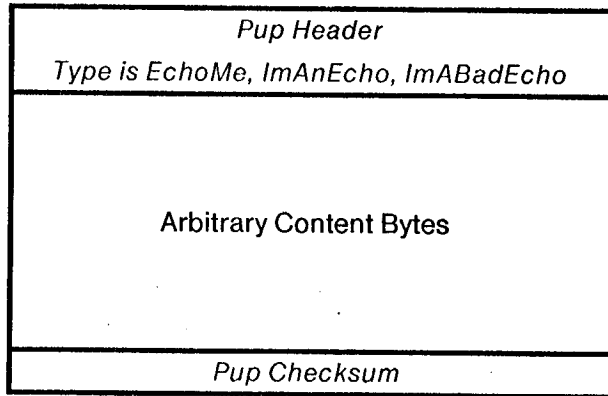


Figure 3. Echo Protocol Pup Format

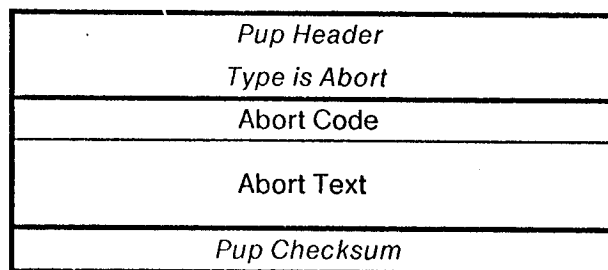
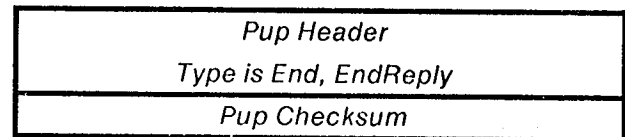
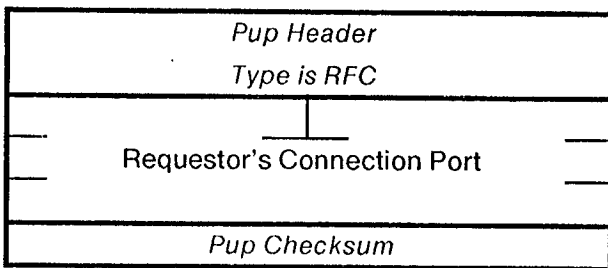


Figure 4. Rendezvous/Termination Protocol Pup Formats

established connection.

The Connection Port is separately specified so that, for example, a server can handle requests for service at a single, widely advertised rendezvous port and provide the service concurrently to a number of users via a number of connection ports. The connection port may be the same as the rendezvous port, and we expect this will be the case for most user processes and for servers not capable of spawning multiple instances of themselves.

The Pup ID of the initiating RFC also defines the *Connection ID* for the resulting connection. It should be chosen in such a way as 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 are 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 between the same pair of ports may be made vanishingly small.

If an initiator's RFC is lost, it should 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. Upon detecting a duplicate RFC, the receiver must, of course, retransmit the appropriate answering RFC before discarding the duplicate. At worst, multiple servers will be generated to which no packets are ever sent; these servers should eventually time out and destroy themselves.

Normal Termination

A connection is normally terminated by a three-way handshake consisting of an *End Pup* and two *EndReply Pups*. The end of a connection may be initiated from either of its ports by transmission of an *End Pup* whose ID matches the *Connection ID*. The *End Pup* must be retransmitted until a matching *EndReply Pup* is received. Upon receiving an *EndReply*, the initiator of the *End* should then send an *EndReply* in response and promptly self destruct.

The receiver of the *End Pup* responds by returning an *EndReply Pup* with matching ID and then *dallying* up to some reasonably long timeout interval (say, 10 seconds) in order to respond to a retransmitted *End Pup* should its initial *EndReply* be lost. If and when the dallying end of the stream connection receives its *EndReply*, it may immediately self destruct.

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 *End* initiator's final *EndReply* is lost), 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.

It may happen that both processes choose to send *End Pups* simultaneously. Upon receiving an *End Pup* in seeming answer to an *End Pup* of its own, a port should at once send an *EndReply* and begin dallying in the normal fashion (i.e., it should abandon sending *Ends*).

Abnormal Termination

The *Abort Pup* should be used to terminate a connection (or a connection attempt) in the event of a detected catastrophe. 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. 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

(by timeout).

The Abort Pup carries a program interpretable code and a human readable explanation of some abnormal condition. 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 itself 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.

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 may be accomplished using a bidirectional byte stream maintained using the Byte Stream Protocol (BSP). A byte-stream connection between two ports is generally established and destroyed by means of the Rendezvous/Termination protocol described previously.

Byte Stream Maintenance

Bytes in a stream are numbered consecutively by a 32-bit number referred to as the Byte ID, which is initialized to the Connection ID (i.e., the Pup ID used for the rendezvous) when the connection is created. A byte stream is carried from one port to another by *Data* Pups, each containing zero to 532 consecutive bytes starting with the one identified by the Pup's ID. In return for these are transmitted Acknowledgment packets with matching identifiers (though not necessarily on a one-to-one basis) 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 for them at the stream receiver. The sender is informed about receiver allocations in the *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.

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.

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.

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. A received Ack whose Pup ID is less than the previous one should be considered a delayed duplicate and discarded. Also, it carries a 3-word allocation block indicating the receiver's state as of the time the Ack was sent. The stream sender should update its state to reflect the cumulative acknowledgment denoted by the Pup ID (e.g., discard Data packets being held for possible retransmission) before considering the updated allocations.

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

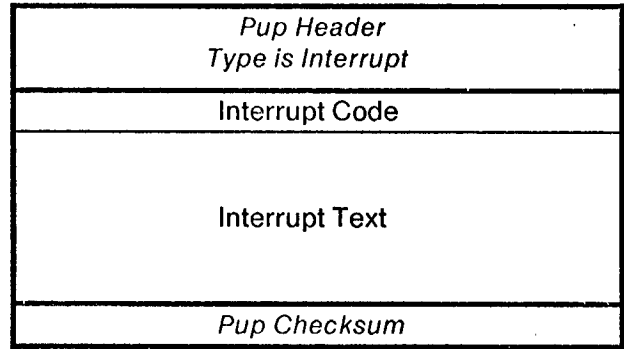
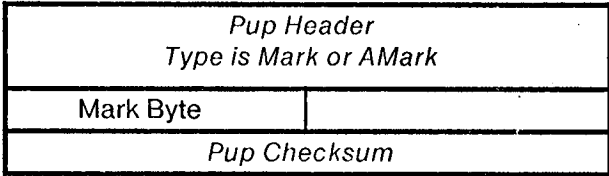
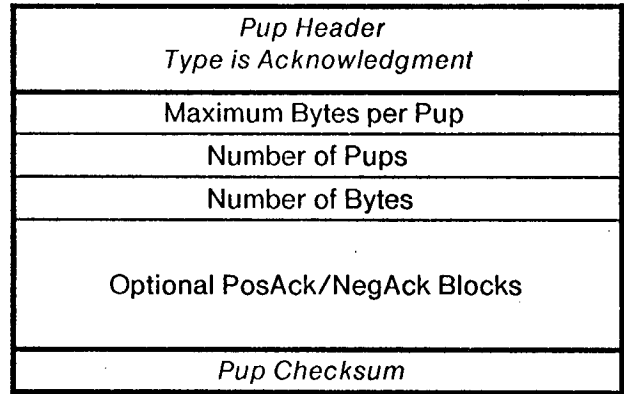
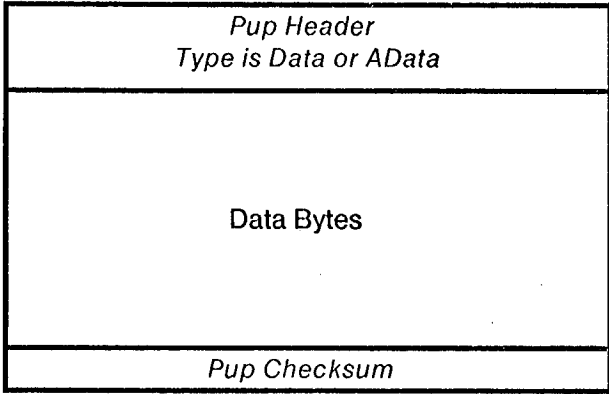


Figure 5. Byte Stream Protocol Pup Formats

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.

The "Number of Pups" allocation indicates the maximum number of additional Data Pups the stream receiver is prepared to handle, over and above any Data Pups it has already received and may be holding in its buffers. In making use of this allocation, the stream sender should assume that any Data Pups that it has transmitted but which have not been acknowledged have in fact not (yet) reached the receiver. Hence, the "Number of Pups" allocation should be compared to the number of unacknowledged output Data packets in order to determine whether or not it is OK to generate additional Data packets.

The "Number of Bytes" allocation should be interpreted relative to the Ack's ID, i.e., the byte ID of the first byte yet to be acknowledged. Adding the allocation to the Pup ID of the Ack in which it came yields the ID of the last byte in the stream which the receiver is prepared to accept. This byte ID implied in Acks should be monotonically increasing so that stream senders need not hold back existing data Pups they had previously committed to transmit. In general, this means that a stream receiver should not decrease the total amount of storage allocated for buffering received Data Pups during the life of a connection. However, strict adherence to this policy may be difficult to implement; hence, small, short-term decreases in allocation should be tolerated by stream senders. Similarly, the maximum number of bytes per Pup should not be decreased during the life of a BSP connection, so that stream senders need not take existing data Pups and break them apart.

Optionally, an Ack Pup may carry up to 85 *specific acknowledgments*, described by *Pos/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 retransmission 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.

Each 3-word Pos/NegAck Block begins with a bit indicating whether the interval is known to be missing or received: a one 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. To simplify processing of Pos/NegAck Blocks by the stream sender, the intervals denoted by successive blocks in the same Ack should start at monotonically increasing Byte IDs and should not overlap.

The transfer of bytes from stream sender to stream receiver should not depend on these Pos/NegAck Blocks being used by either end, except that bytes might flow with less efficiency without them. A receiver may choose not to include Pos/NegAck Blocks in its Ack Pups and a sender may choose to ignore them if present.

Experience has shown that when communicating over high-bandwidth, low-loss networks such as the Ethernet, the software overhead required to generate and interpret specific acknowledgments is not rewarded by any noticeable improvement in performance. No software presently implements specific acknowledgments.

Marks and Interrupts

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 exactly one content byte which indicates which of a possible 256 types of mark is being signalled.

While reading the data from a stream, a process 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.

For purposes of transmission and flow control, Marks are treated exactly the same as Data Pups. They occupy one position in the Byte ID sequence, and are acknowledged in the same manner as any other byte in the stream. An *AMark* Pup is simply a Mark that demands an immediate acknowledgment (in the same manner as an *AData*).

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. 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. Interrupt Pup IDs are generated from the stream's send Interrupt ID, which is initially the Connection ID. Successive interrupts advance the Interrupt ID. An Interrupt should be retransmitted until acknowledged.

Upon receipt of an Interrupt Pup, it should be acknowledged with an *InterruptReply* Pup only if its ID is equal to or one less than the current Interrupt ID. If its ID matches the current Interrupt ID, the using process should be signalled and the Interrupt ID advanced. If its ID is one less than the current Interrupt ID, it is a duplicate and should therefore be acknowledged without giving rise to a new signal to the process.

Interaction with Rendezvous/Termination Protocol

The Byte Stream Protocol interacts with the Rendezvous/Termination Protocol in two important ways. First, the Connection ID (i.e., the Pup ID of the initiating RFC) is used to initialize the stream's Byte IDs and Interrupt IDs for both directions, as has already been explained.

Second, to ensure clean and unambiguous termination of the byte stream, it is required that a stream sender delay transmitting either an End or an EndReply until all outstanding Data and Interrupt Pups have been acknowledged. For the same reason, it is forbidden to transmit (or even *retransmit*) Data Pups once an End or EndReply has been sent.

In the typical scenario for terminating a BSP connection, one of the processes decides that the connection should be closed. It first waits until no unacknowledged Data or Interrupt packets remain, then transmits an End.

At the other end of the connection, the process eventually exhausts the incoming byte stream and is notified that an End has been received, thereby terminating the stream. It now proceeds in turn to transmit any remaining data, wait until no unacknowledged Data or Interrupt packets remain, transmits an End Reply, and begins dallying.

In the meantime, the first process (the one that originally requested termination) reads the remainder of the incoming byte stream. When the stream is exhausted and the End Reply has been received, the process is notified of termination of the incoming stream. It now sends the answering EndReply (to terminate the other, dallying process) and is free to destroy the port.

Registered Pup Types

The Pup Types for the protocols described in this document are assigned the following values:

<i>Type</i>	<i>Assignment</i>
EchoMe	1
ImAnEcho	2
ImABadEcho	3
Error	4
RFC	8
Abort	9
End	10
EndReply	11
Data	16
ADData	17
Ack	18

Mark	19
Interrupt	20
InterruptReply	21
AMark	22