Mesa Pup Package Functional Specification

Version 6.0
October, 1980

This document describes the Mesa Pup Package and other facilities available for communication using the Pup Protocols.

XEROX
SYSTEMS DEVELOPMENT DEPARTMENT
3333 Coyote Hill Road / Palo Alto / California 94304

c Xerox Corporation 1979, 1980

Introduction

This document describes the Mesa Pup Package. The FTP Package is described in the *FTP Functional Specification*. The Stream Interface is described in appendix A.

If you are looking for some simple examples, the Mesa PupTest package is probably a good place to start. Look at PtBSP.mesa and/or PtEcho.mesa.

Currently, November 20, 1979, the Pup Package has the following performance characteristics. These numbers were obtained using FatPup with the default parameters and a very simple test program so there is no disk activity or other significant interference. Even the display and the keyboard/mouse interrupts were turned off. Things would, of course, go faster without the Statistics option.

Checksums	On	Off	Off-Local
Socket:	150kb	614kb	423kb
PktStream:	117kb	365kb	248kb
ByteStream:	106kb	331kb	226kb
Echo long packets:	38kb/110ms	222kb/19ms	
Echo short packets:	21ms	18ms	

The Pup Package consists of a set of modules built up in a layered structure, where each layer adds more service to the structure of the previous layers. The client program may interface to the package at any layer, trading service against speed and space for its particular application.

The layers are:

Ethernet Driver (not directly accessible)
PupRouter (not directly accessible)
Socket
PacketStream
ByteStream

This document will describe the various interfaces to the Pup Package. First comes a section on describing how to turn the Pup Package on. Then the ByteStream interface will be described, followed by the Socket interface. (The PacketStream interface isn't described because of lack of interest. Contact me if you are interested.) Next comes the description of the name utilities, the packet utilities, and the **SIGNAL**s and **ERROR**s generated by the Pup Package. Finally, there are sections describing the EFTP Package and the Stats Package.

Terminology

There are a lot of buzzwords in the network business. Here is my attempt to explain the important ones.

Connections

The Ethernet hardware delivers packets from one machine to another with high probability. Unfortunately, packets will be lost occasionally, so protocols are needed to transfer data reliably. When two processes on two machines agree to communicate with each other, they establish a *connection*. (Actually, they can be two portions of the same process, or two processes on the same machine.) A connection has a *local* end and a *remote* end.

Streams

Once a connection is opened, a *stream* of data flows across it. Actually, since the Pup Package supports full *duplex connections*, each connection has a *send* stream, and a *receive* stream. Note that this is not the normal Mesa meaning of stream. Streams may be punctuated with *marks*, which are useful for synchronizing both ends of a connection. A mark is sent in a seperate packet that includes only one byte of data. Logically, it is like having a ninth bit on the data bytes. The Pup Package uses marks to implement the SST Change operation. It can be considered like an End-Of-Record or End-Of-File. Because they are sent in small (one data byte) packets, and may use the **SIGNAL** machinery, marks are not very efficient. An *interrupt* (called attention in the stream documentation) mechanisim is provided to bypaass the normal flow control restrictions. This is useful, for example, in Chat if the sender wants to flush the output buffer but the receiver is busy processing things in the buffer.

Address

Each end of a connection requires an *address* to identify it. Since there may be more than one active process on a machine, an address includes a socket number (like a post office box number) as well as the obvious network number and host number. In PupTypes, a **PupAddress** is a three word record. It is also copied into PupStream.

```
PupAddress: TYPE = RECORD [
   net: PupNetID, host: PupHostID, socket: PupSocketID];
PupNetID: TYPE = RECORD [Byte];
PupHostID: TYPE = RECORD [Byte];
PupSocketID: TYPE = RECORD [a, b: WORD];
```

There are a number of conventions associated with addresses (the second I in **fillIn** is a capital):
A local network of **fillInNetID** will be replaced by the local network number (if known).
A local host of **fillInHostID** will be replaced by the local host number.
A local socket of **fillInSocketID** will be replaced by a unique local socket number.

A local address of **fillInPupAddress** will be replaced by a unique local socket number.
A remote network of **fillInNetID** means use the local network.
A remote host of **fillInHostID** will be replaced by the local host number.

Servers, Users, and Listeners

Frequently a user will want to talk to another machine without having an operator at the remote site. FTP is the common example. In this case, the remote site is called a *server*, and the local end is referred to as the *user* site. The server is normally passive, merely responding to requests from the user. In actual implementation, there is usually an extra process called a *listener* that only waits passively for a user to try to establish a connection to a well known socket number, and then accepts the connection and creates a server process to do the real work.

Packets

The raw unit of transfer is a data structure called a *packet*. A packet lives in a buffer which also has room for all the other bookkeeping fields needed by the Pup Package. A packet consists of up to 532 bytes of user supplied data, and 22 bytes of control information manipulated by both the user and the Pup Package. Each packet has a *source* address and a *destination* address. These will be the same as the *local* and *remote* addresses of the connection if the packet is flowing towards the remote machine, but they will be reversed if the packet is flowing towards the user's machine. There are utility routines to implement most of the common operations on the control information, but the only operations provided on the data are to set and retrieve its length.

Where to find it

The Alto Mesa Pup Package lives on the <Mesa>Pup> directory of your favorite file server. It comes it two sizes: Tiny and Fat. The Tiny version is for production use. It does not include anything extra. The Fat version includes a lot of consistency checks and the Stats Package, too. Except for a few minor differences, they both export the same interfaces -- you can switch back and forth without recompiling.

You will be interested in the following files:

PupTypes.bcd+mesa includes the basic **RECORD** type definitions and the enumerations for **PupType** and other things that are not likely to change very often.

BufferDefs.bcd+mesa includes the definition for **Buffer** and **PupBuffer**.

PupDefs.bcd+mesa includes miscelleneaous **PROCEDURE** interfaces. You will need this if you are programming at the socket level. Otherwise, you probably won't need it unless you are doing something fancy.

PupStream.bcd+mesa (it is a Defs file in spite of its name) includes the the interface to the Pup ByteStream package. You will also need Stream.bcd to compile.

TinyPup.bcd is the tiny version of the PupPackage.

FatPup>FatPup.bcd is the fat version of the PupPackage. It contains the Stats package, and has been compiled with NIL and bounds checking. If you are having troubles finding an obscure bug, try using FatPup rather than TinyPup.

The bcd's for code modules in FatPup are different from those used to build TinyPup. (The public interfaces are the same.) If you want the debugger to cooperate, be sure to fetch the correct ones.

Turning the Pup Package on and off

The Pup Package uses some core for buffers and resident code if it is *on*. If it is not on, it cannot send or receive Pups. The Pup Package is designed to be shared by several client programs that do not know about each other. Thus each client program must turn it on and reserve its own copy of the Pup Package.

```
PupPackageMake: PROCEDURE;
PupPackageDestroy: PROCEDURE;
```

PupPackageMake will bump a use count and turn the Pup Package on if needed. **PupPackageDestroy** will decrement a use count and turn the Pup Package off if it reaches 0. Turning things on involves allocating the buffer pool, locking things into core, turning on the Ethernet Driver, and trying to initialize the Pup routing table. You can't call **MakeImage** while the Pup Package is on. (It gets very complicated to reinitalize the routing table and anything involving host numbers when starting up again.)

```
AdjustBufferParms: PROCEDURE [bufferPoolSize, bufferSize: CARDINAL];
```

AdjustBufferParms can only be called while the PupPackage is off. **bufferPoolSize** is the number of buffers that will be created when the PupPackage is turned on. The default is 15. Remember that the Pup Package keeps 2 in reserve for itself, and the normal Ethernet Driver uses 2 input buffers. **bufferSize** is the number of data words that will fit into the pup body of each buffer. The default is 266 words which is largest sized packet that gateways will forward. There is also a lower limit on the size of buffers. If you make it too small, you won't be able to talk to other networks because you won't be able to hear routing packets. The gateways break the routing table up into several packets if the size overflows 64 words, so that is the practical minimum. Actually, it is only the Mesa 6 Gateways that break the routing tables up into several packets, so the preceeding description may not be correct when you read this.

```
GetBufferParms: PROCEDURE RETURNS [bufferPoolSize, bufferSize: CARDINAL];
GetPupPackageUseCount: PROCEDURE RETURNS [CARDINAL];
```

GetBufferParms simply returns the current values of **bufferPoolSize** and **bufferSize**. If the PupPackage is not on, they are the numbers that will be used when it is next turned on. **GetPupPackageUseCount** returns the current use count of the communications package. If it is not 0, the Pup Package is on, and you should not call **AdjustBufferParms**, **MakeImage** or similar things.

Byte Stream interface

The Pup Package ByteStream (BSP) implementation uses a Stream interface that is different from the normal Mesa one defined in StreamDefs. See appendix A for a detailed description. This section only discusses the peculiarities of Pup ByteStreams.

```
PupByteStreamCreate: PROCEDURE [
   remote: PupAddress,
   ticks: Tocks]
   RETURNS [Stream.Handle];
Tocks: TYPE = RECORD [CARDINAL];
StreamClosing: ERROR [why: CloseReason, text: STRING];
CloseReason: TYPE = {
   localClose, remoteClose,
   noRouteToNetwork, transmissionTimeout, remoteReject};
```

PupByteStreamCreate is used to create a user end of a connection. It builds the necessary data structures and tries to open a connection to the specified remote address. **StreamClosing** will result if it can not be opened in a reasonable length of time.

If the connection gets into trouble for any reason, it is smashed closed, and any process that tries to do a get or put on the stream will run into the **StreamClosing ERROR**. **text** is likely to be **NIL**. If not, it is the text portion of the error or abort Pup provided by the remote end. **localClose** means that the stream is in the process of being deleted. If you ever get this, you are in trouble because you should not have any processes using a stream when you delete it. **remoteClose** means that the connection was closed by the remote machine. The other three reasons can also happen during **PupByteStreamCreate**. **noRouteToNetwork** means that the pup router does not know (any longer) how to get packets to the remote machine. Currently it isn't generated, but will probably reappear in the future. There are problems with transients in the routing tables. **transmissionTimeout** means that the remote machine has not responded within a reasonable amount of time. Maybe it has crashed, or if during **PupByteStreamCreate**, maybe it is not listening to the Ethernet. **remoteReject** means that the remote machine has explicitly rejected the connection or connection request. This frequently happens if the remote machine has been rebooted.

If the remote end has requested an ack, the PupPackage normally defers sending it until the receiving process sucks up all the data from the buffer. (If it asks again, it will get a prompt reply. Well, it was supposed to, but due to a bug...) If the sender is slightly faster than the receiver, this slight delay greatly improves throughput by keeping the allocation window as large as possible, and hence minimizing the number of ack packets that need to be generated and processed.

In addition to sending the current buffer, **SendNow** also requests that it be acknowledged. It will send an empty packet if necessary. This allows a bulk data transfer operation (FTP) to cooridinate it's (disk) activity with the (disk) activity of the process at the other end. Consider the case where the sender and receiver use the same size buffers, and the Pup buffers are slightly larger than the client buffers. Assume that it takes both ends roughly the same amount of time to generate and consume the data. Without the **SendNow**, the tail of the clump sits in the senders buffers while the sender

is fetching the second clump, and the receiver waits. Early in sending the second clump, the allocation limit will be reached, and the sender must wait for the receiver to process the first clump before the receiver asks for more data to trigger sending the ack. **SendNow** provides a way to keep things synchronized. The result is that the sender and receiver get to overlap the (disk) processing of the data. The current implementation waits in **SendNow** for the ack to arrive. Unless the receiving process at the remote end has fallen behind, this is only a short time. In particular, it is much smaller that the time necessary to write several pages on the disk.

Delete discards any partial buffer, if any, that has not yet been sent. This avoids bumping into **StreamClosing** again if you are cleaning up the connection after the remote end has crashed. If you want to send the data that might be buffered, be sure to call **SendNow**.

When you delete a stream, be sure that you don't have any other **PROCESS**es using it or the rug will get yanked out from underneath them.

Creating a listener tells the Pup Package that the client program will accept RFCs for the local socket number (the Pup Package knows the host and network). A listener also answers echo pups if you want a polite way to discover if it is alive.

```
CreatePupByteStreamListener: PROCEDURE [
   local: PupSocketID,
   proc: PROCEDURE [Stream.Handle, PupAddress],
   ticks: Tocks,
   filter: PROCEDURE [PupAddress] _ DontReject]
   RETURNS [PupListener];
Tocks: TYPE = RECORD [CARDINAL];
DestroyPupListener: PROCEDURE [PupListener];
PupListener: TYPE = ... ;
DontReject: PROCEDURE [PupAddress];
RejectThisRequest: ERROR [error: STRING];
```

CreatePupByteStreamListener makes a ByteStream *listener*. **local** is the local socket number to listen on. To stop listening, call **DestroyPupListener**.

The user supplied procedure will be **FORK**ed to once for each connection that is setup with the local socket number. (The new **PROCESS** gets **Detach**ed.) The client is expected to delete the connection eventually. The **PupAddress** passed to **proc** is the address of the other end of the connection. **ticks** is the timeout interval for get and put used for any streams that get created by the listener.

Optionally, you can supply a procedure that may reject undesired connections. This is useful to keep a server from crashing because it runs out of resources. (The default doesn't reject anything.) To reject a connection, the filter procedure should raise the **ERROR** **RejectThisRequest**. The text will be sent back to the other end in an abort Pup. If the filter procedure returns the connection will be accepted.

The Pup Package currently uses a very simple buffer allocation strategy. It allocates a fixed number of buffers for each half of each ByteStream. (It does the correct things if the other end allocates fewer buffers.) Thus it is easy to generate deadlocks. The default number is 5. (If you are counting buffers, don't forget the extra one that is being emptied/filled.) The default buffer pool size is 15. That is large enough to prevent lockups if a program uses both halves of a stream, or 2 halves of 2 streams. If you are tight on core and only using one half of a stream (at a time -- FTP is a good example), you should consider changing the buffer pool size.

```
SetMaxAllocation: PROCEDURE [CARDINAL];
SetMaxBufferSize: PROCEDURE [CARDINAL];
```

SetMaxAllocation will change the allocation limit for any ByteStreams that are created in the future. **SetMaxBufferSize** will change the maximun number of bytes allowed in a data packet in case you want to use less than the default. Existing streams are not effected.

```
SetPinging: PROCEDURE [BOOLEAN];
```

Normally, the PupPackage will try to verify that the remote end is still alive if the connection is idle for a minnute or so. If it has crashed, **StreamClosing[transmissionTimeout]** will eventually be generated. **SetPinging** will change this parameter for any ByteStreams that are created in the future. Again, existing streams are not effected.

There are currently four occasions for Timeouts. First, when trying to open a connection, an **ERROR** will be generated in one minute. Second, after a connection is opened, it will be smashed closed in 1 minute if an allocate is not received. Third, while sending data, the connection will be closed if a packet is not acknowledged within about 3 minutes. The Pup ByteStream package uses an adaptive timeout heuristic, so it is impossible to predict the exact number of retransmissions. If the other end goes into the debugger or otherwise dies, the retransmission interval should stabilize at 5 seconds. Fourth, if a connection is idle for a minute, a ping (empty aData) will be generated. Thereafter, they will be generated at the normal retransmission rate until an answer is received. If an answer is not received within two more minutes, the connection will be smashed closed.

Various other goodies

SetPupStormy: PROCEDURE [BOOLEAN];

SetPupStormy controls the *lightning* option in the PupRouter. If lightning is on, then occasional packets going in or out will be thrown away to simulate transmission errors. This is very handy for debugging high level protocols. The default is off. This option is not included in **TinyPup**. If you need it, use **FatPup**.

UseAltoChecksumMicrocode: PROCEDURE;
SetPupCheckit: PROCEDURE [BOOLEAN];

The packet format for a Pup provides for a software checksum in addition to any error checking that the hardware provides. This is very useful for avoiding trashed files in spite of flakey hardware or sick gateways. Internally, there are 3 options: **ignore**, **software**, and **microcode**. The default is **software**. **SetPupCheckit[**FALSE**]** sets the switch to **ignore**. You should do this only if you really need to go faster and won't complain if your data gets corrupted occasionally. **SetPupCheckit[**TRUE**]** sets the switch to **software**. **UseAltoChecksumMicrocode** sets the switch to **microcode**. You should use this only if you are sure that the correct microcode is in the RAM. RunMesa.run loads the RAM correctly if you are running on an Alto with 2K of PROM or a 3K RAM. The reason that the default isn't to use the microcode if the configuration looks reasonable is to avoid confusion when run with programs that load special microcode. Beware of getting restarted on a machine with a different configuration after if you use **UseAltoChecksumMicrocode** and call **MakeImage**.

SetLocalOnly: PROCEDURE [BOOLEAN];

SetLocalOnly controls the activation of the normal Ethernet Driver. If it is on, a dummy device is used rather than the Ethernet Driver. This allows bugs to be reproduced slowly without the complications of random packets coming in over the ethernet at unscheduled times. Of course, you won't be able to talk to any other machines if **localOnly** is on, but you can talk to yourself. This is not included in **TinyPup**. If you need it, include **LoopBackPlug** in your config.

CaptureErrors: PROCEDURE [PROCEDURE[ERROR]];

There are many internal consistency checks within the Pup Package. When a problem is detected, a procedure is called with an **ERROR** as an argument. (The **ERROR** is simply a convient way to get the compiler/system to generate a unique value. When an enumerated type is used, it is more time consuming to add a new category of problem to the list.) The default error handler simply dives into the debugger via **CallDebuger** from **MiscDefs**. If that happens to you, look at the stack. Its argument will get printed in english if you have enough symbols on your disk. If not, the **SignalLister** may help.

GetDoStats: PROCEDURE RETURNS [BOOLEAN];

The Pup Package includes a (compile time) optional statistics gathering facility. **GetDoStats** returns **TRUE** if it has been compiled in.

```
InspectIncomingPups: PROCEDURE [BOOLEAN, PROCDURE [CARDINAL, PupBuffer]];
InspectOutgoingPups: PROCEDURE [BOOLEAN, PROCDURE [CARDINAL, PupBuffer]];
    incomingPup: CARDINAL = 100;
    outgoingPup: CARDINAL = 101;
    zappedIncommingPup: CARDINAL = 102;
    zappedOutgoingPup: CARDINAL = 103;
ShowPupBuffer: PROCEDURE [CARDINAL, PupBuffer];
```

There are also hooks for inspecting each Pup as it goes in or out, and a routine that prints them. **ShowPupBuffer** is *not* part of the normal Pup Package. It lives in PupShow.bcd.

InspectIncomingPups[TRUE,ShowPupBuffer] will print all incoming Pups. You can, of course, supply your own routine to print them in the format that you prefer and/or print only the interesting ones.

Watchers

The *watcher* concept has two ideas. First, it allows a client program to look at Pups arriving when there is no socket ready to receive them. This would allow it to fire up a listener when an RFC arrives, and not clutter up core or other resources by listening when there is no need.

The second idea is to allow programs to pay attention to the Ethernet when the Pup Package is not really on. This saves a lot of core if there is no real need for the Pup Package. Again, if an RFC arrives, the Pup Package can be turned on and a listener started. (This has not been reimplemented since Mesa 3; If you need it, please let me know.)

```
InspectStrayPups: PROCEDURE [
  on: BOOLEAN,
  seeBroadCast: BOOLEAN,
  proc: PROCEDURE [PupBuffer] RETURNS [BOOLEAN]];
```

If **on** is TRUE, then watching is started. If not, it is stopped. After watching has been started, **proc** will be called each time a Pup arrives for an unknown socket. If it is a broadcast Pup, then **seeBroadCast** must also have been TRUE at the last call to **InspectStrayPups**. The **proc** must not do anything to the buffer - in particular, do not call **ReturnFreePupBuffer**. If the **proc** returns TRUE, the Pup Package sends an Error Pup (with subtype no-socket) back to the sender if it is not a broadcast packet or an error Pup.

PupDebug

There is also a set of debugging routines that are accessed via Menu commands in the Debugger. They only work if you are using FatPup. They are contained in PupDebug.bcd. It needs to be loaded into the Debugger's core image with a command line similar to the following:

```
>XDebug PupDebug/l
```

It adds its own menu to the main menu stack. The various commands will print the routing table, the socket table, a summary of the buffer headers, summary of the packet contents, or the current contents of the counters maintained by the Stats Package.

Name processing utilities

The Pup Package also provides a collection of commonly needed routines for translating names and addresses.

```
GetPupAddress: PROCEDURE [POINTER TO PupAddress, STRING];
PupNameLookup: PROCEDURE [POINTER TO PupAddress, STRING];
EnumeratePupAddresses: PROCEDURE [
   STRING, PROCEDURE [PupAddress] RETURNS [BOOLEAN] ],
   RETURNS [BOOLEAN];
PupAddressLookup: PROCEDURE [PupAddress, STRING];


PupNameTrouble: ERROR [e: STRING, code: NameLookupErrorCode];
NameLookupErrorCode: TYPE = {noRoute, noResponse, errorFromServer};
```

GetPupAddress first calls **ParsePupAddressConstant**, and if that fails, it calls **PupNameLookup**. **PupNameLookup** makes a special case check for a name of "ME" (both letters must be capitals). If so, it returns the address of the local machine. Otherwise it sends the text string to the name lookup server which is normally running on a nearby Gateway. Since a machine may have several addresses, **PupNameLookup** selects the best one of the answers returned.

There are three reasons why **PupNameLookup** may not be able to supply a reasonable answer. If so, it will generate the ERROR **PupNameTrouble** with a text string that might help a person and an error code that might be useful by a program. In the first case, **noRoute**, the information was received from the gateway, but there is no way to get to the target machine. In the second, **noResponse**, no name lookup server responded within a reasonable length of time. Unfortunately, the third case, **errorFromServer**, does not help a program much. The protocol does not supply error codes useful to programs, but the desired information is probably in the text string.

If the socket number from the text string is not explicitly specified, the socket number in the **PupAddress** will not be updated. Be sure to initialize it!. This allows a program to initialize a default value that can easily be overridden for debugging by a person typing in a text string that is normally just a machine name.

The text string is either the *name* of a machine or the actual *address* of the machine encoded as "<host>#" for a machine on the local network, or "<net>#<host>#" to specify an explicit network number. If you also want to specify the remote socket number use "<name>+<soc>" or "<net>#<host>#<soc>". <name> is a text string registered in the file <Portola>PUP-NETWORK.TXT on Ivy. <net>, <host>, and <soc> are octal numbers. See Ed Taft's memo, [Maxc]<Pup>PupName.press, for the fine print.

EnumeratePupAddresses is similar to **PupNameLookup** but it gives the client all of the addresses in the database in case there is more than one. This is useful for locating backup servers in case the primary one is down. The nearest addresses are presented first and the order in the name lookup database is preserved when there are several answers equally close. It an address is currently not reachable, the entry is still presented to the client. (**PupNameLookup** simply picks

the first one.) Although it is possible to send and receive Pups from within the client supplied procedure (for example, to see if that machine is up) that should probably be avoided since the buffer that the answer arrived in (and others with duplicate answers if there is more than one name lookup server on the local network) is still tied up. If the client procedure returns **TRUE**, the rest of the entries are ignored. **EnumeratePupAddresses** returns **FALSE** if the client procedure never returns **TRUE**.

> **ParsePupAddressConstant: PROCEDURE [POINTER TO PupAddress, STRING]**
> **RETURNS [BOOLEAN];**

ParsePupAddressConstant tries to parse the text string into a network address. The string must be of the form <host>#, <net>#<host>#, or <net>#<host>#<soc>. Since socket numbers may be 32 bits, the form <soc-high>|<soc-low> may also be used to specify a socket number. All numbers are octal. Invalid characters, oversize numbers and whatever merely return **FALSE**.

> **AppendPupAddress: PROCEDURE [STRING, PupAddress];**
> **AppendHostAddress: PROCEDURE [STRING, PupAddress];**
> **AppendMyName: PROCEDURE [STRING];**

AppendPupAddress appends a text translation of the specified address to the string. The format is net#host#soc, where soc may be more than 16 bits. All numbers are in octal. **AppendHostAddress** uses **PupAddressLookup** to translate the address into a name. If it runs into troubles, it calls **AppendPupAddress**. **AppendMyName** fabricates an address for the local machine, and then calls **AppendHostAddress**.

Although not really needed for name processing, **GetHopsToNetwork** is normally used along with other routines in this section, so it is described here.

> **GetHopsToNetwork: PROCEDURE [PupNetID] RETURNS [CARDINAL];**

GetHopsToNetwork can be used to help you decide which machine you want to talk to. It returns a huge number if the network number is unreasonable or there is currently no way to get there from here. In the future, the Pup Package may be able to provide more information.

Packet utilities

This section describes the basic things that client programs can do to packets and/or buffers. They will be used by client programs that interface at the Socket level.

We have adopted the convention that fields in the buffer that can be simply read/written by client programs should be accessed directly for efficiency. Such fields include the **pupType**, the **source** and **destination PupAddress**es, and the data. A **PupBuffer** is an **OVERLAID RECORD**, so you can access the data portion of it as **b.pupBytes[i]** (a **PACKED ARRAY OF Byte**s), **b.pupChars[i]** (a **PACKED ARRAY OF CHARACTER**s) or **b.pupWords[i]** (an **ARRAY OF WORD**s). If your data is a **RECORD** (a very good idea) you can **LOOPHOLE** a **POINTER** to your record to be **@b.pupBody**. We have also provided a pair of simple routines to access the other interesting field - the length.

> **SetPupContentsBytes: PROCEDURE [PupBuffer, CARDINAL];**
> **SetPupContentsWords: PROCEDURE [PupBuffer, CARDINAL];**
> **GetPupContentsBytes: PROCEDURE [PupBuffer] RETURNS [CARDINAL];**
> **DataWordsPerPupBuffer: PROCEDURE RETURNS [CARDINAL];**

There are two lengths involved here. **pupLength** is the total length of the pup *in bytes, not words*, and exists as a field in the packet. The contents length includes only the text supplied by the user. Today, **pupLength** is the contents length plus 22. Be sure not to overfill a buffer -- you will clobber the header of the next buffer. **DataWordsPerPupBuffer** can be used to determine the maximum number of words that will fit in the body of a Pup. There is no **GetPupContentsWords** because I couldn't think of anything reasonable to do if there was an odd number of data bytes in the Pup.

Fine point: If somebody is playing with large buffers, and you expect your packets to be forwarded by a gateway, be sure to round down to 266 words (**PupDefs.maxDataWordsPerGatewayPup**) before deciding how full you can fill a buffer.

> **ReturnFreePupBuffer: PROCEDURE [PupBuffer];**
> **GetFreePupBuffer: PROCEDURE RETURNS [PupBuffer];**

ReturnFreePupBuffer is used to put a buffer back on the **freeQueue**. It must be called to return the buffers obtained from **socket.get**. **GetFreePupBuffer** obtains one buffer from the **freeQueue**. It will wait if there is not a buffer currently available. This is how a client program gets a buffer to give to **socket.put**.

The Pup Package builds (at **PupPackageMake** time) a pool of 15 buffers. The Ethernet driver uses two and **GetFreePupBuffer** always leaves two in reserve, so there are really only 11 left for all the active users to share. Pup byte streams allocate a default of five packets, so lockups are easily possible if you have several active streams. Note that each active half of a stream can use up to 6 buffers.

SwapPupSourceAndDest: PROCEDURE [PupBuffer];

SwapPupSourceAndDest exchanges the **source** and **dest** fields of the PupBuffer. It fixes up the new destination if the old destination Pup was broadcast. Of course, it only makes sense if the buffer contains a packet that was transmitted from some (probably different) machine.

MoveStringBodyToPupBuffer: PROCEDURE [PupBuffer, STRING];
AppendStringBodyToPupBuffer: PROCEDURE [PupBuffer, STRING];

MoveStringBodyToPupBuffer copies the contents of the **STRING** into the body of the **PupBuffer** and sets the length accordingly. It is handy for very simple protocols and is used by **PupNameLookup**. **AppendStringBodyToPupBuffer** appends the contents of the **STRING** into the body of the **PupBuffer** and adjusts the length accordingly. Both routines will call **Glitch** if the **STRING** is **NIL** or it won't fit in the buffer.

Socket interface

The lowest level of the Pup Package accessible to the user is the Socket interface. At this level packets are delivered only with some reasonable probability, and there is no guarantee that they will not arrive out of order or even twice. The functions performed at this level have to do only with delivery. There is still the idea of local and remote addresses, and there is a routing function which decides what to do with the packets depending on those addresses. The socket is basically an interface to that routing function, which makes its name known in the appropriate tables and allows multiple users to coexist in the same machine.

```
PupSocket: TYPE = POINTER TO PupSocketObject;
PupSocketObject: TYPE = RECORD [
   put: PROCDURE [PupBuffer],
   get: PROCDURE RETURNS [PupBuffer],
   setRemoteAddress: PROCDURE [PupAddress],
   getLocalAddress: PROCDURE RETURNS [PupAddress]  ];
```

The Socket Operations are:

```
PupSocketMake: PROCEDURE [
   local: PupSocketID, remote: PupAddress, ticks: Tocks]
   RETURNS [PupSocket];
Tocks: TYPE = RECORD [CARDINAL];
```

PupSocketMake creates the data structures necessary to communicate to the specified destination. It initiates no actual communications. If **fillInSocketID** is specified for the local socket number, a pseudo unique random socket number will be assigned.

```
PupSocketDestroy: PROCEDURE [PupSocket];
```

PupSocketDestroy reclaims the data structures used by a socket and removes the socket from the routing tables.

```
socket.put[b];
```

socket.put sends the buffer to the previously specified destination. Any problems encountered are ignored. When the buffer has been sent, which is normally right away, but may take several seconds, the client's requeue procedure is called. The requeue procedure lives in ***b*.requeueProcedure**. It is initialized to **ReturnFreePupBufffer** by **GetFreePupBuffer**. If you want to keep a buffer so that you can retransmit it (or otherwise reuse it) you must store something into ***b*.requeueProcedure**. If you do use your own requeue procedure, be sure that the buffer is eventually returned via **ReturnFreePupBuffer**.

```
PupBuffer _ socket.get[];
```

socket.get returns a buffer containing the next packet that is sent to the socket's local socketID. The Pup Package does not do any filtering on the source address, so you will get packets from unexpected places. When you are finished processing the buffer, you should call **ReturnFreePupBufffer**. If a packet does not arrive before the specified timeout, **NIL** is returned instead. Note that the Pup may have come from anywhere, the remote address is only used for sending. If you care, you should check the source of the Pup.

socket.setRemoteAddress[*PupAddress*];

socket.setRemoteAddress changes the destination used by **socket.put**. Until it is changed again, all Pups send via **socket.put** will go to the specified address.

***PupAddress* _ socket.getLocalAddress[];**

socket.getLocalAddress returns the local address of a socket. This tells you two things. One, if you used **fillInSocketID** for the local socket number when you created the socket, it will tell you the actual socket number that is in use. Second, if you have more than one hardware interface on your machine, it will tell you which one is the best path.

The normal sequence of operations is to make a socket and then, enter a loop that gets a **PupBuffer** from the **freeQueue**, fills it in, sends it, waits for a reply, processes the incoming **PupBuffer**, and returns it to the **freeQueue**. Finally, when finished, don't forget to delete the socket. It is quite reasonable to have two **PROCESS**es, one to look at incoming packets and another to send things out. Even if you don't expect any incoming data, be sure to have some **PROCESS** getting packets occassionally or buffers will pile up there if anybody ever does send you something. It is quite likely that error pups will be returned if a Gateway runs out of buffers, or the other end ever gets confused, rebooted, or ....

There are two other routines that are useful for sending Pups. In both cases the client program must fill in the source and destination socket numbers.

PupRouterBroadcastThis: PROCEDURE [PupBuffer];
PupRouterSendThis: PROCEDURE [PupBuffer];

PupRouterBroadcastThis will broadcast a Pup on all directly connected networks. The Pup Package will provide the correct destination net, source host, and source net numbers for each network as the Pup is transmitted.

NB: Do not use the broadcast facility indiscriminately. Random undesired packets can have a severe performance impact on some programs. In general, it should only be used to locate a machine that you want to comunicate with.

PupRouterSendThis is the simplest way to send a packet. Don't forget to fill in the source socket, destination address, pupType and set the length. The Pup Package will fill in the correct source net and host numbers. There is a special case check for a destination host of **allHosts** (0) and a destination net of **allNets** (0). If so, the buffer is handed off to **PupRouterBroadcastThis**. If you specify **allHosts** and a non 0 network, the Pup is a directed broadcast, and it will be broadcast on that specific network, and not on all directly connected

networks. There is no way to broadcast a Pup on network 0 without broadcasting it on the other networks too.

```
SendPup: PROCEDURE [b: PupBuffer, type: PupType, bytes: CARDINAL];
ReturnPup: PROCEDURE [b: PupBuffer, type: PupType, bytes: CARDINAL];
```

SendPup fills in the pupType and the length of the buffer, and then calls **PupRouterSendThis**. **ReturnPup** calls **SwapPupSourceAndDest** and then calls **SendPup**.

```
SendErrorPup: PROCEDURE [PupBuffer, PupErrorCode, STRING];
```

SendErrorPup treats the buffer as a Pup that was sent to this machine, converts it into an Error Pup with the specified error code and text, and sends it back to the original source. The buffer must have been sent to this machine since **SendErrorPup** needs to know the network address through which the packet arrived, so don't pass it a buffer that you obtained from **GetFreePupBuffer**. If **STRING** is **NIL**, a built in list of defaults is searched. Actually, the buffer isn't sent if it is already an error Pup, or if it is a broadcast Pup.

Tock conversion

There are several routines in the Pup Package that normally **WAIT** until something has happened. The argument to the routines that setup the wait times is a simple record.

```
Tocks: TYPE = RECORD [CARDINAL];
    veryLongWait: Tocks = [177777B];
    veryShortWait: Tocks = [0];
```

veryLongWait is used to disable timeouts completely. **veryShortWait** is used to avoid waiting at all. All other values are passed directly to **ProcessDefs.SetTimeout**.

```
SecondsToTocks: PROCEDURE [CARDINAL] RETURNS [Tocks];
MsToTocks: PROCEDURE [CARDINAL] RETURNS [Tocks];
```

These routines simply do a bit of arithmetic and the appropriate **TYPE** conversion. Their input is in seconds and milliseconds respectively. Note that it is not possible to specify wait times longer than about a minute in milliseconds.

Queue operations

The Pup Package provides a simple FIFO Queue Package which may be useful if you are operating at the socket level. NB: The Queue Package does *not* provide any locks against preemption. The client is expected to call the queue routines from within an appropriate **MONITOR**.

Queue: TYPE = POINTER TO **QueueObject;**
QueueObject: TYPE = RECORD **[length:** CARDINAL**, ...];**

length is the number of buffers currently on the queue. It is handy for various heuristics. In particular, you may want to check for an empty queue before calling **DequeuePup** if you are worried about performance.

QueueInitialize: PROCEDURE **[Queue];**
QueueCleanup: PROCEDURE **[Queue];**

QueueInitialize simply initializes the data structures within a **QueueObject** allocated by the caller. It must be called before any other queue operations reference the **QueueObject**. **QueueCleanup** returns any buffers on the queue to the **freeQueue** and marks it as invalid. You must call **QueueInitialize** again before reusing the queue.

EnqueuePup: PROCEDURE **[Queue, PupBuffer];**
DequeuePup: PROCEDURE **[Queue]** RETURNS **[PupBuffer];**
ExtractPupFromQueue: PROCEDURE **[Queue, PupBuffer]** RETURNS **[PupBuffer];**

EnqueuePup puts a **PupBuffer** onto the tail of a queue. **DequeuePup** takes a **PupBuffer** off of the head of a queue. If the queue is empty, it returns **NIL**. **ExtractPupFromQueue** pulls a specified buffer out of the middle of a queue. If the buffer is not on the queue, it returns **NIL**.

QueueLength: PROCEDURE **[Queue]** RETURNS **[**CARDINAL**];**
QueueEmpty: PROCEDURE **[Queue]** RETURNS **[**BOOLEAN**];**

These are just fast, clean, and simple ways to see if there is anything in a **Queue**.

Bugs and gremlins

Note that most program bugs do not show up as uncaught **SIGNALs**, but instead call an error handler, which defaults to a routine that calls the debugger. If you suddenly end up in the debugger with a "PupGlitch" message, look at the stack. If you find **Glitch**, look at its parameter. If you have enough symbols on your disk the name will usually be reasonably clear what the problem is. If not, consult a friend or dig out the listings. Glitches are problems from which it is not reasonable to recover. Usually, they are caused by bugs in the client program and/or inadequate documentation. If you get one, you should not attempt to continue execution of your program. Because of various validity checks in the Pup Package, core clobbers frequently show up as Pup Glitches.

Signals

Except for the EFTP routines, the following summary is an exhaustive list of the signals explicitly generated by the Pup Package. It does NOT include signals generated from within the system when the Pup Package calls a system routine. They all represent well understood circumstances, and even if you didn't want the signal at that time, the Pup Package knows what is going on and is passing the problem back to the client program where it belongs.

```
Stream.Timeout: SIGNAL;
```

Stream.Timeout is generated by the ByteStream input routines if enough data does not arrive in time. The actual wait times are not very accurate.

```
StreamClosing: ERROR [why: CloseReason, text: STRING];
CloseReason: TYPE= {localClose, remoteClose,
  transmissionTimeout, noRouteToNetwork, remoteReject};
```

StreamClosing is generated by **ByteStreamOpen** when the remote site explicitly rejects the connection, does not accept it in time, or there is not any way to get a packet to the remote site. (Currently, the timeout is about 60 seconds.)

After a stream is opened, **StreamClosing** is activated by many ByteStream routines whenever an attempt to use a connection is made after the Pup Package connection has been closed. It may be closed by the remote end, or automatically by the Pup Package when it discovers some problem. The connection is closed automatically when ever the remote end explicitly rejects a packet (maybe it has been rebooted) or whenever a packet is not acknowledged soon enough (maybe it is in the debugger). On input, all the data from the remote site is processed before activating the **StreamClosing** **ERROR**.

```
PupNameTrouble: ERROR [e: STRING, code: NameLookupErrorCode];
```

```
NameLookupErrorCode: TYPE = {
  noRoute, noResponse, errorFromServer};
```

PupNameTrouble is only generated by **EnumeratePupAddresses** and **PupNameLookup** and **GetPupAddress** which call it, and **PupAddressLookup**. There are 3 reasons why **EnumeratePupAddresses** or **PupAddressLookup** may not be able to supply a reasonable answer. If so, it will generate the **ERROR** **PupNameTrouble** with a text string that might help a person and an error code that might be useful by a program. In the first case, **noRoute**, the information was received from the name server, but there is no way to get to the target machine. In the second, **noResponse**, no name lookup server responded within a reasonable length of time. Unfortunately, the third case, **errorFromServer**, does not help a program much. The protocol does not supply error codes useful to programs, but the desired information is probably in the text string.

EFTP Package

The EFTP Package implements a very simple Pup-based Ethernet File Transfer Protocol. This protocol is currently used to communicate with Press printers in the Parc environment. The package comes in two separate modules that are not included in either FatPup or TinyPup: EFTPSend.bcd and EFTPRecv.bcd. Only *one* connection per module is supported: if you want more than one active connection in the same direction you will have to make another instance.

Each half should not be called from more than one **PROCESS**. I doubt if it will recover correctly if you **ABORT** it. The intention is that it will timeout often enough so that there will not be a need to use **ABORT**.

```
EFTPTimeOut: SIGNAL;
```

Many of the procedure calls may generate the resumable **SIGNAL** **EFTPTimeOut**. The timeouts used within the EFTP Package are very simple. The default timeout for both send and receive wakeups is 1 second. Someday, I may get around to implementing an adaptive timeout scheme. The **SIGNAL** is generated under the following situations:

Open for Sending **SIGNAL**s every 10 retransmissions.

Put **SIGNAL**s every 25 retransmissions.

Finish Sending **SIGNAL**s every 10 retransmissions.

Open for Receiving **SIGNAL**s every 10 ticks.

Get **SIGNAL**s every tick.

```
EFTPSetSendTimeout: PROCEDURE [ms: CARDINAL, tries: CARDINAL];
EFTPSetRecvTimeout: PROCEDURE [ms: CARDINAL];
```

EFTPSetSendTimeout is used to change the timeout parameters used in the send side of the EFTP Package. **EFTPSetRecvTimeout** is used to change the timeout used in the receive side of the EFTP Package. The values of the timeouts are measured in milliseconds.

```
EFTPAbortCode: TYPE = RECORD [WORD];
  eftpOK: EFTPAbortCode = [0]; -- pseudo code
  eftpExternalSenderAbort: EFTPAbortCode = [1];
  eftpExternalRecieverAbort: EFTPAbortCode = [2];
  eftpReceiverBusyAbort: EFTPAbortCode = [3];
  eftpOutOfSyncAbort: EFTPAbortCode = [4];
  eftpRejected: EFTPAbortCode = [1001]; -- pseudo code

EFTPOpenForSending: PROCEDURE [
  to: PupAddress, waitForAck: BOOLEAN _ TRUE];
      EFTPAlreadySending: ERROR;
```

EFTPTroubleSending: ERROR **[e: EFTPAbortCode, s:** STRING**];**
EFTPTimeOut: SIGNAL**;**

EFTPOpenForSending is used to create an EFTP connection to the remote address **to**. If there is already an EFTP connection then the ERROR **EFTPAlreadySending** is generated. If **waitForAck** is TRUE, **EFTPOpenForSending** transmits a packet with sequence number 0 and no data, until the packet is acknowledged. This is a way to be reasonably sure that the other end is willing to talk to you if you don't want to open a file or such until the connection is open. If there is any problem in transmitting the packet, then the ERROR **EFTPTroubleSending** is generated. The SIGNAL **EFTPTimeOut** will be generated if the initial packet is not acknowledged soon enough. It may be resumed.

EFTPSendBlock: PROCEDURE **[p:** POINTER**, l:** CARDINAL**];**
EFTPNotSending: ERROR**;**
EFTPTroubleSending: ERROR **[e: EFTPAbortCode, s:** STRING**];**
EFTPTimeOut: SIGNAL**;**

EFTPSendBlock is used to send a block of data over the open EFTP connection. **p** is a pointer to the data and **l** is the length in bytes. If the EFTP connection has not been opened then the ERROR **EFTPNotSending** is generated. If there is any problem in transmitting the data, then the ERROR **EFTPTroubleSending** is generated. The SIGNAL **EFTPTimeOut** may be generated. This SIGNAL may be resumed.

EFTPAbortSending: PROCEDURE **[s:** STRING**];**

EFTPAbortSending is used to abort the EFTP connection and to transmit the string **s** to the destination indicating some reason for aborting the EFTP connection.

EFTPFinishSending: PROCEDURE**;**
EFTPNotSending: ERROR**;**
EFTPTimeOut: SIGNAL**;**

EFTPFinishSending is used to terminate the EFTP connection. If the EFTP connection has not been opened then the ERROR **EFTPNotSending** is generated. If there is any problem in transmitting the data and closing the connection, then the ERROR **EFTPTroubleSending** is generated. The SIGNAL **EFTPTimeOut** may be generated. This SIGNAL may be resumed.

EFTPOpenForReceiving: PROCEDURE **[PupAddress]** RETURNS **[PupAddress];**
EFTPAlreadyReceiving: ERROR**;**
EFTPTimeOut: SIGNAL**;**

EFTPOpenForReceiving is used to wait for an EFTP connection to be opened from any remote address. If there is already an EFTP connection then the ERROR **EFTPAlreadyReceiving** is generated. This procedure waits for a packet with sequence number 0, and acknowledges it. When this happens the EFTP connection is open. The SIGNAL **EFTPTimeOut** may be generated. This SIGNAL may be resumed.

```
EFTPGetBlock: PROCEDURE [p: POINTER, l: CARDINAL] RETURNS [CARDINAL];
    EFTPNotReceiving: ERROR;
    EFTPEndReceiving: ERROR;
    EFTPTroubleReceiving: ERROR [e: EFTPAbortCode, s: STRING];
    EFTPTimeOut: SIGNAL;
```

EFTPGetBlock is used to receive a block of data over the open EFTP connecction. **p** is a pointer to the data block and **l** is the length in bytes. The procedure call returns the actual number of bytes stored into the block. If the EFTP connection has not been opened then the **ERROR** **EFTPNotReceiving** is generated. If the EFTP connection is terminated or aborted by the remote end, then the **ERROR** **EFTPEndReceiving** is generated. If there is any problem in receiving the data, then the **ERROR** **EFTPTroubleReceiving** is generated. The **SIGNAL** **EFTPTimeOut** may be generated. This **SIGNAL** may be resumed.

```
EFTPAbortReceiving: PROCEDURE [s: STRING];
```

EFTPAbortReceiving is used to abort the EFTP connection and to transmit the string **s** to the destination indicating some reason for aborting the EFTP connection.

```
EFTPFinishReceiving: PROCEDURE;
    EFTPNotReceiving: ERROR;
```

EFTPFinishReceiving is used to terminate the EFTP connection. If the EFTP connection has not been opened then the **ERROR** **EFTPNotReceiving** is generated.

Multiple EFTP connections

Since there is no handle passed to the EFTP routines, it is not possible for a particular instance of the code to keep track of more than one connection (in a particular direction) at a time. If you need more than one, you will have to Bind in more instances of **EFTPSend** or **EFTPRecv** inot your config. Be sure that you do not use code links on your module that is calling EFTP if you have several instances if it. Also, be careful when using code links on **EFTPRecv**. It imports **EFTPTimeOut** from **EFTPSend**, and if you have several instances of both there is opportunity for confusion.

ByteBlt

Within the Pup BSP implementation module, there is a complication that arises when buffers are not aligned conveniently. In case anybody else ever runs into the same problem, the critical routine has been split out into a separate interface called **ByteBltDefs**. **ByteBlt** uses Blt for the easy case, and BitBlt to do the ripple.

```
ByteBlt: PROCEDURE [to, from: Stream.Block] RETURNS [nBytes: CARDINAL];
        StartIndexGreaterThanStopIndexPlusOne: ERROR;
Block: TYPE = RECORD [
  blockPointer: LONG POINTER,
  startIndex, stopIndexPlusOne: CARDINAL];
```

ByteBlt moves as much as it can, and tells you what that was. Zero is ok. **StartIndexGreaterThanStopIndexPlusOne** will be generated if either argument is trying to wrap around memory.

ByteBlt will probably not work as desired if the source and destination overlap.

```
ShortenPointer: PROCEDURE [LONG POINTER] RETURNS [POINTER];
        HyperSpaceNotSupported: ERROR;
        NilRejected: ERROR;
```

On an Alto, the **LONG POINTER** in a **Block** must be shortened before it can be used. If either of two unreasonable conditions are detected the obvious **ERROR**s are generated.

Stats Package

The Stats Package may help you to understand what your program is doing. Basically, it just maintains a set of counters that can be printed out when desired. The counters are **LONG CARDINAL**s, and they are indexed by a huge enumerated type. Most of the types are pretty specific to the Pup Package, but there are a clump of spares. The Stats Package is included in **FatPup**. The interface is **StatsDefs**.

```
StatNew: PROCEDURE;
StatStart: PROCEDURE [STRING];
StatPrintCurrent: PROCEDURE;
StatFinish: PROCEDURE;
```

StatNew should be called before calling **MakeImage**. It just remembers the current date and time to be used in a header line by **StatStart**. **StatStart** resets all the counters, and prints out a header line. **StatPrintCurrent** prints out the current values of the counters without changing them. **StatFinish** prints out the current values of the counters and resets them in case you start another test. The printout also includes a few derived numbers, like bits per second sent and received.

```
StatIncr: PROCEDURE [StatCounterIndex];
StatBump: PROCEDURE [StatCounterIndex, CARDINAL];
StatLog: PROCEDURE [StatCounterIndex, POINTER, CARDINAL];
StatCounterIndex: TYPE = {....};
```

StatIncr adds one to the specified counter. **StatBump** adds a specified amount. You will have to look in **StatsDefs** to find the values of **StatCounterIndex**. They change every now and then. **StatLog** adds one to the specified counter, and prints out the specified block of memory. Someday, it may do something more interesting, like send the data someplace for later analysis. The idea is that if an obscure glitch has happened, the data may be useful in tracking it down.

```
StatReady: PROCEDURE;
StatSince: PROCEDURE;
```

There is also another set of counters that can be used for specific tests without disturbing the main counters. **StatReady** resets the secondary counters. **StatSince** prints out everything that has happened since the last call to **StatReady** or **StatSince**, and then resets the secondary counters.

```
StatsStringToIndex: PROCEDURE [STRING] RETURNS [StatCounterIndex];
    StatStringsFull: ERROR;
```

If you want to add your own counters at runtime, **StatsStringToIndex** will search the string table, and assign a free slot if it isn't already there. The answer can then be used as an argument to **StatIncr** or **StatBump**. **StatStringsFull** will be generated if there are no more spare slots. Feel free to use counters that are already there if you prefer.

```
StatUpdate: PROCEDURE;
StatsGetCounters: PROCEDURE RETURNS [
  POINTER TO ARRAY StatCounterIndex OF LONG CARDINAL];
StatsGetText: PROCEDURE RETURNS [
  POINTER TO ARRAY StatCounterIndex OF STRING];
```

If you get really desperate and need to wander around in the bits, these routines will give you pointers to the inner secrets. If a string is **NIL**, that slot is considered to be unused. **StatUpdate** updates several clock counters. You should call it before looking at any of them.

References

For details of the EFTP protocol: [Maxc]<Pup>EftpSpec.press.

For details on Error Pups: [Maxc]<Pup>Error.press.

For the truth on Pup Names: [Maxc]<Pup>PupName.press.

For general background about Pups: [Maxc]<Pup>PupSpec.press.

You might also want to browse through (the rest of) [Maxc]<Pup>*.press.

Appendix A: Stream Package

The Pup Package uses a stream interface different from the standard Alto/Mesa one. Pup's version includes, among other things, subsequences (a generalization of mark bytes) and an attention (break) facility. Note that this appendix describes the stream interface in a general (device and implementation independent) way; see the previous section on the Byte Stream Interface for Pup specifics.

The stream package defined by the interface **Stream** provides a device and format independent interface for sequential access to a stream of data. In particular,

> It provides a vehicle by which processes or subsystems can communicate with each other, whether they reside on the same machine or on different machines.
>
> It permits processes or subsystems to transmit arbitrary data to or from storage media in a device-independent way.
>
> It defines a standard way for transforming the detailed interface for a device into a uniform, high level interface which can be used by other client software.
>
> It provides an environment for implementing simple transformations to be performed on the data as it is being transmitted.
>
> It provides optional access to and control over the mapping of data onto the physical format of the storage or transmission medium being used.

The stream package provides several facilities, not all of which may be important to an individual client. First, there is the *stream interface*, the set of procedures and data types by which a client actually controls the transmission of a stream of information. Each of the operations of the stream interface takes as a parameter a **Stream.Handle** which identifies the particular stream being accessed. Second, the stream package defines the concepts of *transducer* and *filter*. A transducer is a software entity (e.g. module or configuration) which implements a stream connected to a specific device or medium. A filter also implements a stream, but only for the purpose of transforming, buffering, or otherwise manipulating the data before passing it on to another stream. Transducers and filters may be provided either by the system or by clients. Third, the stream package provides a standard way of concatenating a sequence of filters (usually terminated with a transducer) to form a compound stream called a *pipeline*. A pipeline is accessed by means of the normal stream operations, and causes a sequence of separate transformations to be applied to data flowing between the client program at one end and the physical storage (or transmission) medium at the other.

The use of pipelines permits clients to interpose new stream manipulation programs (filters and transducers) between clients (producers and consumers of data) without modifying the interfaces seen by the clients. For example, a data format conversion program can obtain its data either from a tape or from a disk, using the same stream interface, and hence the same program logic for both. Similarly, filters performing such functions as code conversion, buffering, data conversion, encryption, etc. may be inserted into a pipeline without affecting the way the client sends and receives data through the stream interface.

The stream facility transmits arbitrary data, regardless of format and without prejudice to its type or characteristics. The data may comprise a sequence of bytes, words, or arbitrary Mesa data structures. The stream facility does not presume or require the encoding of information according to any particular protocol or convention. Instead, it permits clients to define their own protocols and standards according to their own needs.

In this appendix, sections *A.1*, *A.2*, and *A.3* will be of interest to all clients. Section *A.4* will be of interest only to those clients wishing to control the physical record characteristics of a particular stream. Section *A.5* will be of interest only to those clients wishing to implement their own filters or transducers. In addition, the clients of a particular stream type (e.g. disk, tape, etc.) will normally have to consult separate documentation regarding the details of that kind of stream.

A.1 Semantics of Streams

The stream facility supports transmission of a sequence of eight-bit bytes. This sequence may be divided into identifiable *subsequences*, each of which has its own *subsequence type*.

```
Stream.Byte: TYPE = [0..255];

Stream.SubSequenceType: TYPE = [0..255];
```

A subsequence may be null; i.e., it may be of zero length and contain no bytes but still contain the **SubSequenceType** information. This information allows all subsequences to be easily identified and separated from each other while shielding clients from the bothersome problems of control-codes (i.e. embedding control-codes into the stream, making them transparent, and building a parser to implement such transparency).

Additionally, an *attention* flag may be inserted into a stream sequence. This is neither a byte nor a **SubSequenceType**, but simply an indication of an extraordinary situation. Attention flags are transmitted through the stream as quickly as possible, possibly bypassing bytes and changes in **SubSequenceType** which were transmitted earlier but which are still in transit. This provides a simple mechanism for implementing breaks (similar to the "attention-key" of many time-sharing systems).

Streams *per se* have no notion of a byte number (i.e. an array index); they deal only with the sequential order of successive bytes comprising the arbitrary binary data being transmitted. The stream interface is thus unsuitable for applications requiring random access.

Streams have no intrinsic notion of the bytes passing through them being grouped into physical records. The client program can completely ignore physical record structure and is thus relieved of the burden of dealing with the associated packing and unpacking problems. If, however, it becomes necessary to control or determine the underlying physical record structure, as determined by the particular storage (or transmission) medium, the interface provides extended facilities which allow this.

All of the procedures described here are synchronous and none returns until the indicated operation is complete. That is, an input operation does not return until the data is actually available to the client, and an output operation does not return until the data has been accepted by the stream and client buffers may be reused. Note, however, that a stream component *may* do internal buffering and that the acceptance of data means only that the stream component itself has a correct copy and is in a position to proceed asynchronously to write or send it.

Streams are inherently full-duplex -- i. e., separate processes may be transmitting and receiving simultaneously. The stream interface does *not* guarantee mutual exclusion among different processes attempting to access the same stream. However, individual transducers or filters may restrict themselves to half-duplex operation and/or may implement such mutual exclusion or more elaborate forms of synchronization as are appropriate. Documentation for such filters and transducers should be consulted on a case-by-case basis for details.

A.2 Operations on Streams

The stream interface provides the following information transmission operations: **GetBlock**, **GetByte**, **GetChar**, **GetWord**, **PutBlock**, **PutByte**, **PutChar**, **PutWord**, **SendNow**, **SetSST**, **SendAttention**, and **WaitForAttention**.

A client program identifies a particular instance of the stream interface by means of a **Stream.Handle**.

```
Stream.Handle: TYPE = . . . ;
```

A **Handle** identifies an object (see :*A.5.1*) which embodies all of the information concerning the transfer of data to and/or from the client program via stream operations. It is passed as a parameter to each of the data transmission operations of the following sections to specify the stream to which the operations apply.

A.2.1 **GetBlock** *and* **PutBlock**

The principal operations for transferring blocks of data are **Stream.GetBlock** and **Stream.PutBlock**. Each of these takes a parameter specifying the block of virtual memory to or from which bytes are to be transmitted.

```
Stream.Block: TYPE = RECORD [
    blockPointer: LONG POINTER,
    startIndex, stopIndexPlusOne: CARDINAL];
```

A **Block** describes a section of memory which will be the source and/or sink of the bytes transmitted; the section of memory described is a sequence of bytes (not necessarily word aligned). The **blockPointer** may be regarded as a base of a packed array of bytes; it selects a word such that a **startIndex** of zero would select the left byte of that word (i.e., bits 0 - 7). The selected block consists of the bytes **blockPointer^[*i*]** for *i* in **[startIndex..stopIndexPlusOne)**. Notice that a **Block** cannot describe more than 2^{16}-1 bytes or 2^{15}-1 words.

Note: Some of the operations of this section and the next may cause signals to be generated. If such a signal is **RESUME**d, transmission continues where it left off -- i.e., any changes made by the catch phrase to the **Block** record and/or to the input options (see below) are ignored. If, however, such a signal is **RETRY**'ed, the next byte of the stream sequence is transmitted to or from the byte specified by the current value of the **Block** record and/or input options, either of which might have been updated by the catch phrase. In no case is the stream sequence itself "backed up." That is, bytes previously received from input are not re-received, and bytes previously transmitted on output are not withdrawn.

The primary input operation is **Stream.GetBlock**.

```
Stream.GetBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block]
    RETURNS [bytesTransferred: CARDINAL, why: Stream.CompletionCode,
    sst: Stream.SubSequenceType];
```

Stream.CompletionCode: TYPE = {normal, endRecord, sstChange, endOfStream};

The parameter **block** describes the memory area into which the bytes will be placed. **GetBlock** does not return until the input is terminated. Its exact behavior, however, is controlled by a set of input options which may be set by the client using the following operation:

Stream.SetInputOptions: PROCEDURE [sH: Stream.Handle, options: Stream.InputOptions];

Stream.InputOptions: TYPE = RECORD [terminateOnEndPhysicalRecord, signalLongBlock, signalShortBlock, signalSSTChange, signalEndOfStream: BOOLEAN];

Stream.defaultInputOptions: Stream.InputOptions = [FALSE, FALSE, FALSE, FALSE, FALSE];

SetInputOptions controls exactly how **GetBlock** terminates and what signals it generates. Ordinarily (i.e., with the parameter **options** set to **defaultInputOptions**) the transmission will not terminate until the entire block of bytes is filled. However, under exceptional conditions described in *:A.4*, the transmission may terminate before the **block** is filled and may also result in a signal. In all cases the procedure will return the actual number of bytes transferred, a **CompletionCode** indicating the reason for termination, and the latest **SubSequenceType** encountered. The input operation may conveniently be restarted where it left off by first adding the result **bytesTransferred** to **block.startIndex** to update the record describing the block of bytes.

Two circumstances which *always* suspend the transmission of data before the **block** is filled are (a) the detection of a change in **SubSequenceType** and (b) the **endOfStream**. If the input option **signalSSTChange** is **FALSE** (the default case), then the procedure **GetBlock** terminates immediately and returns the number of bytes transferred, with **why** = **sstChange**, and **sst** set to the new value of the **SubSequenceType**. If the input option **signalSSTChange** is **TRUE**, then the signal

Stream.SSTChange: SIGNAL [sst: Stream.SubSequenceType, nextIndex: CARDINAL];

is generated. The parameter **sst** specifies the new **SubSequenceType**, and the parameter **nextIndex** specifies the byte index within the **block** where the first byte of the new subsequence will be placed. This signal may be resumed, and the effect is to continue the data transmission as though the change in **SubSequenceType** had not occurred (i.e., in the same block of bytes).

Caution: A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

Implementation of the end-of-stream feature is strictly transducer and filter specific, and optional. All transducers and filters *need not* implement this feature. Transducer and filter implementors may implement an end-of-stream mechansim using any protocol they desire. Typically, this feature would be implemented by exchanging subsequence types in some specific order. If implementors use this mechansim, then they must document the subsequence types reserved for this, so that clients do not inadvertantly use the same ones for their own protocol. When putting together a

pipeline from a transducer and filters, great care needs to be taken to preserve the end-of-stream feature through all the stream components. If the input option **signalEndOfStream** is **TRUE** and the stream component detects that the end-of-stream has occured, then the signal

```
Stream.EndOfStream: SIGNAL [nextIndex: CARDINAL];
```

is generated. The parameter **nextIndex** specifies the byte index immediately following the last byte of the stream sequence filled into a client's block.

Note: Stream component implementors may provide special procudure calls in order to actively cause a stream to be terminated.

The principal output operation is **Stream.PutBlock**.

```
Stream.PutBlock: PROCEDURE [sH: Stream.Handle, block: Stream.Block,
    endPhysicalRecord: BOOLEAN];
```

This operation is analogous to **Stream.GetBlock**. The parameter **block** describes the area of memory from which information is transmitted. This procedure returns only after the data has been accepted by the stream, at which time the client may reuse the **block**. If the client is ignoring physical record boundaries (the default case) then parameter **endPhysicalRecord** should be set to **FALSE**. Otherwise, see :*A.4*.

Note: Stream operations have the right to discard empty blocks, hence a **PutBlock** operation specifying a **block** of length zero may be a no-op.

A.2.2 Additional Data Transmission Operations

In addition to **GetBlock** and **PutBlock**, the following operations are provided:

```
Stream.GetByte: PROCEDURE [sH: Stream.Handle] RETURNS [byte: Stream.Byte];

Stream.GetChar: PROCEDURE [sH: Stream.Handle] RETURNS [char: CHARACTER];

Stream.GetWord: PROCEDURE [sH: Stream.Handle]
    RETURNS [word: Stream.Word];

Stream.Word: TYPE = [0..65535];
```

GetByte and **GetChar** operations get the next **Byte** or **CHARACTER** from the stream sequence and return it. They are equivalent to a call upon **Stream.GetBlock**, specifying a **Block** containing one byte. The **GetWord** operation gets the next **Word** from the stream sequence and returns it. It is equivalent to a call upon **Stream.GetBlock**, specifying a **Block** containing **AltoDefs.BytesPerWord** bytes. Input options are assumed to be **signalShortBlock**, **signalLongBlock**, and **endPhysicalRecord** = **FALSE**, and **signalEndOfStream** and **signalSSTChange** = **TRUE**. Thus, these operations may result in signal **SSTChange** or **EndOfStream**.

Note: When the **SIGNAL**s **SSTChange** or **EndOfStream** are generated, **nextIndex** should be equal to 0. In the case of **GetWord** if **nextIndex** = 1, the caller is responsible for processing the "half-word".

Stream.PutByte: PROCEDURE [sH: Stream.Handle, byte: Stream.Byte];

Stream.PutChar: PROCEDURE [sH: Stream.Handle, char: CHARACTER];

Stream.PutWord: PROCEDURE [sH: Stream.Handle, word: Stream.Word];

The **PutByte** and **PutChar** operations transmit the next **Byte** or **CHARACTER** to the medium. They are equivalent to a call on **Stream.PutBlock**, specifying a **Block** containing one byte. The **PutWord** operation transmits the next **Word** to the medium. This procedure is equivalent to call on **Stream.PutBlock**, specifying a **Block** containing **AltoDefs.BytesPerWord** bytes. These output operations specify **endPhysicalRecord** = **FALSE**.

Stream.SendNow: PROCEDURE [sH: Stream.Handle];

This operation flushes the stream sequence; it guarantees that all information previously output (by means of **PutBlock**, **PutByte**, **PutChar**, **PutWord** or **SetSST**) will actually be transmitted to the medium (perhaps asynchronously). This procedure is equivalent to a call on **Stream.PutBlock**, specifying a **Block** containing no bytes and **endPhysicalRecord** = **TRUE** (see ∶*A.4*). Client programs which are not concerned with physical record boundaries should nevertheless call **SendNow** at appropriate times to ensure that the bytes and changes in **SubSequenceType** have actually been sent and are not buffered internally within the stream, awaiting additional output operations.

Stream.SetSST: PROCEDURE [
sH: Stream.Handle, sst: Stream.SubSequenceType];

This operation causes all subsequent bytes to have the indicated **SubSequenceType**. Even if the subsequent sequence of bytes is null (i.e., a call on **SetSST** is immediately followed by another), the **SubSequenceType** change demanded by this call will still be available to the receiver of the stream sequence.

Note: **SubSequenceType**s are intended to be used to delineate different kinds of information flowing over the same stream (e.g. to identify control information, indicate end-of-file, etc.) The interpretation of a **SubSequenceType** value is a function of the particular stream.

Note: A **SetSST** operation specifying a **SubSequenceType** identical to the previous **SubSequenceType** is a no-op. Otherwise, **SetSST** *always* has the side effect of completing the current physical record, as explained in ∶*A.4*.

A.2.3 Attention Flags

The following operation causes an attention flag and an associated byte of data to be transmitted via the stream facility.

Stream.SendAttention: PROCEDURE [sH: Stream.Handle, byte:Stream.Byte];

Note that neither the attention flag nor the data byte occupy a byte in the stream sequence. They are out of band signals. Note also that an attention is not necessarily transmitted in sequence, but may bypass bytes and changes in **SubSequenceType** which were transmitted before it.

Note: This operation may have the side effect of completing the current physical record, as explained in *:A.4*. A client process will typically follow the **SendAttention** by a change in **SubSequenceType** or some recognizable pattern of bytes. This permits the recipient to identify where the **SendAttention** occurred.

Note: **byte** may be used by the client protocol to transmit other information regarding this attention.

The following operation awaits the arrival of an attention flag.

Stream.WaitForAttention: PROCEDURE [sH: Stream.Handle] RETURNS [Stream.Byte];

When an attention is received on the stream **sH**, this procedure returns the byte of data associated with the attention. It is the responsibility of the client program to determine the appropriate action to take. If more than one attention flag has been sent, these will be queued by the stream. Each return from a call on **WaitForAttention** corresponds to precisely one attention sent by **SendAttention.**

Note: This operation is usually executed by a different process from that operating upon the stream. It returns as soon as the attention is received, whether or not all of the bytes preceding it in the stream have been transferred.

A.2.4 Timeouts

Any of the operations of this section (except **SendAttention** and **WaitForAttention**) may fail to complete within a reasonable amount of time due to external conditions. For example, a stream operation on a Pup stream may fail because the process at the other end has terminated, aborted, or ceased to pay attention. In such a case the following signal is generated:

Stream.TimeOut: SIGNAL [nextIndex: CARDINAL];

The parameter of this signal indicates the position within the block of bytes where the next byte would be placed. This signal may be resumed.

Note: If this signal is **RETRY**'ed all previously received data may be lost. This is because it is likely that a stream component is performing internal bufferring (transferring data from its buffer into the client's block), and the action of **RETRY**ing the **SIGNAL** may not tell the component that it must refill the client's block. Even if the component was informed of this fact, it may have discarded data already transferred into the client's block from its internal buffer

Caution: A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

A.3 Creating and Deleting Streams

There are no general operations for creating streams. The reason for this is that the components of a stream -- i.e., pipelines, transducers, and filters -- must be able to take arbitrary parameters at the time they are created. It is not possible for the system to specify a general interface for their creation without either compromising the basic typesafeness of Mesa or constraining the flexibility and power of client-provided streams. Thus, the create function is implemented on a case-by-case basis, and clients must therefore refer to documentation for individual stream components for the correct interface for this operation. In this section, the general style is illustrated by means of a hypothetical example.

For example, if a utility package implements a transducer to a particular device, it is obligated to provide a means by which other clients can create instances of that transducer, use them, and later delete them. Suppose the name of the interface module providing this function is **DeviceStream**. Then it would provide the following operation:

```
DeviceStream.Create: PROCEDURE [ --optional parameters-- ]
    RETURNS [Stream.Handle, --optional other results--];
```

A client wishing to use the stream interface to access this device would thus call **DeviceStream.Create**, then use the **Stream.Handle** returned from it as a parameter to the stream operations of this chapter.

Similarly, a security package providing, say, an encryption facility might implement this by means of a filter for a stream. In this case, the interface might be called **EncryptionFilter**, and it would provide the following operation:

```
EncryptionFilter.Create: PROCEDURE [Stream.Handle, --optional other
    parameters-- ] RETURNS [Stream.Handle, --optional other results--];
```

The client could easily couple an instance of this filter with the transducer above. This is done by calling **EncryptionFilter.Create**, passing as a parameter the **Stream.Handle** returned from **DeviceStream.Create**. Then the **Stream.Handle** returned from **EncryptionFilter.Create** would be the one used in **GetBlock**, **PutBlock**, and the other operations of :*A.2*. The net effect would be stream components which, on input, read bytes from the device, decrypt them, and pass them to the client and which, on output, encrypt the bytes supplied by the client and write them on the device.

In general, creating a filter accepts one **Stream.Handle** as a parameter and returns another as its result. Thus, several filters, each implementing a simple transformation, may be concatenated together to implement a more interesting transformation on the stream. The parameter passed to each one is the result returned from the adjacent one. Such a concatenation, called a *pipeline*, is illustrated in Figure A.1.

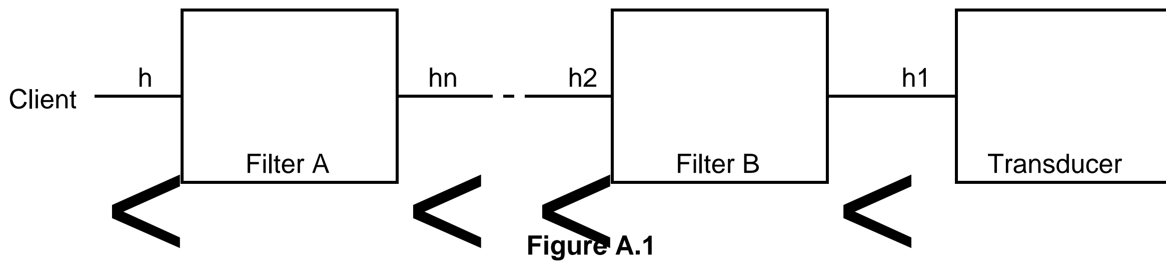

Figure A.1

This diagram illustrates how each **Stream.Handle** returned from a transducer or filter is passed as parameter to the next adjacent filter, and how the last one is used directly by the client. In particular, **h1** is returned from the procedure which creates **Transducer**; it is passed to the procedure which creates **Filter B**, returning **h2**. This is passed, in turn, to the next filter, and so on, until **hn** is returned and passed to **Filter A**. **Filter A** is the last one in the pipeline, and its **Stream.Handle**, **h**, is returned to the client.

Figure A.2 shows the flow of data through the pipeline and the use of the various **Stream.Handle**s as a result of a client call on **Stream.GetBlock** (calls on other data transmission operations are analogous).

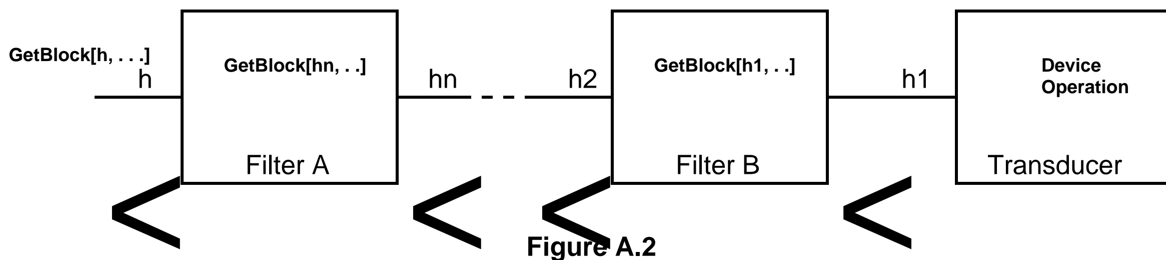

Figure A.2

Here, the client calls **Stream.GetBlock[h, . . .]**, which is transformed by the stream interface into an appropriate call on **Filter A**. **Filter A**, in turn, calls **Stream.GetBlock[hn, . . .]**, which is passed to the next filter in the pipeline, and so on, until eventually a call is made on **Stream.GetBlock[h2, . . .]**. This is transformed into a call on **Filter B**, which then calls **Stream.GetBlock[h1, . . .]**, to invoke **Transducer**, which actually operates the device.

Note that the only difference between a transducer and a filter is that a transducer interfaces to some device, while a filter interfaces to another stream -- i.e., indirectly to another filter or transducer.

Note also that the client can construct a pipeline "manually," by tediously assembling the various components, instantiating each of them, and binding them together. However, a pipeline can also be presented as an integrated package, already assembled. For example, the two components described above may have been assembled into a pipeline called **EncryptingDeviceStream**. This pipeline might then provide the following two operations, which clients can call to create and delete an instance of this pipeline:

> **EncryptingDeviceStream.Create: PROCEDURE [--optional parameters--]**
> **RETURNS [Stream.Handle, --optional other results--];**

> **EncryptingDeviceStream.Delete: PROCEDURE [Stream.Handle, --optional other**
> **parameters--] RETURNS [--optional results--];**

The client of such a stream would merely invoke these procedures to create and destroy the stream without having to bother about finding and putting together the individual components.

A.4 Control over Physical Record Characteristics

Most of the time, the client will not wish to know about how the data comprising a stream sequence is divided into physical records for recording or transmission. For some applications, however, this is of vital importance. The stream facility has been designed so that the details of the physical encoding can be ignored when desired, or completely known and controlled when that is necessary. On output, complete control of the placement of bytes in physical records can be achieved for most media. On input, complete information is available about how the bytes were arranged in physical records.

Note: These facilities to control the placement of bytes on physical records are *not* meant to be used as a means of transmitting information. In particular, a transducer might suppress or generate empty physical records and will necessarily partition oversize "physical" records into smaller ones. Any filter may rearrange (or completely obliterate) physical record boundaries. Documentation for the individual transducer or filter and for the individual transmission or storage medium should be consulted for full details.

The output and input cases will be treated separately. On output, bytes will be placed in turn into the same physical record until one of the following events occurs:

1) The **SendNow** procedure is called; it has the side effect of causing the current record to be sent. The next byte output will begin a new physical record. Thus, this is the main mechanism for controlling physical record size on output.

2) A **PutBlock** procedure is called with an **endPhysicalRecord** parameter of **TRUE**. After the transmission of this block of bytes, the current physical record is ended. If, at this point, the physical record is at its maximum size (see (4) below), an empty record will *not* be transferred.

3) A **SetSST** procedure has been called. The first byte of a new subsequence always begins a new record and has the new **SubSequenceType**. This may cause the previous record to be sent.

4) Enough bytes have been output to fill the physically maximal record. At this point the record will be written and a new record started. This maximum number is a function of the medium being written, hence documentation concerning the medium must be consulted to determine this value.

5) Some other device-dependent event, such as a timeout, occurs. In this case, a buffer may be flushed automatically. Details are documented with individual transducers.

On input, bytes will be placed in turn into the record until one of the following events occurs:

1) The end of the physical record is reached, the block of bytes described in the **Block** record is not exhausted, and either of the input options **endPhysicalRecord** or **signalLongBlock** is **TRUE.**

If **signalLongBlock** is **TRUE**, the following signal is generated:

Stream.LongBlock: SIGNAL [nextIndex: CARDINAL];

This signal indicates in **nextIndex** the position within the block of bytes where the next byte will be placed. If it is resumed, transmission continues as if it had not been generated.

Caution: A catch phrase for this signal must not attempt any other stream operations using the same **Stream.Handle**, for this will corrupt the internal state information maintained for the stream.

If **endPhysicalRecord** is **TRUE**, the input is terminated, indicating the number of bytes transferred and **why** = **endPhysicalRecord**. This applies whether or not a signal was generated.

2) The end of the physical record is reached at the same time that the block of bytes is exhausted. In this case, no signal is generated. If the input option **endPhysicalRecord** is **TRUE**, then **why** is set to **endPhysicalRecord**; otherwise, it is set to **normal**.

3) The block of bytes is exhausted, the end of the physical record has not been reached, and the input option **signalShortBlock** has the value **TRUE**. At this time the input is terminated (without losing the subsequent bytes of the physical record, which are still available for reading by subsequent **GetBlock**), and the signal **Stream.ShortBlock** is generated.

Stream.ShortBlock: ERROR;

This signal may not be resumed.

The easiest approach is usually to establish a **Block** longer than the longest expected physical record and specify input options **signalLongBlock** = **FALSE**, **signalShortBlock** = **TRUE** and **endPhysicalRecord** = **TRUE.** At this point the transmission will cease with the entire contents of the physical record in the block of bytes, and the number of bytes transmitted will be returned as the result of the **GetBlock** procedure. In this way a signal will be generated only under unusual circumstances.

A.5 Transducers, Filters, and Pipelines

The stream package is designed so that clients can implement their own stream components -- i.e., their own transducers, filters, and pipelines. The implementor of one of these has three different obligations to fulfill. First, he must design an interface (i.e., Mesa **DEFINITIONS** module) in the style described in :*A.3*, by which his clients create instances of that stream component. Such an interface (together with its accompanying implementation modules) is called a *stream component manager*. Second, he must provide a functional specification describing this interface and the detailed behavior of the stream component, including any specific signals, errors, parameters, etc., which it defines. Third, he must implement the actual component, if it is a filter or transducer. (Pipelines are assumed to be composed of previously implemented components which already have their own component managers and documentation.)

This section describes the standards, data types, and operations to be used in defining a new stream component. In :*A.5.1*, the precise interface which each instance of each filter or transducer must provide is specified. In :*A.5.2*, a typical method for implementing a filter or transducer manager is outlined.

A.5.1 Representing Filters and Transducers

At run time, a filter or transducer is represented by six procedures which execute in a common context to provide the data transmission operations of the that filter or transducer. Descriptors for these procedures are stored in a record defined by the stream package, pointed to by a **Stream.Handle**.

```
Stream.Handle: TYPE = POINTER TO Stream.Object;

Stream.Object: TYPE = RECORD [
    options: Stream.InputOptions,
    get: Stream.GetProcedure,
    put: Stream.PutProcedure,
    setSST: Stream.SetSSTProcedure,
    sendAttention: Stream.SendAttentionProcedure,
    waitAttention: Stream.WaitAttentionProcedure,
    delete: Stream.DeleteProcedure];
```

A client call on a stream operation is normally converted by the stream package into a call on the appropriate procedure named in the **Stream.Object** pointed to by the **Stream.Handle** parameter of that operation. Thus, it is the responsibility of the implementor of each filter and transducer to exactly satisfy the specifications of the stream package. The stream package assists in this task by utilizing the Mesa typechecking machinery and by defining the uniform interface encapsulated by **Stream.Object**.

In this section, the meanings of the fields of **Stream.Object** are enumerated.

The **options** field specifies the currently valid input options for the stream.

```
options: Stream.InputOptions,
```

This field is set by **Stream.SetInputOptions** and its current value is passed as a parameter to the **get** procedure described below. Implementors of filters and transducers need not be concerned with maintaining or inspecting this field.

The **get** field specifies the input procedure of the transducer or filter.

```
get: Stream.GetProcedure,

Stream.GetProcedure: TYPE = PROCEDURE [
    sH: Stream.Handle, block: Stream.Block, options: Stream.InputOptions]
    RETURNS [bytesTransferred: CARDINAL,
    why: Stream.CompletionCode, sst: Stream.SubSequenceType];
```

This procedure is called by **GetBlock, GetByte, GetChar,** and **GetWord**. It must implement the semantics of **GetBlock** as described in *:A.2.1* and *:A.4*. In particular, it must terminate according to the specifications of that section and must generate the signals **SSTChange**, **LongBlock**, **ShortBlock**, **EndOfStream**, and **TimeOut** (*:A.2.4*) as required.

Note: In a filter, the body of a **GetProcedure** will typically contain one or more calls on **GetBlock, GetByte, GetChar,** or **GetWord** with a **Stream.Handle** parameter pointing to the next stream component in the pipeline (i.e., the parameter passed at the time this filter was created). In a transducer, the body of a **GetProcedure** will typically have calls on input operations for the specific device being supported.

The **put** field specifies the output procedure provided by the filter or transducer.

```
put: Stream.PutProcedure,

Stream.PutProcedure: TYPE = PROCEDURE [
    sH: Stream.Handle, block: Stream.Block, endPhysicalRecord: BOOLEAN];
```

This procedure is called by **PutBlock**, **PutByte**, **PutChar**, **PutWord**, and **SendNow**. It must implement the semantics of **PutBlock** as described in *:A.2.1* and *:A.4*. In particular, it must regard the parameter **endPhysicalRecord = TRUE** as an indication to flush any output buffers and actually initiate the physical transmission of information. It may suppress output requests specifying a block of no bytes provided that there is no previous output, change in **SubSequenceType**, or attention flag still waiting to be sent. This procedure may generate the signal **TimeOut** if necessary.

Note: In a filter, the body of a **PutProcedure** will typically contain one or more calls on **PutBlock**, **PutByte**, **PutChar**, **PutWord**, or **SendNow** with a **Stream.Handle** parameter pointing to the next stream component in the pipeline (i.e., the parameter passed at the time this filter was created). In a transducer, the body of a **PutProcedure** will typically have calls on output operations for the specific device being supported.

The **setSST** field specifies the procedure to change the current **SubSequenceType** of the output side of the filter or transducer.

```
setSST: Stream.SetSSTProcedure,

Stream.SetSSTProcedure: TYPE = PROCEDURE [
    sH: Stream.Handle, sst: Stream.SubSequenceType];
```

This procedure is called by **Stream.SetSST** and must conform to the semantics of that operation as described in *:A.2.2*. In particular, it should be a no-op if the new **SubSequenceType** is the same as the old one. Otherwise, it should have the effect of completing the current physical record (as if a call on **Stream.SendNow** had been made immediately before).

Note: A call on **setSST** may have the effect of changing the internal state of the stream component, or in the case of a filter, it may result in a call to **SetSST** to the next stream component in the pipeline, or both.

The **sendAttention** and **waitAttention** fields specify the two procedures implementing the sending of and waiting for attention flags in the transducer or filter.

```
sendAttention: Stream.SendAttentionProcedure,
waitAttention: Stream.WaitAttentionProcedure,
```

```
Stream.SendAttentionProcedure: TYPE = PROCEDURE [
    sH: Stream.Handle, byte: Stream.Byte];

Stream.WaitAttentionProcedure: TYPE = PROCEDURE [sH: Stream.Handle]
    RETURNS [Byte: Stream.Byte];
```

These two procedures will be called by **Stream.SendAttention** and **Stream.WaitForAttention**, respectively, and they must conform to the semantics of those operations as specified in *:A.2.3*.

Note: In a filter, it is not always necessary to implement these two procedures (or any others of this section) if all they do is call the corresponding operation in the next stream component in the pipeline. Instead, it is satisfactory to copy the procedure descriptor from one **Stream.Object** to the other. This has the effect of making the call pass "straight through" the filter with no overhead.

The **delete** field specifies a procedure to be used during deletion of a filter or transducer.

```
delete: Stream.DeleteProcedure,

Stream.DeleteProcedure: TYPE = PROCEDURE [sH: Stream.Handle];
```

This procedure provides a convenient way of deleting an instance of a transducer or filter once it is no longer needed; an example of its use is given in the next section. The parameter **sH** is provided for convenience and may be used for whatever purpose is useful.

A.5.2 Stream Component Managers

Implementors of stream components may create instances of them by whatever means is most appropriate to their requirements. A particular filter or transducer may, for example, consist of one module, a collection of modules, a local frame used in conjuction with the Mesa **PORT** facility, or some other construct. Moreover, it may exist on a given machine in only one or a limited number of copies which are regarded as "serially reusable" resources (for example, a transducer to a particular device, of which there is only one or a limited number on a machine), or it may exist in as many copies as appropriate. It is the responsibility of the stream component manager to create (or control access to) instances of that stream component, as appropriate. When access is granted, the component manager must also provide a pointer to a **Stream.Object** containing procedure descriptors for that component.

The typical way of implementing a component is as a single module which is instantiated at run-time by the Mesa **NEW** statement. Declared within this module would be the procedures of the component plus a **Stream.Object** containing their procedure descriptors. The component manager executes **NEW** to create a new instance of the stream, followed by **START** to initialize it (possibly passing parameters) and to obtain a pointer to its **Stream.Object**.

The component manager deletes instances of stream components by calling **FrameDefs.UnNew** or **FrameDefs.SelfDestruct**.

FrameDefs.UnNew (the inverse of **NEW**) takes a **GlobalFrameHandle** as its parameter. Component managers can determine the value of this parameter by calling **FrameDefs.GlobalFrame**, which takes a procedure descriptor of a procedure residing in the component to be deleted as a parameter, and returns a pointer to its global frame. **FrameDefs.UnNew** frees space occupied by the component's global frame.

Caution: The client must ensure that there are no outstanding references to the component module being deleted -- i.e., no procedure descriptors or pointers which might be used. In addition, any process waiting for attentions (i. e., a process which has called but not returned from **WaitForAttention**) must be aborted and allowed to exit from the module. Failure to observe this caution will result in unpredictable effects. In particular, **FrameDefs.UnNew** *must be called from outside* the module being deleted.

FrameDefs.SelfDestruct sets the internal state of the process so that the module in which the calling procedure is declared will be unnewed *after* the calling procedure returns to its caller.

This operation has the effect of placing a "self-destruct" mechanism in the module which will take effect after the calling process exits from it. Thus, it is a means of deleting the stream component from *within* that component.

Caution: As above, the client must ensure that there are no references to the module currently in use by any process.

The typical use of **FrameDefs.SelfDestruct** will be from a procedure named in the **delete** entry of the **Stream.Object**. The component manager will call **h.delete[h]** (where **h** is a **Stream.Handle**). This procedure will perform the necessary finalization, such as flushing buffers, closing files or connections, releasing storage and resources, etc. It will then call **FrameDefs.SelfDestruct** and finally return to the component manager. After this return, the module representing this instance of the stream component will be deleted and space occupied by the stream's global frame will be freed.