Inter-Office Memorandum

To	Communication Protocols	Date	June 30, 1978
From	Ed Taft	Location	Palo Alto
Subject	Implementation of Pup in Tenex	Organization	PARC/CSL

XEROX

Filed on: [Maxc1]<Pup>Tenex-Pup.press

This is another revision of a memo with the same title dated October 18, 1975. The purpose of this edition is to bring the documentation up-to-date (only minor changes have been made) and to re-issue it in Bravo and Press formats.

Bibliography

Pup Specifications
Describes the Pup philosophy, basic packet format, and second-level Pup-based protocols (Echo, Rendezvous/Termination, BSP). File <Pup>PupSpec.press.

Naming and Addressing Conventions for Pup
File <Pup>PupName.press.

The Pup Network Directory and the PUPNM JSYS
Describe conventions and facilities for naming and addressing Pup ports. File <Pup>PupDirectory.press.

Pup Connection State Diagram
Presents a model for implementation of the Rendezvous/Termination Protocol. File <Pup>RTPStates.press.

Pup Telnet Protocol
Describes a BSP-based protocol for performing character-oriented communication with dumb terminals. File <Pup>Telnet.press.

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* (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.

Copyright Xerox Corporation 1979

Network Filenames

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

PUP: <local port> <socket type> . <foreign port>

where all fields except the device name "PUP:" are optional.

The *local* and *foreign port* specifications are as described in the memos *Naming and Addressing Conventions for Pup* and *The Pup Network Directory and the PUPNM JSYS*. They each evaluate to an 8-bit *network number*, an 8-bit *host number*, and a 32-bit *socket number* (or a list of such addresses in the case of the foreign port). 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 *socket type* attribute (if present), determines the actual local socket number to be used, in the following manner:

1. If the local port specification is followed by "!A" (for "Absolute"), the 32-bit socket number is completely determined by the socket number field of the local port specification. The use of "!A" 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 local port specification is followed by "!U", the high-order 17 bits are set equal to the process's connected directory number, hence making a user-relative socket. "!U" is the default if no socket type is specified.

3. If the local port specification is followed by "!J", the high-order 17 bits are set equal to the process's job number plus 100000 (decimal), hence making a job-relative socket.

4. 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 one or more <network, host, socket> triples 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 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, Tenex first 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.

A network file may be opened in either of two principal modes: *raw packet* mode (16 octal) and *byte stream* mode (0 through 4).

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, which transfer one complete Pup per command. 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. 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.

Source address filtering and Pup Checksum generation and checking are optional. The user program is responsible for these operations if the options are not requested.

The calling sequence for PUPI and PUPO is as follows:

Accepts in	1:	B0:	Never dismiss for I/O, give PUPX3 error instead.
		B1:	Generate (PUPO) or check (PUPI) Pup Checksum, give PUPX5 error if incorrect.
		B2:	(PUPI only) Perform source address check, give PUPX7 error if incorrect.
		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 (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 remainder of the Pup will be lost. Then, if B1 of 1 is set, the Pup Checksum will be checked and the error return taken if it is incorrect (PUPX5, "PUPI: Checksum incorrect"). If B2 of 1 is set, the Source Port will be checked against the foreign port specification for this network file, and the error return taken if it doesn't match (PUPX7, "PUPI: Source address incorrect").

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. 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, if possible. 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 is inconsistent with the local port specification or designates a <network, host> pair that does not refer to Maxc, or either the Source Port or Destination Port contains any elements which are still zero. (Exception: some networks permit *broadcasting* a packet to all hosts on that network by specifying a zero destination host.)

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 non-nil Pup Checksum. A program supplying a non-nil Pup Checksum must therefore specify fully the Source Port and Destination Port (thereby preventing any substitution by Tenex), and use these in computing the checksum. Alternatively, B1 of 1 may be set in the call to PUPO, causing Tenex to generate the Pup Checksum.

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.

Byte Stream Mode

Byte stream mode operates in accordance with the Byte Stream Protocol (BSP), described in *Pup Specifications*.

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 and other JSYSes 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. Both JFNs must be opened by the same job, but may be opened by different forks of that job. 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 dissociate the two JFNs from the port (the connection is actually closed by the CLOSF on the JFN opened for writing).

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 will ordinarily 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), and if the foreign port specification evaluates to a list of addresses, one of them is chosen at this time by Tenex. 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 completely unspecified.

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). Ordinarily a maximum-length Pup contains 532 data bytes, but the Byte Stream Protocol permits a receiver to specify a smaller maximum Pup length.

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 case of a Mark, the value of the Mark byte may be read by MTOPR 23, and 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*, and the port is destroyed after all JFNs referencing it have been closed.

If at any time an Abort is received, the connection will be terminated abnormally. If the user program was 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" or IOX5,

"I/O data error"). 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*, 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 within the inactivity timeout interval (default 2 minutes), 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. (The setting of *Timeout* also causes a state change interrupt if armed.)

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

Operations initiated by OPENF and CLOSF are also subject to being timed out. If the port remains in the same (non-*Closed*) state for more than the error timeout interval, an Abort will be sent and the error return will be taken (OPNX20, "Connection attempt timed out" or IOX5, "I/O data error"). Ports in the *Listening* state are not timed out.

The timeout interval may be supplied in B10-17 of AC2 during the OPENF. The unit of time is 4.096 seconds and the default value is 32 (approximately 2 minutes).

Miscellaneous JSYSes

A new JSYS called PUPNM has been added to perform conversion between Pup names and addresses. This is described in a separate memo *The Pup Network Directory and the PUPNM JSYS*.

CVSKT is extended to deal with Pup ports. It takes a Pup JFN in 1 and returns 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 output to be acknowledged; that function is performed by DOBE (see below).

MTOPR function 22 causes an Interrupt packet to be generated. The Interrupt Code should be supplied in AC3, and AC4 (if nonzero) is used to supply the Interrupt Text.

MTOPR function 23 returns the value of the most recently received Mark byte right-justified in AC3.

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. All armed interrupts are directed to the fork that most recently executed this MTOPR.

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 connection to be terminated abnormally and the port placed in the *Closed* state. The contents of AC3 are used as the Abort Code and AC4 (if nonzero) 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 (if nonzero) 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). If 3 is nonzero in the call, it is used as an address table pointer (in the same manner as the PUPNM JSYS) to return the foreign port address(es). That is, LH 3 is the length and RH 3 the address of a table in the user address space into which GDSTS stores the foreign address(es). Two words are used for each address: the first contains the network number in the left half and the host number in the right, and the second contains the socket number right-justified. The number of words actually stored is then returned in LH 3.

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 following is the format of the port status word. Most bits are undefined for a port not open in BSP mode. Bits marked '*' are for internal use only and should not be relied upon (they may change without notice).

*B0 Port is locked.

*B1 BSP processing request pending.

*B2 BSP input available.

*B3 BSP output possible.

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 2 minutes, or protocol interaction failed to complete within 2 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.

B8 Port is open for reading.

B9 Port is open for writing.

B10 Packets are to be discarded at random intervals by Tenex (this facility is

provided for the purpose of testing lost message detection and flow control algorithms, and is implemented for both Raw Packet and BSP ports). May be set or cleared by SDSTS.

*B11 Need to send Acknowledgment.

*B12 Received Acknowledgment.

*B13 Last Acknowledgment sent gave zero allocation.

*B14 Interrupt outstanding.

*B15 Port is or was listening.

*B16 BSP output queue non-empty.

*B17 BSP not possible in this state.

B18-31 Currently unused.

B32-35 Port state (see memo *Pup Connection State Diagram*).

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. For SOBE, the count does not include bytes contained in the packet currently being assembled (if any). DOBE first forces transmission of any partially-filled Pup, then dismisses until all data has been cumulatively acknowledged. Note that completion of DOBE does not imply that the receiving process has actually seen the data, merely that it has been delivered successfully to the destination host.

The privileged OPRFN JSYS is extended to include a few Pup-related operations, encoded by means of a SIXBIT name in AC1:

PUPBGF Enables printing of internally detected Pup errors on the logging console if AC2 is nonzero, and disables printing if zero.

PUPROU Sets the Pup routing table entry for network (AC2)-1 to the value in AC4 masked by AC3 (i.e., only the bits corresponding to ones in AC3 are changed). The routing table may be read from the PUPROU GETAB table.

PUPDIR Maps in the highest version of <SYSTEM>PUP-NETWORK.DIRECTORY.

Values of Pup Symbols

JSYS numbers:

PUPI	JSYS 441
PUPO	JSYS 442
PUPNM	JSYS 443

Error codes:

PUPX1	602010	PUPI/O: Block size error
PUPX2	602011	Pup address error
PUPX3	602012	PUPI/O: Operation not possible now
PUPX4	602013	(Not presently used)
PUPX5	602014	PUPI: Checksum incorrect
PUPX6	602015	PUPI/O: JFN not open in mode 16
PUPX7	602016	PUPI: Source address incorrect
PUPX8	602017	PUPI/O: JFN doesn't refer to device PUP:
PUPNX1	602030	PUPNM: Name or address not found
PUPNX2	602031	PUPNM: Name ambiguous
PUPNX3	602032	PUPNM: Syntax error or illegal address
PUPNX4	602033	PUPNM: Inconsistent values in name expression
ATPX14	602040	ATPTY: JFNs don't refer to same device
ATPX15	602041	ATPTY: Pup JFNs don't refer to same local port
ATPX16	602042	ATPTY: Pup JFNs don't refer to BSP port
ATPX17	602043	ATPTY: Pup connection not open

Port state codes:

S.CLOS	0	Closed
S.RFCO	1	RFC Out
S.LIST	2	Listening
S.OPEN	3	Open
S.ENDI	4	End In
S.ENDO	5	End Out
S.DALY	6	Dally
S.ABOR	7	Abort

PUP device type: 402

Pup-Related GETAB Tables

The first four tables are parallel and are indexed by a "Pup port index".

PUPLSK — Local socket numbers, right-justified. Zero denotes a free port and -1 a deleted port.

PUPLNH — Local network/host numbers and BSP linkage. The network and host numbers of the local port address are in B0-7 and B8-15 respectively (zero means wildcard). The right half points to the BSP data block, if any (see below).

PUPFPT — Foreign port address(es). If zero, the foreign port is completely wildcard. If nonzero, the LH is the negative of the number of words in the foreign port address table and the RH is a pointer to a block containing the address table (the address table begins in the second word of the block). The address table format is the same as for the PUPNM and GDSTS JSYSes.

PUPSTS — Port status words (as returned by GDSTS).

NVTPUP — Data for Pup network terminals (NVTs), indexed by (TTY # minus the TTY # of the first Pup NVT). Zero denotes a free NVT. An assigned NVT has the sign bit set and the Pup port index of the associated connection in the RH. The rest of the word is for internal use only.

PUPPAR — Pup parameter table, for miscellaneous constants. Indexed by item number, as follows:

0 - # of Pup NVT's ,, TTY # of first Pup NVT.
1 Monitor address of Pup free storage region.
2 B0=1 if this Maxc system is a Gateway.

PUPBUF Pup free storage region, in which foreign port address tables and BSP data blocks are allocated. This GETAB table encompasses the entire region. Its monitor address is given by the second word of the PUPPAR table. Hence, to examine the blocks that the PUPLNH and PUPFPT tables point to (these pointers being monitor addresses), one should subtract the starting address from the pointer to yield an index into PUPBUF. Consult PUP.MAC for the format of the BSP data block (which is subject to change).

PUPROU Pup routing and address table, indexed by network number minus 1. B0 is set if the network is known to be inaccessible. B1 is set if it is possible to send "broadcast" packets on that net. B2-9 and B10-17 give, respectively, the network and host numbers of a gateway to which packets for the given network are to be routed by Tenex (zero means that packets are to be routed directly to their destinations). The RH gives the local host address on that network (zero means that Maxc is not connected to the network).

PUPSTA Pup statistics table (subject to change). At present, the only statistics recorded are a breakdown of where the Pup background process spends its time. Indexed by item number, as follows:

0 Count of calls to release used output buffers.
1 Count of calls to assign new input buffers.
2 Count of calls to garbage-collect port table.
3 Count of calls to do BSP background processing.
4 Count of NVT input/output scans.
5 Count of Telnet protocol timeout checks.
6 Time spent releasing used output buffers.
7 Time spent assigning new input buffers.
10 Time spent garbage-collecting port table.
11 Time spent in BSP background processing.
12 Time spent in NVT input/output scans.
13 Time spent in Telnet protocol timeout checks.
14 Total time spent by the Pup background fork.

PUPBGT Hash table in which all PUPBUGs are recorded (whether or not they are also printed on the logging terminal). Each word in the table is either zero (unused) or contains the PC of the call to BUG(PUP,<...>) in RH and a count of occurrences in LH. The PUPBUG subsystem prints out the contents of this table.

Telnet Protocol

Telnet is a higher-level protocol, entirely based upon the Byte Stream Protocol. We mention 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 on socket 1. This socket is maintained by the PUPSRV program. 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 byte streams in both directions 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).

The Telnet protocol itself is described in the memo *Pup Telnet Protocol*. We describe here only those aspects of it that are peculiar to Tenex.

When a Synch signal (Interrupt and Data Mark) is received, it causes the attached terminal input buffer to be cleared in addition to its normal effect of flushing the incoming byte stream.

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 outgoing byte stream to be flushed.

The DOBE JSYS, when executed for a network terminal, invokes the Telnet Timing Mark protocol so as to wait until all previous terminal output has been processed by the Telnet user.

Upon receiving a terminal parameter command, the Tenex server Telnet sets the corresponding local parameter for that network terminal. The default terminal type is NVT (7), and the default line width and page length are 72 and 66 (decimal), respectively.

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.

Tenex/NVIO Communication

Tenex performs I/O to all networks through NVIO (or AltIO on Maxc2). NVIO's role is limited to passing packets between Maxc and the various networks, and providing a moderate amount of buffering. It has proven necessary for NVIO to have some knowledge of Pup format so as to distinguish between Pups and old-protocol MCA and Ethernet packets and to permit filtering of obviously bad input packets at the lowest level. However, no logical processing is performed on Pups by NVIO. Even forwarding of packets to other hosts (Maxc's gateway function) is 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 the number of Maxc words in the packet itself and 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 has taken the packet. 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 have taken the opportunity to do away with the previous word-at-a-time protocol used for Arpanet communication. The word-at-a-time protocol was originally implemented in an attempt to emulate the BBN PDP-10/Imp interface so as to minimize changes to the Tenex Imp driver. In retrospect, this goal proved not to be worthwhile. 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 was formerly done at interrupt level by Tenex.

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. The address fields in the Pup header are checked for reasonableness. Zero network numbers are defaulted to their proper values (see the memo *Naming and Addressing Conventions for Pup*).

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 of each port checked, ignoring wildcard elements. If no match is found, the Pup is discarded. (Source address filtering is not performed at interrupt level.)

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. If the port is a BSP port, the Byte Stream processor is awoken. Actually, for ports controlled by JFNs (as opposed to NVTs), the awakening of the Byte Stream processor is delayed for 100 milliseconds so as to give the user fork an opportunity to do the BSP processing itself, assuming it is blocked for I/O over that connection. This reduces process-switching overhead.

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 a single *output buffer queue* (the number of Pups and the amount of storage occupied by them are, however, maintained on a per-port basis). 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 output (if any), as will be described shortly. A buffer is then removed from the front of the output queue, its address is passed to NVIO, and the interrupt is dismissed.

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).

Routing of each Pup to the correct immediate destination network and host is accomplished by means of a routing table, which contains an entry for every accessible network. Directly-connected networks have their entries assembled in. Entries for other networks are initially empty (inaccessible) and are maintained by the PUPSRV program on the basis of its interactions with other Gateways using the Gateway Information Protocol.

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 sends an Acknowledgment if an AData was encountered. 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. The current version of the software generates no PosAck/NegAck blocks in outgoing Acknowledgments, and ignores PosAck/NegAck blocks in incoming Acknowledgments.

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.

The last retransmitted packet on the BSP output queue is send as an AData rather than a Data Pup if (a) the oldest unacknowledged packet on the queue is older than the "hold time" (currently 0.5 second), or (b) allocation has fallen to less than 25% of its value in the most recently received Ack.

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 Pup 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 described in the memo *Pup Connection State Diagram*.

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 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 number of Pups or bytes on the BSP output queue exceeds the maximum permitted for one port.

Byte stream processing is performed at JSYS level where possible so as to reduce the frequency of wakeups of the Pup background process. To prevent anomalies due to race conditions, each port has a lock associated with it, serving as a global interlock for BSP-related processing for that port.

Gateway Processor

Maxc may optionally be a Pup Gateway between the networks to which it is connected. This is controlled by the GATEWF assembly switch. At present, Maxc1 is a Gateway and Maxc2 is not.

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 non-nil, 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 Pup is routed and linked to the end of the Pup output queue in the same manner as is done for normal Pup output.