NAME

overview -- overview of Pup library routines

SYNOPSIS

cc [flags] filename [...] -lpup

DESCRIPTION

Stanford's Pup library is intended to be a system-independent library for dealing with the Pup environment. Although anyone intending to use these routines should be familiar with the Pup protocols and data structures, a brief description of the protocol layering follows.

Transport Layer — The lowest level of the library is technically not part of the Pup protocol system; rather, it provides a means of moving uninterpreted packets from one host to another over a single data link. Examples of such links are serial lines, Arpanet connections (as viewed from the Pup world), and Ethernet cables. The local operating system provides some sort of communication with the interface hardware, and the transport layer uses this to provide a system-independent interface to the network. [Currently, only an Ethernet layer is implemented, supporting both 3mb and 10mb Ethernets.]

Pup Packet Layer — The first true Pup level is concerned with moving Pup packets from host to host [in the Pup world, all routing is done by gateways, so Pup transfers can be thought of as an end-to-end service.] This layer takes a data buffer and some connection information, and choses the proper encapsulation and transport medium. In the Stanford Pup library, there is a data structure called a PupChan ("Pup Channel") which hides the information about the underlying transport mechanism. Actions at this level include writing packets to, and reading packets from, a PupChan.

The following layers are parallel (more or less) in level:

BSP (Byte Stream Protocol) — BSP is used to create reliable, sequenced, full-duplex streams of bytes between two processes. It is used to implement Telnet, FTP, and Mail protocols.

EFTP — EFTP is very simple file transfer protocol used primarily for sending Press-format files to be printed, and for boot-loading small systems. It is easier to implement than a BSP-based FTP, and thus is also useful for communicating with stone-age systems such as Sail.

Miscellaneous Services — This is a collection of simple operations based on a single packet request/ack scheme. It supports such services as time-of-day, user authentication, mail-checking, etc. A similar system is used to implement internetwork routing.

SEE ALSO

Xerox Pup documentation

BUGS

Would be obsolete if it didn't work so well.

NAME

atoo -- convert ASCII to octal numbers

SYNOPSIS

long atoo(nptr)
char *nptr;

cc files... -lpup

DESCRIPTION

This function converts a string pointed to by *nptr* to octal representation. The first unrecognized character ends the string.

Atoo recognizes an optional string of tabs and spaces, then a string of octal digits.

SEE ALSO

scanf(3), atof(3)

BUGS

There are no provisions for overflow.

NAME

bmove — block move a buffer

SYNOPSIS

bmove(src, dst, len)
char *src;
char *dst;
int len;

cc files ... -lpup

DESCRIPTION

Bmove is used to copy a block of data from one place to another. It is coded in machine language on the Vax as one instruction, thus becoming quite efficient. A c language version exists for transportation to other machines, although any machine with a block-move instruction warrants machine-coding for efficiency.

AUTHOR

Jeff Mogul

SEE ALSO

swab(3), bstring(3)

BUGS

The Vax machine-code version of *bmove* only works on buffers up to 64k bytes long. This is sufficient for most applications; if there is a chance that the buffer might be longer, then *bcopy* (see *bstring*(3)) must be used. However, *bmove* is faster for small buffers, by perhaps 20%.

NAME

byteorder — discussion of byte-ordering and the Pup package

SYNOPSIS

#include <pup/puplib.h>

cc files ... -lpup

DESCRIPTION

Different machines order small objects within larger ones in different ways. The PDP-11 and the VAX-11, for example, order bytes so that the least significant byte in a 16-bit word has the lower address (i.e., has a lower index in an array of bytes), and they order words within longwords (32-bit integers) so that the least significant word in the longword has the lower address. On the other hand, many other machines (Altos, DEC-10s, MC68000s (Suns), and IBM equipment) use the opposite order; the least significant byte in a word has an address higher than the most significant byte, and similarly for words within longwords.

Since the Pup protocols were developed at Xerox, using Altos and PDP-10ish machines, the packets normally are arranged by the ordering natural on those machines. This ordering will thus be referred to as the "Standard Byte Ordering", or SBO, while the PDP-11 ordering will be referred to as the "Non-Standard Byte Ordering", or NSBO.

The Pup package is designed to compile and run on either sort of machine (actually, it would almost certainly not work on a machine whose data objects were not multiples of 8 bits long.) This is achieved with some pain, but most of the work has been hidden for you.

The Pup library and the related header files are conditionallized on the type of machine you are using; you must define to the C compiler a symbol indicating which machine you are compiling for; e.g.,

cc -DVAX foo.c -lpup

will work on the VAX (note that the Pup library must be recompiled for each machine; however, each C compiler will presumably look for object libraries in the right place.) The allowable machine names are: VAX, PDP_11, and MC68000. You may also simply define PUP_SBO or PUP_NSBO to indicate the kind of ordering in use. If one of these names is not defined, then a VAX is assumed.

Packets contain arrays of bytes; therefore, the following points arise:

words	reading/writing a short word in a buffer involves byteswapping the two bytes iff the host machine is NSBO.
long words	read/writing a long word involves swapping the low and high short words (and the bytes withing them) iff the host machine is NSBO.
byte arrays	Are order-independent.

If you follow the following rules, you will not go far wrong. Always:

retrieve longs from buffers using getlong()

make longs for insertion into buffers using makelong()

retrieve shorts from buffers using getshort()

make shorts for insertion into buffers using makeshort()

Shorts, longs, and character strings should always begin on a word boundary, that is, at an even byte address in a Pup data buffer. This is because some systems (such as Altos) often have hardware memory addressing restrictions. Getshort(), makeshort(), getlong(), and makelong() are described

in *pupmisc(9)*.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz, Bill Nowicki

SEE ALSO

pupmisc(9), pupstring(9)

DIAGNOSTICS

eHpl !'I mrtpaep dniisedt ehA tl!o

BUGS

BCPL strings violate the rule about strings beginning on a word boundary, in that their stringy parts sort of don't. *GetBCPLstring* and *PutBCPLstring* (see *pupstring*(9)) take care of this, though.

NAME

checksum -- compute Pup-style checksum

SYNOPSIS

```
unsigned short checksum(buffer,len)
unsigned short *buffer;
int len;

cc files ... -lpup
```

DESCRIPTION

Checksum calculates a Pup-style checksum over a buffer. The buffer is considered to consist of unsigned shorts (16 bits each.) The length parameter is, however, the length of the buffer in bytes (since this is more usually available.) Odd lengths are rounded down.

The checksum algorithm is: start with 0, and repeatedly (add in the next buffer word using one's complement arithmetic, then do a 16-bit rotate on the result.) If the final result is "= 0", replace it with 0.

This routine is coded both in C, and in machine code. On the VAX, the machine code version takes about 1 millisecond for a maximal Pup packet.

AUTHOR

Jeff Mogul

BUGS

Someone should write a routine that recalculates the checksum for a modified buffer.

NAME

eftp – Pup EFTP package

SYNOPSIS

```
#include <pup/eftp.h>
#include <pup/pupstatus.h>

EfRecOpen(Efchan,Sender,timeout,bswap)
struct EftpChan *Efchan;
struct Port *Sender;
int timeout;
int bswap;

EfRecOpenS(Efchan,Sender,timeout,bswap,socket)
Socket socket;

EfRecPckt(Efchan,buf,buflen)
char *buf;
int *buflen;

EfRecEnd(Efchan)

EfSendOpen(Efchan,Dest,timeout,bswap)
struct Port *Dest;

EfSendOpenS(Efchan,Dest,timeout,bswap,socket)

EfSendPckt(Efchan,buf,buflen)

EfSendEnd(Efchan)

EfSendClose(Efchan)

EftpRead(Efchan,buf,buflen,rbuflen)
int *rbuflen;

EftpWrite(Efchan,buf,buflen)

cc files .... -lpup
```

DESCRIPTION

The routines described in this manual entry constitute a fairly complete set of functions for implementing EFTP transfers. They are based on the Pup Library, and thus can be used without regard to underlying transport medium.

All of the routines refer to structure called an *EftpChan*, which describes the state of an EFTP transfer. A user should do nothing to an EftpChan except allocate space for it, and pass it to the routines in the package. Space should not be deallocated until the EftpChan has been properly closed. An EFTP transfer is unidirectional; one is either sending or receiving a file, but not both.

There are two levels of data transfer routines. The higher level, using *EftpWrite*, involves sending an entire file as one buffer (i.e., the calling program need not worry about packet disassembly). This works well on a virtual memory system, but can exceed the memory capacity of a smaller system, so the packet-level routine *EfSendPckt* is provided as well. For receive transfers, one may use either *EftpRead* or *EfRecPckt*; EftpRead provides a slightly simpler interface than EfRecPckt.

These three routines provide reliable transport of single packets, but the user program must assemble or disassemble the file accordingly.

An EFTP file read transfer follows the following general sequence: the EftpChan is created using *EfRecOpen*, the file is received using EftpRead or EfRecPekt, and the EftpChan is closed using *EfRecEnd*.

An EFTP file write transfer follows the following general sequence: the EftpChan is created using *EfSendOpen*, the file is sent using EftpWrite or EfSendPekt, *EfSendEnd* is used to indicate that the end of file has been reached, and the EftpChan is closed using *EfSendClose*.

DETAILS OF ROUTINES

These routines return OK on success unless noted otherwise.

EfRecOpen — This opens an EFTP channel in preparation for receiving a file; the *Sender* Port structure is used to indicate the address of the desired sender. If this has both net and host fields set to zero, then the first incoming file will be received, idependent of the sending host. The *timeout* parameter is the channel timeout in seconds; if *bswap* is true, then bytes will be taken from the net swapped from Network Standard byte order if EftpRead is used. (See *byteorder*(9)). This routine can return NOCHAN or NOROUTE.

EfRecOpenS — This is exactly the same as EfRecOpen, except that it allows you to specify the socket to receive on (otherwise, the standard EFTP socket is used.)

EfRecPekt — This routine is used to received a packet from the net. The returned buffer can be up to MAXPUPDATALEN bytes long; the length is returned in *buflen*. Byte-order considerations are ignored. EFTP_ENDOFFILE is returned upon sucessfully receiving the end of the file. The possible failure codes are TIMEOUT, EFTP_RESTART, EFTP_OUTOFSYNCH, EFTP_ERROR, or EFTP_ABORT. On EFTP_ABORT, the global *EftpErrMsg* will contain a human-readable error message, and the global integer *EftpAbortCode* will contain the protocol-defined abort code. An EFTP_RESTART means that the packet just received is really the first of the file, which is being retransmitted; the caller of this routine should move the returned buffer to the appropriate place and reset its various pointers.

EfRecEnd — This releases resources used in a completed EFTP read. It returns EFTP_ERROR if the EftpChan wasn't open for receiving.

EfSendOpen — This routine is used to create an EftpChan for sending a file. The *Dest* Port structure is the address to which the file will be sent; the timeout and bswap parameters are as described before. On failure, NOCHAN and NOROUTE can be returned.

EfSendOpenS — This is exactly the same as EfSendOpen, except that it allows you to specify the socket to send to (otherwise, the standard EFTP socket is used.)

EfSendPekt — This routine is used to send a data packet; the maximum data length that it will send is EFTP_MAC_PACKET bytes. On failure, it can return TIMEOUT, EFTP_BADACK, EFTP_ERROR, or EFTP_ABORT; upon an EFTP_ABORT, EftpErrMsg and EftpAbortCode are set as described before.

EfSendEnd — This is used to indicate to the receiver that the end of file has been reached. No data is sent. The possible returns are the same as with

NAME

(enarp) en10mbpuparp, ennoarp - Address Resolution Protocol (ARP) routines

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupstatus.h>

en10mbpuparp(PupHost, HardwareHost, fid, devparams)
Host *PupHost;
char *HardwareHost;
int fid;
union DevParams *devparams;

ennoarp(PupHost, HardwareHost, fid, devparams)

cc files ... -lpup
```

DESCRIPTION

These routines provide a variety of Address Resolution Protocols (ARPs) for use on ethernets. An ARP is used when the formats of the software host address and the hardware host address are so different that there is no simple mapping between them.

En10mbpuparp is used for resolving Pup addresses on the 10mb ethernet. *Ennoarp* is used when there *is* a simple mapping; that is, when hardware addresses are 8-bit values, i.e., on the 3mb ethernet.

The parameters to all routines in this group are the same, so that they can be used by generic code. *PupHost* and *HardwareHost* are the addresses of buffers for Pup and hardware addresses, respectively. *Fid* is the Unix file id on which the ethernet device is open (these routines may only be used with an already-open device.) *Devparams* is the address of a device parameters structure returned by *enopen*(9).

These routines provide three related operations:

What's my Pup Host Number?	If *HardwareHost* is NULL, then the ARP routine returns the local Pup host number in *PupHost*.
Pup Host to Hardware Host	Given a Pup host number in the buffer refered to by *PupHost*, returns the associate hardware host address in the *HardwareHost* buffer. (*HardwareHost* must be non-NULL.)
Hardware Broadcast Address	As above; if the buffer refered to by *PupHost* is zero, returns the hardware broadcast address.

AUTHOR

Jeff Mogul

SEE ALSO

enopen(9), RFC826 for a more flexible ARP.

DIAGNOSTICS

Returns OK if it worked, or ARPFAILURE if it didn't.

BUGS

It would be nice if there were a reverse mapping function (Harware host number to Pup host number).

En10mbpuparp is not yet implemented!

EfSendPckt.

EfSendClose — This releases resources used in a completed EFTP write. It returns EFTP_ERROR if the EftpChan wasn't open for sending.

EftpRead — This is used to read a packet from the net, but it has a cleaner interface than EfRecPckt. The parameter *buflen* is used to indicate the maximum allowable size of the returned buffer; *rbuflen* is used to return the number of bytes read (there is no indication of truncated buffers!) At end of file, EFTP_ENDOFFILE is returned. Possible failure codes are TIMEOUT, EFTP_OUTOFSYNCH, and EFTP_ERROR, which are fatal, EFTP_ABORT, which is fatal but which returns information in EftpErrMsg and EftpAbortCode. Finally, if EFTP_WAIT is returned, the transfer has failed but may be restarted after waiting for a period given (in seconds) by the global *EftpWaitTime*.

EftpWrite — This routine writes a buffer of arbitrary length over an EFTP channel. It can be called more than once; the final call must be followed by a call to EfSendEnd. The possible failure returns are TIMEOUT, EFTP_ERROR, EFTP_ABORT, EFTP_WAIT, and EFTP_RESTART. If EFTP_ABORT is returned, appropriate information is available in EftpErrMsg and EftpAbortCode. If EFTP_WAIT is returned, the caller should wait for several seconds, then start from scratch. If EFTP_RESTART is returned, the caller should restart from scratch but does not have to wait.

SEE ALSO

eftp(1), pupport(9)

Note that the eftp command does not use this implementation of EFTP.

DIAGNOSTICS

The possible values for EftpAbortCode are:

EFTPA_EXTSENDER	External Sender Abort
EFTPA_EXTRECEIVER	External Receiver Abort
EFTPA_RECBUSY	Receiver Busy Abort
EFTPA_OUTOFSYNCH	Out of Synch Abort
EFTPA_MISC	Misc Abort
EFTPA_LONGWAIT	Long Wait Abort
EFTPA_MEDWAIT	Medium Wait Abort
EFTPA_SUSPREQ	Suspend Request Abort

BUGS

A routine to send an EftpAbort is needed.

EftpRead might be redone to handle packet assembly.

A function is needed to determine the actual sender of a file if the Sender was not specified when calling EfRecOpen.

It should be easier to override the assumption that the EFTP receiver is always on socket 020; this would be useful for bootstrap loaders.

NAME

(enetfilter) ensetfilt, efinit, efwdinsert, efl ginsert, efchinsert, efAND — build ethernet filters

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupstatus.h>

ensetfilt(fid, enfilt)
int fid;
struct enetfilter *enfilt;

efinit(devparams, enfilt, priority, packettype)
union DevParams *devparams;
unsigned char priority;
unsigned short packettype;

efwdinsert(devparams, enfilt, offset, value)
int offset;
unsigned short value;

efl ginsert(devparams, enfilt, offset, value)
unsigned long value;

efchinsert(devparams, enfilt, offset, value)
unsigned char value;

efAND(enfilt)

cc files ... -lpup
```

DESCRIPTION

This is a collection of routines to make building ethernet packet filters (see *enet(4)*) relatively painless.

ensetfilt takes an enetfilter structure and makes it the applicable filter for the ethernet device open on *fid*, using device parameters *devparams*, both of which should have been obtained from *enopen*(9).

Efinit takes an enetfilter structure and initializes it to an empty filter, with the given priority. (The filter is "empty" except that it specifies which *packettype* is to be accepted. If *packettype* is zero, then all packet types will be accepted.)

Efwdinsert, *efl ginsert*, and *efchinsert* add equality tests for unsigned shorts, longs, and chars to a filter. The *offset* parameter is in units of bytes, starting from the data part of an ethernet packet. If you want to filter on ethernet packet header fields, you must use a negative offset.

EfAND adds a logical conjunction operation to the filter; if you put n terms into a filter, you should put $n - 1$ ANDs in afterwards to result in one boolean value for the lot of them.

Filters have maximum sizes; the current maximum is big enough for sane users, and the bigger the filter, the slower things go.

AUTHORS

Jeff Mogul

SEE ALSO

enopen(9), pupsetfilter(9), pupsetdfilt(9), enet(4)

DIAGNOSTICS

Returns OK if things work. Returns FILTERTOOBIG if the next operation would make the filter structure overflow.

BUGS

It might be nice to have other logical operations available.

NAME

enflush – flush an ethernet input file

SYNOPSIS

```
#include <pup/pupstatus.h>

enflush(fid)
int fid;

cc files ... -lpup
```

DESCRIPTION

Enflush is used to remove any packets that may be available to be read from the queue for an ethernet minor device. The argument should be a Unix file id of an ethernet device.

To avoid an infinite loop, the filter set for the ethernet device should reject all packets, so that no new packets arrive while the queue is being flushed. The filter that is set when an ethernet device is opened is such a filter.

The *enflush* routine sets the timeout delay for the ethernet device so that it returns immediately if no input is available; the routine does not restore the previous timeout delay. This may be considered a bug by some, although this routine is usually called just once, right after the file is opened.

AUTHOR

Jeff Mogul

SEE ALSO

enopen(9), enetfilter(9)

DIAGNOSTICS

None; this routine always returns OK.

NAME

engethost – determine Pup host number of ethernet interface

SYNOPSIS

```
engethost(fid, ARP)
int fid;
in (*ARP)();

cc files ... -lpup
```

DESCRIPTION

Engethost is passed the Unix file id (as returned by *enopen*(9)), of an ethernet file, and a pointer to the appropriate address resolution protocol (ARP) routine (see *enarp*(9)), and returns the Pup host number of the local machine, as found on the ethernet interface which is open on the given file id.

Note that any given machine may be connected to several networks, and may have different host numbers on each network.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

enopen(9)

BUGS

This is a nasty layer-crossing. We could use a routine to return the actual hardware address of the interface, simpler to use than the ENIOCDEVP ioctl (see *enet*(4).)

NAME

enopen -- open an ethernet file

SYNOPSIS

#include <pup/puplib.h>

enopen(devname, devparams)
char *devname;
union DevParams *devparams;

cc files ... -lpup

DESCRIPTION

Enopen attempts to open one of the ethernet minor devices available on the specified interface. E.g., if *devname* is "/dev/enetA", then it will try to open "/dev/enetA0", "/dev/enetA1", etc. It then sets a packet filter (see *enet(4)*) for the device which ignores all packets, flushes any pending input packets, and returns the Unix file id of the ethernet device. It also returns a device parameters structure in *devparams*, for use with other Pup library routines at this level. The device parameters include such things as hardware address formats, addresses, maximum packet size, etc.

There is no corresponding *enclose* routine; use the normal *close(2)* system call.

AUTHORS

John Seamons, Jeff Mogul

SEE ALSO

close(9), enet(4)

DIAGNOSTICS

Returns -1 if no ethernet minor devices are free.

NAME

enread -- read a packet from an ethernet file

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/puppacket.h>
#include <pup/pupstatus.h>

enread(fid, devparams, packet, filter, timeout)
int fid;
union DevParams *devparams;
struct EnPacket *packet;
struct enetfilter *filter;
int timeout;

enreadquick(fid, devparams, packet)

cc files ... -lpup
```

DESCRIPTION

Enread is used to read one packet from an ethernet minor device. The *fid* parameter is the Unix file id of the device to read from, *devparams* is the device parameters structure returned by *enopen*(9), *filter* is an ethernet packet filter (see *enet(4)*) to be set for the device, and *timeout* is the time (in clock ticks, 1/60 of a second) to wait before returning empty-handed.

Packet is the address of the *Pup* field of a *NetPacket* structure where the received packet will be put. The caller must allocated space for the entire netpacket, but must pass the address of the data part of the packet; this makes it possible for the caller to use this routine without knowing how large the ethernet header is. For example:

```
struct NetPacket pbuf;

...

status = enread(fid, &devparams, &(pbuf.Pup), &filter, timeout);
```

If the value NULL is passed for the filter address parameter, no new filter is set. This means that whatever previously set filter exists will be used.

Enreadquick works the same as *enread*, except that it does not take *filter* or *timeout* parameters; if these are already set, then the cost of passing them can be avoided.

AUTHORS

John Seamons, Jeff Mogul

SEE ALSO

enopen(9), enwrite(9)

DIAGNOSTICS

Returns TIMEOUT if the read fails (for any reason, including such things as file not open, etc.), otherwise returns OK.

NAME

ensetbacklog – set ethernet input queue backlog

SYNOPSIS

ensetbacklog(fid, backlog)
int fid;
int backlog;

cc files ... -lpup

DESCRIPTION

Ensetbacklog is used to set the input queue backlog (the maximum number of received packets that will be queued pending a read) for the ethernet minor device denoted by the *fid* parameter. *Backlog* specifies the maximum queue length; if *backlog* is zero, a default value will be used, and if it is higher than the maximum allowable value, the maximum will be used.

AUTHOR

Jeffrey Mogul

SEE ALSO

enopen(9), enet(4)

NAME

ensignal – enable or disable signal on ethernet packet arrival

SYNOPSIS

#include <sys/signal.h>

ensignal(fid, signum)
int fid;
int signum;

cc files ... -lpup

DESCRIPTION

Ensignal is used to specify a Unix signal to be delivered when a packet arrives for the ethernet minor device denoted by the *fid* parameter. *Signum* is the signal number; it is wise to choose one that is otherwise unlikely to occur.

Once the signal has been set, it will be delivered each time a packet arrives (even if the packet is discarded by the ethernet driver because the input queue is full.) To disable the signal, call *ensignal* with a *signum* of zero.

The idea is that the calling program has set up a routine to catch the specified signal and then read a packet from the ethernet.

AUTHOR

Jeffrey Mogul

SEE ALSO

enopen(9), enread(9), pupint(9), signal(3), sigvec(2)

DIAGNOSTICS

None.

BUGS

Writing code to handle signals properly is a black art. It is far safer to use *select*(2) when possible.

NAME

enwrite — write a packet to the ethernet

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/puppacket.h>
#include <pup/pupstatus.h>

enwrite(DstHost, Ptype, fid, devparams, packet, len)
char *DstHost;
unsigned short Ptype;
int fid;
union DevParams *devparams;
struct EnPacket *packet;
int len;

cc files ... -lpup
```

DESCRIPTION

Enwrite is used to write a packet onto the ethernet. The buffer refered to by *DstHost* is the hardware address of the immediate destination host; *Ptype* is the Ethernet packet type (*not* the Pup packet type) to be inserted into the packet; *fid* is the Unix file id of the ethernet device and *devparams* the device parameters structure, both returned by *enopen(9)*.

Packet is the address of the *Pup* fi f a *NetPacket* structure where the encapsulated data has been put. The caller must allocated for the entire netpacket, but must pass the address of the data part of the packet; *enwrite* will truct the ethernet header. This makes it possible for the caller to use this routine without know how large the ethernet header is. For example:

```
struct NetPacket pbuf;

...

status = enwrite(&DstHost, fid, &devparams, &(pbuf.Pup), length);
```

Len is the length of the *encapsulated* part of the packet, i.e., not including the ethernet header. The length of the encapsulated packet should be an even number.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

enread(9), enopen(9)

DIAGNOSTICS

If the *devparams* structure is invalid, returns PCNOTOPEN (a poor choice of codes). Otherwise, always returns OK.

NAME

madd to name — translate a Pup Port address to a name

SYNOPSIS

```
# include <pup/puplib.h>
# include <pup/pupconstants.h>
# include <pup/pupstatus.h>

madd to name(RemPort,name)
struct Port *RemPort;
char    *name;

cc files ... -lpup
```

DESCRIPTION

Madd to name is used to translate a Pup Net/Host/Socket identifier (a *Port)* into a string name. This is done by making a request to the *MiscServices* server on the local net (the request is broadcast to all hosts.)

The input parameter is a Port, which includes the net number, the host number, and the socket number of the desired name. Any of these can be zero, and the name server will interpret this as it pleases. The returned string is of the form "HostName+SocketName", or just "HostName" if the socket is zero. The value OK is returned on a successful translation.

AUTHOR

Jeffrey Mogul

SEE ALSO

nleokup(9)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channel was available.

TIMEOUT means that the server did not respond within a reasonable timeout period.

NOTFOUND indicates the specified Net/Host/Socket was not found. The returned string **name** contains a human-readable error message.

BUGS

The function assumes that the server host is one which is on the local network, and that it is accepting broadcast packets. Both assumptions are reasonable.

NAME

mattributes -- get Pup Network Directory entry attributes for an address

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mattributes(RemPort,attributes)
struct Port *RemPort;
char    *attributes;

cc files ... -lpup
```

DESCRIPTION

Mattributes is used to get the "attributes" associated with the network directory entry for a Pup Net/Host/Socket identifier (a *Port)* into a string *attributes.* This is done by making a request to the *MiscServices* server on the local net (the request is broadcast to all hosts.)

The input parameter is a Port, which includes the net number, the host number, and the socket number of the desired site. Any of these can be zero, and the name server will interpret this as it pleases. (However, this is most likely to succeed if the Port denotes a specific host.) The returned string is of the form

AttributeName: AttributeValue [AttributeName: AttributeValue ...]

That is, it gives the name and value of one or more attributes for the specified host. (Actually, if there are no attributes, the string will be empty.) The most likely attributes are "Location" and "Owner". Thus, this can be used to find the supposed location of a host.

OK is returned if the call is successful.

AUTHOR

Jeffrey Mogul

SEE ALSO

mlookup(9), maddtoname(9)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channel was available.

TIMEOUT means that the server did not respond within a reasonable timeout period.

NOTFOUND) indicates the specified Net/Host/Socket was not found. The returned string **attributes** contains a human-readable error message.

BUGS

This protocol is peculiar to Stanford.

It is also pretty dumb; there is no convention for delimiting attributes, and there is no way of guaranteeing that any given attribute will be returned. Also, if the list of attributes won't fit in a packet, things will probably not work well.

PROTOCOL

For lack of a better place, the protocol is documented here.

Request Format The PupType is ATTRIBUTESREQ (0,310) and the data portion

is the Port.

Reply Format

The PupType is ATTRIBUTESREP (0311) and the data portion is the attribute string. Errors are indicated by PupType LOOKUPERR (Name Lookup Error), with a human-readable explanation string in the data portion.

NAME

mbootdir — get a boot file directory from a boot server

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mbootdir(ServerPort, buffer, buflen)
struct Port *ServerPort;
char *buffer;
int *buflen

cc files ... -lpup
```

DESCRIPTION

Mbootdir is used to request (from the boot file server whose address is in ServerPort) a list of the availble boot files. What is returned is a buffer full of entries that *may be larger than the maximum size of a pup!The*lengthof*/*the the reference parameter buflen. In order to get this (almost) right, the routine may take a fair amount of time (on the order of 3 seconds) to get a boot file.

The structure of each entry in the returned buffer is: a short word giving the "boot file number", two shorts giving an Alto format date (low-order word first), and a boot file name in BCPL string format (see *pupstring*(9)).

The ServerPort may be a broadcast address.

AUTHOR

Jeffrey Mogul

SEE ALSO

mbootreq(9), pupstring(9)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channel was available.

TIMEOUT means that the server did not respond within a reasonable time.

BUGS

The protocol used involves sending one request packet, then awaiting one *or more* reply packets. There seems to be no way to get this right short of some hairy timeout hackery, so the routine may occasionally return a partial directory. The user should try again if things don't look right.

NAME

mbootrequest -- request a boot-load

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mbootrequest(DstPort, BootSock, BFNumber)
struct Port *DstPort;
unsigned long BootSock;
unsigned long BFNumber;
```

DESCRIPTION

Mbootrequest is used to request that a boot server, designated by *DstPort*, send a file via the Alto bootstrap protocol, which is documented elsewhere. The *BootSock* parameter is sent to the server to indicate what socket on the local host is to receive the boot file. The *BFNumber* is sent to the server to indicate the Boot File Number which is requested.

Only one attempt is made, and no acknowledgement is expected. The bootstrap loader should retry this function if no file arrives within a reasonable timeout. The normal return value is OK.

AUTHOR

Jeffrey Mogul

SEE ALSO

unsmbootreq(9)

DIAGNOSTICS

NOCHAN is returned if no Pup channels are available.

NAME

mkissofdeath – send a KissOfDeath Pup

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mkissofdeath(DstPort, DPupID)
struct Port *DstPort;
unsigned long DPupID;
```

DESCRIPTION

Mkissofdeath is used to send a KissOfDeath packet to the specified port. The PupID of the packet is the one specified by the caller. Only one attempt is made, and no acknowledgement is expected. The normal return value is OK.

AUTHOR

Jeffrey Mogul

SEE ALSO

miscserver(9)

DIAGNOSTICS

NOCHAN is returned if no Pup channels are available.

NAME

mlookup — translate a name to a Pup address

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mlookup(name,RetPort)
char    *name;
struct Port *RetPort;

cc files ... -lpup
```

DESCRIPTION

Mlookup is used to translate a string name into a Pup Net/Host/Socket identifier. This is done by making a request to the *MiscServices* server on the local net (the request is broadcast to all hosts.)

The input parameter is a string which is to be translated into a numeric identifier. It can either be of the form "HostName+SocketName", or of the form "HostName", in which case the returned socket will be zero. The output parameter is the Port address of the named entity. This is passed back to the caller, so the actual parameter is an address. The function value OK is returned on a successful lookup.

AUTHOR

Jeffrey Mogul & John Scamons

SEE ALSO

maddtoname(9), port(9), puperrmsg(9)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channel was available.

TIMEOUT means that the server did not respond within a reasonable time.

NOTFOUND indicates the specified name was not found. The (global) returned string **PupErrMsg** contains a human-readable error message.

BUGS

The function assumes that the server host is one which is on the local network, and that it is accepting broadcast packets. Both assumptions are reasonable.

When translating an octal host number of the form "300", the function returns "0#300#0". This is intuitively right, but wrong according to Xerox software. (On the other hand, it parses "50#300" as "0#50#300"; i.e., the assumption is that #'s appear starting on the right.)

NAME

mmailcheck -- find out if a user has new mail at a Pup host

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mmailcheck(RemPort, UserName, message)
struct Port *RemPort;
char    *UserName;
char    *message;
```

DESCRIPTION

Mmailcheck is used to find out if a given user has new (unread) mail waiting at a given host. This is done by making a *MailCheck* request to the *MiscServices* server at the specified host. (Specifically, a "Laurel-style Mail Check Request" is made.)

The input parameters are the address of the host (the socket field is ignored; mail check requests are always sent to the MiscServices socket) and the username (actually, mailbox name) **UserName.** The host and net numbers can be zero, and the request will be broadcast to all hosts -- only the first answer will be returned. The following non-error return codes are possible:

NEWMAIL indicates that new mail exists; there may be a human-readable explanation in **message.**

NONEWMAIL indicates that no new mail exists; a human-readable explanation is in **message.**

NOSUCHMBOX indicates that no mailbox with the given name exists at the host; a human-readable explanation is in **message.**

The parameter *RemPort* is passed by reference to allow *mmailcheck* to return the address of the host that actually responded, in the case of a broadcast request. The calling program can then use *maddtoname(9)* to determine the name of the responding host.

AUTHOR

Jeffrey Mogul

SEE ALSO

mlookup(9), maddtoname(9), mailcheck(1)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channels were avaiable.

TIMEOUT means that the server did not respond within a reasonable period.

NAME

msesrvreq – make a MiscServices request

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

msesrvreq(RemPort, InBuf, InBufLen, OutBuf, OutBufLen, ReqType, Timeout)
struct Port *RemPort;
char *InBuf;
int InBufLen;
char *OutBuf;
int *OutBufLen;
unsigned char ReqType;
int Timeout;

cc files ... -lpup
```

DESCRIPTION

Msesrvreq is a routine meant to be the core of most requests to a Miscellaneous Services server. The caller sets up a buffer to send to the server, described by *InBuf* and *InBufLen*, and a Port structure (*RemPort*) that gives the host/net address of the server to use (using the address 0#0# will cause the request to be broadcast on the local net.) The caller also passes (in *ReqType*) the Pup Type of the request (e.g., MAILCHECK, ATIMEREQ, etc.), and the number of seconds to wait before timeout.

The function returns the Pup Type of the reply received, and uses *OutBuf* to store the reply buffer, and *OutBufLen* to store the length of the reply buffer. The true host/net address of the responding server is returned in *RemPort*.

AUTHORS

John Seamons, Jeff Mogul

SEE ALSO

pupport(9)

DIAGNOSTICS

Returns TIMEOUT if the request timed out; returns NOCHAN if there were no available Pup channels.

BUGS

The manner of signalling error on return succeeds only because TIMEOUT and NOCHAN are distinct from the possible MiscServices Pup types. This is a lousy way to code (sorry.)

NAME

msendumsg – send a message to one or all users at a Pup host

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

msendumsg(SenderName, ReceiverName, message, RemPort);
char    *SenderName;
char    *ReceiverName;
char    *message;
struct Port *RemPort;

cc files ... -lpup
```

DESCRIPTION

Msendumsg is used to send a short message (no more than 500 characters) to a user, or all users, logged in at a host on a Pup network. The remote host must be running the Stanford MiscServices server.

The protocol is a connectionless one (one packet per message), so only as many bytes as will fit into one Pup packet may be sent. The message is sent with a Pup type of SENDUMSG; the Pup Data portion of the packet contains the single null-terminated string "SenderName:ReceiverName:message"; the message has new-lines embedded. If the Receiver-Name field is '*', then the server at the receiving host delivers the message to all logged-in users.

The address of the receiving host is designated by *Remport*. The value SENDUACK is returned if the remote host received and delivered the message. If the host address is zero, then all hosts will receive the message; the function will only keep retrying until it receives one response, so delivery is only "guaranteed" to one host. (Actually, it is not guaranteed at all.)

AUTHOR

Jeffrey Mogul

SEE ALSO

miscserver(9), send(1)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channel was available.

TIMEOUT means that the server did not respond within a reasonable period.

SENDUERR means that the remote host responded but did not accept the message, either because the user was not logged in, or because the host did not grant permission.

BUGS

The protocol used is peculiar to the Stanford implementation of Pup.

The message may be delivered more than once (there is a race in the timeout logic.)

I haven't implemented this one yet.

NAME

msunbootreq – request a boot-load

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

msunbootreq(DstPort, BootSock, BFNumber, BFName)
struct Port *DstPort;
unsigned long BootSock;
unsigned long BFNumber;
char *BFName;

cc files ... -lpup
```

DESCRIPTION

Msunbootreq is used to request that a boot server, designated by *DstPort*, send a file via the Sun bootstrap protocol, which is documented in /usr/sun/doc/sunboot/SunBoot.press. The *BootSock* parameter is sent to the server to indicate what socket on the local host is to receive the boot file. The *BFNumber* is sent to the server to indicate the Boot File Number which is requested. However, if the *BFName* parameter points to a non-null string, then the boot server will ignore the number given, and return the named file (the name must be meaningful to the server's host.)

Only one attempt is made, and no acknowledgement is expected. The bootstrap loader should retry this function if no file arrives within a reasonable timeout. The normal return value is OK.

AUTHOR

Jeffrey Mogul

SEE ALSO

mbootrequest(9)

DIAGNOSTICS

NOCHAN is returned if no Pup channels are available.

NAME

mtimecheck — get time from a Pup server

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

mtimecheck(TimePort, GMTime)
struct Port *TimePort;
long    *GMTime;

cc files ... -lpup
```

DESCRIPTION

Mtimecheck is used to find out what time a host thinks it is. This is done by making a *TimeCheck* request to the *MiscServices* server at the specified host. (Specifically, a "New Standard Alto-style Time Check Request" is made.)

The input parameter (passed by reference) is the Port address of the remote host. (The socket field is ignored; time check requests are always directed to the MiscServices socket.) The host and net numbers can be zero, and the request will be broadcast to all hosts — only the first answer will be returned. In any case, the Port structure pointed to by the parameter will be set to the actual Port address of the responding server. The time returned in **GMTime** is the number of seconds since midnight, January 1, 1901, Greenwich Mean Time (GMT). Note that this is different from the Unix base for time, which is midnight, January 1, 1970. The function returns the value OK if all goes well.

AUTHOR

Jeffrey Mogul

SEE ALSO

miscserver(9), timecheck(1m), timeconvert(9)

DIAGNOSTICS

There are several possible error returns:

NOCHAN means that no Pup channels were available.

TIMEOUT means that the server did not respond within a reasonable period.

BUGS

The MiscServices servers which properly implement this service return a few more words, indicating things like the local timezone and the local period when Daylight Savings Time should be in effect. *Mtimecheck* does not return these useful tidbits; Unix has essentially equivalent mechanisms anyway.

NAME

pupchan -- data structure describing a Pup channel

SYNOPSIS

#include <pup/puplib.h>

DESCRIPTION

A *PupChan* is a data structure which is used to describe a Pup channel, over which Pup-based communication may flow. The *only* operations a user should ever do on a PupChan are (1) define space for it (e.g., struct PupChan mychan;), and (2) pass its address to the routines in the Pup package (e.g., pupclose(&mychan); .) Any other reference to a PupChan is forbidden. PupChans, once opened, should not be deallocated until they are closed.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9), pupclose(9), pupsetmode(9), etc.

BUGS

No enforcement mechanism exists to keep foolish programmers from abusing PupChans.

NAME

pupclose – close a pup channel

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupstatus.h>

pupclose(Pchan)
struct PupChan *Pchan;

cc files ... -lpup
```

DESCRIPTION

Pupclose is used to close a Pup channel opened by *pupopen(9)*.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9)

DIAGNOSTICS

Normally, the return value is OK. However, if you try to close a Pup Channel that was not really open, you (probably) will get PCNOTOPEN.

NAME

(pupdescrip) pupindescrip, pupoutdescrip, pupfidtochan – access mapping between PupChans and fids

SYNOPSIS

```
pupindescrip(pchan)
struct PupChan *pchan;

pupoutdescrip(pchan)

struct PupChan *pupfidtochan(fid)
int fid;

cc files ... -lpup
```

DESCRIPTION

For each open PupChan (as returned by *pupopen*(9)), there is one or more Unix file descriptor ("fid") associated. If a program uses the *select*(9) system call in conjunction with Pup, it may need to find out what the associated fid is for a PupChan. *pupindescrip* returns the fid associated with input on the specified PupChan. *Pupoutdescrip* is provided for completeness, but is probably not useful.

The typical use of the file descriptor is to create a mask for use with *select*.

Also important for users of *select* with multiple file descriptors (the usual case) is way to get at the inverse mapping; once you know that input is pending on a given fid, *pupfidtochan* returns a pointer to the PupChan for which the fid is open.

For example,

```
long selmask = 0;       /* assuming no more than 32 fids */
        .
        .
        pupopen(&pchan, srcsocket, Dst);
        selmask |= (1<<pupindescrip(&pchan));
        .
        .
        .
        readfds = selmask;
        select(32, &readfds, 0, 0, 0);
        for (i = 0; readfds; i++) {
            if ((readfds & (1<<i)) == 0) continue;
            readfds &= ~(1 << i);
            chanp = pupfidtochan(i);
            pupread(chanp, ....);
            .
            .
            .
        }
        .
        .
        .
```

AUTHOR

Jeff Mogul

SEE ALSO

pupopen(9), select(2), puplisten(9)

NAME

PupErrMsg – human-readable error message from Pup package routines

SYNOPSIS

 # include <pup/puplib.h>

cc files ... -lpup

DESCRIPTION

PupErrMsg is a globally defined character string that is used to hold human-readable error messages from routines in the Pup package. It is for convenience only.

AUTHORS

Jeff Mogul

BUGS

The maximum size of the error message is (MAXPUPDATALEN – 1) bytes.

NAME

pupgethost, pupgetnet – get host, net numbers of local end of pup channel

SYNOPSIS

pupgethost(pchan)
struct PupChan *pchan;

pupgetnet(pchan)

cc files ... -lpup

DESCRIPTION

Pupgethost is passed a PupChan (as returned by *pupopen*(9)), and returns the host number of the local machine, with respect to the network over which the channel is communicating.

Pupgetnet is passed a similar PupChan, and returns the network number of the network over which the channel is communicating

Note that any given machine may be connected to several networks, and may have different host numbers on each network.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9), pupgetport(9)

BUGS

NAME

(pupgetport) pupgetsreport, pupgetdstport -- get address of local, remote ends of pup channel

SYNOPSIS

pupgetsreport(pchan, prt)
struct PupChan *pchan;
struct Port *prt;

pupgetdstport(pchan, prt)

cc files ... -lpup

DESCRIPTION

Pupgetsreport is passed a PupChan (as returned by *pupopen*(9)), and returns the Pup port address (net # host # socket) of the local end of the specified Pup channel. *Pupgetdstport* returns the address of the remote end.

AUTHOR

Jeff Mogul

SEE ALSO

pupopen(9), pupgethost(9)

NAME

pupint, pupnoint – enable or disable interrupts for received Pup Packets

SYNOPSIS

```
#include <pup/puplib.h>

pupint(Pchan, signum, routine, userarg)
struct PupChan *Pchan;
int signum;
void (*routine)();
char *userarg;

pupnoint(Pchan)

cc files ... -lpup -ljobs
```

DESCRIPTION

Pupint is used to enable the delivery of an interrupt to the user process when a packet arrives on the given Pup channel. This is used to avoid blocking on Pup reception, which is a problem in a duplex environment.

This call, by its nature, is currently Unix-dependent. The parameter *signum* is the number of a Unix signal which may be used in the process of delivering the interrupt. It is not a good idea to chose this number blindly; for example, Unix does not allow a process to catch SIGKILL. A particular signal should not be used elsewhere in the program, nor is it wise to use the same signal for two different Pup channels, since the signal number is what enables the code to differentiate between interrupts from different channels (unless you use a *select*(2).)

When a packet arrives, the specified routine is invoked as if it were called with

```
routine(Pchan, userarg);
```

This allows the interrupt service routine to know which Pup channel is involved. The *userarg* parameter can provide additional information; it could point to a data structure of some sort.

If the same routine is specified for more than one Pup channel, it should be written re-entrantly. However, it will not be called re-entrantly for more than one packet on any single Pup channel.

Pupnoint may be used to disable interrupts on a given Pup channel.

One warning: the interrupt is triggered by the arrival of a packet for the specified channel; however, it is delivered before it has been determined if the packet's checksum is good. Therefore, if you have set PCM_RCHECKSUM (using *pupsetmode*(9)), you will block if the received packet had a bad checksum and you call *pupread*(9). This can be avoided by setting PCM_IGNBADCKS; in this case, the caller of *pupread* will have to check the return status and throw away damaged packets.

SEE ALSO

signal(3), sigvec(2) for symbolic definitions of signals, so that you will know which of them to avoid.

AUTHOR

Jeffrey Mogul

DIAGNOSTICS

None. However, if a signal number out of the range of legal Unix signals is specified, nothing is done.

BUGS

This is highly Unix-dependent: the Pup-level implementation should not be in terms of signals at all. However, since the underlying implementation (i.e., the Unix ethernet driver) uses signals, it is important to be able to specify what signals are used. Otherwise, an "automatic" choice of signal assignments might conflict with a user programs's used of one the limited set of Unix signals. Expect that the signal argument will be obsoleted in the future.

The wise programmer will prefer to use *select*(2); see *pupdescrip*(9) for hints on how to do this.

NAME

puplisten, puplistenall — open Pup channels on all connected networks

SYNOPSIS

#include <pup/puplib.h>
#include <pup/pupstatus.h>

puplisten(action, LocalSocket)
int (*action)();
unsigned long LocalSocket;

puplistenall(action, LocalSocket)

cc files ... -lpup

DESCRIPTION

Puplisten is used to open a Pup channel (see *pupchan(9)*) on each of the networks to which the local machine is connected. It is much like *pupopen*(9), but it does not take a destination specification, and instead of returning a single PupChan to the caller, it instead calls the *action* function, passing the open PupChans to it. I.e., the *action* function should be declared

action(pchan)
struct PupChan *pchan;

Action will be called one or more times, and should do all of the option-setting that is usually done after a *pupopen*. Space for the PupChans that *puplisten* opens is allocated by *malloc*(3); if they are ever closed (with *pupclose*(9)), they should then be deallocated using *free*(3).

Puplistenall is identical to *puplisten, except that it opens a* reachable network in the Pup internet. It has a very specific and limited usefulness.

Puplisten and *puplistenall* is intended for use in server programs.

AUTHOR

Jeff Mogul

SEE ALSO

pupopen(9), pupdescrip(9), pupreopen(9)

DIAGNOSTICS

Returns OK if it succeeds; otherwise it returns NOCHAN if no channels are available; returns NOROUTE if you can't get there from here; returns ABORT if *malloc*(3) fails.

Even if it returns something other than OK, there may have been some PupChans successfully open; your code should keep track of this and close/free them if necessary.

BUGS

Depends upon the gatewayinfo server. If it cannot reach gatewayinfo via IPC, it will only open a channel for the default network. This could be fixed, but mostly by duplicating a lot of gatewayinfo's code in this routine.

Puplistenall won't work if there are too many Pup networks to have them all open at the same time.

NAME

(pupmisc) getlong, makelong, getshort, makeshort, getHiWord, getLoWord — miscellaneous Pup routines

SYNOPSIS

```
#include <pup/puplib.h>

unsigned long getlong(xl)
unsigned long xl;

unsigned long makelong(xl)

unsigned short getshort(xs)
unsigned short xs;

unsigned short makeshort(xs)

unsigned short getHiWord(xl)

unsigned short getLoWord(xl)

cc files ... -lpup
```

DESCRIPTION

Some miscellaneous routines for the Pup package. Note that most of these are coded as macros; you'd better not try to take their address or pass them to other functions.

Read *byteorder*(9) before reading this.

getlong	is the approved way of reading a 32-bit integer from a Pup data buffer or header.
makelong	is the approved way of creating a 32-bit integer to be stored in a Pup data buffer or header.
getshort	is the approved way of reading a 16-bit integer from a Pup data buffer or header.
makeshort	is the approved way of creating a 16-bit integer to be stored in a Pup data buffer or header.
getHiWord	returns the most significant 16 bits of a longword as a short integer.
getLoWord	returns the least significant 16 bits of a longword as a short integer.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz, Bill Nowicki

SEE ALSO

byteorder(9)

BUGS

NAME

(pupnettab) setpupnettab, getpupnettab, endpupnettab -- pup configuration table

SYNOPSIS

#include <pup/puplib.h>

struct pupnettab *getpupnettab()

int setpupnettab()

int endpupnettab()

cc files ... -lpup

DESCRIPTION

These routines are used to find out the configuration of Pup network interfaces attached to the local host.

Getpupnettab returns a pointer to an object with the following structure containing the broken-out fields of a line in the Pup network description file, */etc/pup/pupnettab*.

```
struct pupnettab {
        char            *pnt_devname;
        struct  Port    pnt_address;
};
```

The meanings of the fields are:

pnt_devname	a string giving the "base name" of the interface special file.
pnt_address	gives the Pup network and host numbers of the interface (the socket field is zero.) Either of these fields may be zero, to indicate "unknown".

Getpupnettab reads the next line of the file, opening the file if necessary.

Setpupnettab opens and rewinds the file.

Endpupnettab closes the file.

FILES

/etc/pup/pupnettab

FILE FORMAT

Comment lines begin with '*#*' in first column. Other lines have

devicename <white-space> *net # host #*

Numbers are in octal. For example,

```
# Comment line
/dev/eneta      50 # 327 #
```

The network number field is only important for those networks where a gateway is not present when *gatewayinfo* starts. To be safe, you should get this right: but getting it wrong won't be a disaster in most (not all) situations.

The host number field is only important for 10mb networks, so that *pup10arpser* can know what the local host number is.

The order of entries is important; the first entry is the one that will be used by default for name lookups and the like, and so should refer to the interface that is connected to the network best populated with server hosts.

DIAGNOSTICS

Null pointer (0) returned on EOF or fatal error. Syntax errors on stderr.

BUGS

All information is contained in a static area so it must be copied if it is to be saved.

NAME

pupopen – open a pup channel

SYNOPSIS

#include <pup/puplib.h>
#include <pup/pupstatus.h>

pupopen(Pchan, LocalSocket, RemotePort)
struct PupChan *Pchan;
unsigned long LocalSocket;
struct Port *RemotePort;

cc files ... -lpup

DESCRIPTION

Pupopen is used to open a Pup channel (see *pupchan(9)*) to be used for further Pup data transfers. The *Pchan* parameter is the address of a PupChan structure which will hold the state of the channel; the user is warned against meddling with this data structure. The *LocalSocket* parameter gives the socket on this host for the Pup channel (the routine gets the source host and net address internally.) The *RemotePort* parameter gives the net/host/socket address for the other end of the Pup channel; i.e., the destination of packets sent over the channel. *Pupopen* uses this information to establish routing and to chose a transport medium. If *RemotePort.net* is zero, the underlying routing code will choose some directly connected network; this probably make sense only for broadcasts. *RemotePort* may **not** be NULL (this is change from previous implementations.)

A channel opened with *pupopen* should be closed with *pupclose(9)*.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupclose(9), pupread(9), pupwrite(9), pupreopen(9)

DIAGNOSTICS

Returns OK if it succeeds; otherwise it returns NOCHAN if no channels are available; returns NOROUTE if you can't get there from here.

BUGS

NAME

port -- a data structure used in the Pup package, with support routines

SYNOPSIS

```
#include <pup/puplib.h>
#include <stdio.h>

... which defines ...

struct Port     {       /* describes Host# Net# Socket tuple */
        Host    host;
        Net     net;
        Socket  socket;
        };

PortEQ(P1,P2)
struct Port *P1;
struct Port *P2;

PortCopy(P1,P2)

PortPrint(P1)

FPortPrint(out, P1)
FILE *out;

SPortPrint(buffer, P1)
char *buffer;

PortDecode(out, P1)

PortPack(UP,PP)
struct Port UP;
struct PackedPort PP;

PortUnpack(PP,UP)

cc files ... -lpup
```

DESCRIPTION

The *Port* data structure is used to define the source or destination of a Pup packet. Note that the socket field is interpreted as a longword value, and that the actual packets may contain this sort of thing in a different format.

There are a variety of miscellaneous routines defined for manipulating Port structures; some are conveniences, but some are needed to provide machine-independence.

PortEQ	takes pointers to two Port structures, and returns true (non-zero) iff all three fields of both Ports are equal.
PortCopy	copies the Port structure pointed to by P1 to that pointed to by P2.
PortPrint	prints (on stdout) the "standard" octal representation of the Port structure whose address is passed. E.g., one might see "50#200#4" (the quotes are not printed) for net = 50, host = 200, socket = 4. Note that no spaces are printed.

FPortPrint	works like *PortPrint* but uses the specified stream instead of defaulting to stdout.
SPortPrint	works like *PortPrint* but puts its output in the buffer specified instead of printing it.
PortDecode	prints on the specified stream a rendition of the Port, giving the net and host numbers in octal, but the socket number in a human-readable form if possible. E.g., the same example would print as "50#200#MiscServices".

Unfortunately, the Port structure cannot be used inside of Pup packets for two reasons. First, the C compiler throws in a gratuitous short word before the socket field, in an attempt to be efficient. Secondly, the order of the host and net fields (as well as the word order within the socket field) depends on the byte-ordering convention of the local host. Therefore, a structure called a PackedPort is defined internally for use within packets; it should be considered to be a six-byte structure that must be short-aligned within a packet. It should only be manipulated with one of the following routines:

PortPack	packs the Port structure whose address is UP into the PackedPort structure whose address is PP.
PortUnpack	unpacks the PackedPort structure whose address is PP into the Port structure whose address is UP.

BUGS

The type definitions for Host, Net, and Socket are (all unsigned) char, char, and long, respectively. This should be made more well-known.

NAME

(pupprint) PupPrint, PrintErrorPUP, PupTypeName -- printing routines for Pup

SYNOPSIS

```
#include <stdio.h>
#include <pup/puplib.h>
#include <pup/pupconstants.h>

PupPrint(out, buffer, length, PrintData)
FILE *out;
char *buffer;
int length;
int PrintData;

PrintErrorPUP(out, buffer, length)

char *PupTypeName(puptype, dstsocket, srcsocket)
unsigned char puptype;
unsigned long dstsocket;
unsigned long srcsocket;

cc files ... -lpup
```

DESCRIPTION

These routines are useful in debugging Pup programs, and providing low-level monitoring.

PupPrint — The buffer (and buffer length) passed to this routine should be an entire Pup packet. A formatted rendition of the Pup will be printed on the stream specified by *out*. If *PrintData* is non-zero, the entire data segment of the packet will be printed.

PrintErrorPUP — The buffer (and buffer length) passed to this routine should be the data portion of an ERRORPUP packet. A formatted rendition of the ErrorPup will be printed on the stream specified by *out*.

PupTypeName — Tries to figure out a human-readable name for the given Pup type, based on the specified source and destination sockets. The name is returned in a static string.

AUTHORS

Jeffrey Mogul, Bill Nowicki

SEE ALSO

pupport(9)

BUGS

None of these are perfect.

NAME

pupread -- read a pup packet from a pup channel

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

pupread(Pchan, buf, buflen, Ptype, ID, DstPort, SrcPort)
struct PupChan *Pchan;
char *buf;
int *buflen;
unsigned char *Ptype;
unsigned long *ID;
struct Port *DstPort;
struct Port *SrcPort;

cc files ... -lpup
```

DESCRIPTION

Pupread is used to read a pup packet from a pup channel (see *pupchan(9).*) The parameters are:

Pchan	The address of a PupChan structure as initialized by *pupopen(9).* A timeout and a packet filter should already have been set for this channel using *pupsettimeout(9)* and *pupsetfilter(9).*
buf	The address of a buffer to receive the data portion of the Pup packet. The buffer should be large enough to hold the largest possible Pup data packet, which is normally no more than 532 bytes (MAXPUPDATALEN). You should *not* expect that the bytes in the buffer whose indices are greater than the length specified by *buflen* are not modified by *pupread.* If this parameter is NULL, it is ignored (no data is returned.)
buflen	The address of an integer which will receive the length (in bytes) of the data portion of the pup packet. If this parameter is NULL, it is ignored (no data is returned.)
Ptype	The address of an unsigned char which will receive the PupType of the packet. If this parameter is NULL , it is ignored (no data is returned.)
ID	The address of an unsigned long which will receive the PupID of the packet. If this parameter is NULL, it is ignored (no data is returned.)
DstPort	The address of a Port structure (see *port(9)*) which will receive the host number, network number, and socket to which the pup packet was addressed. This is useful for packets received which may need to be distributed among several sockets, and for gateway purposes. If this parameter is NULL, it is ignored (no data is returned.)
SrcPort	The address of a Port structure which will receive the host/net/socket address of the sender of the Pup packet. If this parameter is NULL, it is ignored (no data is returned.) Unless PCM_WANTOSRC is set in the channel mode for *Pchan*, *pupread* will replace a zero network number in this structure with the number of the network it is recieved on.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9), pupclose(9), pupread(9), port(9), pupchan(9)

DIAGNOSTICS

Pupread can return OK if things went right, TIMEOUT if no packet was received during the channel's timeout period, and BADCKSUM if the Pup checksum was detected as being bad.

The data locations referenced by the parameters will be modified *only* if (1) the return value is OK, or (2) the return value is BADCKSUM, *and* the channel mode for *Pchan* contains the PCM_IGNBADCKS attribute. This allows programs that would like to receive even damaged packets to do so.

BUGS

Byte-swapping is problematical.

NAME

pupreopen -- change the destination for a Pup channel

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupstatus.h>

pupreopen(Pchan, DstPort)
struct PupChan *Pchan;
struct Port *DstPort;

cc files ... -lpup
```

DESCRIPTION

Pupreopen is used to change the destination for an open Pup channel (see *pupchan(9)*). The *DstPort* parameter gives the new net/host/socket address for the remote end of the Pup channel. *Pupreopen* uses this information to establish routing and to chose a transport medium.

AUTHORS

Jeff Mogul

SEE ALSO

pupopen(9), puplisten(9)

DIAGNOSTICS

Returns PCNOTOPEN if the PupChan isn't already open, or if it closes the channel and cannot reopen it; returns NOCHAN if no channels are available; returns NOROUTE if you can't get there from here; otherwise, it returns OK.

BUGS

NAME

puproute - find a route for a Pup Internet packet

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

puproute(DestPort, ViaHost, ViaNet, DevName, DevNameLen)
struct Port *DestPort;
Host *ViaHost;
Net *ViaNet;
char *DevName;
int DevNameLen;

cc files ... -lpup
```

DESCRIPTION

Puproute is used to find a route through the Pup internet to the specified destination port (net/host/socket address). It returns (in the reference parameters *ViaHost* and *ViaNet*) the first hop host address and net that should be used to reach the destination. *ViaHost* is either the Pup host number of the destination, if it is on a directly-connected network, or the Pup host number of the first gateway along the route. *ViaNet* is used to determine which directly connected net should be used, if there are more than one. If either *ViaHost* or *ViaNet* is NULL, it is ignored.

Puproute also returns in the buffer addressed by *DevName* the name of the device to be opened (i.e., the interface to *ViaNet*); *DevNameLen* should be the size of this buffer in bytes.

If *DestPort.net* is zero, then *puproute* will choose an arbitrary connected network; this is useful, for example, when the "destination" is a nameserver on whatever network is available.

Note that *puproute()* is normally called only by *pupopen*(9), and should not be seen at higher levels.

SEE ALSO

pupopen(9), puprouting(9)

DIAGNOSTICS

Normally returns OK; returns NOROUTE if a route cannot be found.

AUTHOR

Jeffrey Mogul @ Stanford

BUGS

There are two versions of this routine; one uses Unix IPC to try to ask the *gatewayinfo*(9) server for a route; if this fails, it proceeds to go out over the network to find a gateway. The routing table is constructed once (the first time it is needed) and if the first table received from the internet is deficient, or if the internet connectivity changes during the lifetime of the process, nasty things may happen.

NAME

puprouting – Pup Internet routing table maintenance routines

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>
#include <pup/puproute.h>

InitRoutingTable(devname)
char *devname;

UpdateRoutingTable(GWIPupData, GWILen, GWport)
char *GWIPupData;
int GWILen;
struct Port *GWport;

PurgeRoutingTable()

PrintRoutingTable()

extern struct PupRouteEntry PupRoutingTable[MAXPUPNETS];

cc files ... -lpup
```

DESCRIPTION

The routines described here are used to maintain the table used for determining internet routes for Pup packets. They are not normally seen by code outside the pup library (except perhaps in gateway servers), but they are being described for posterity's sake.

A routing table contains entries of type *PupRouteEntry*, defined in <pup/puproute.h>.

```
struct PupRouteEntry {
        Net     TargetNet;      /* net for which route is given */
        Net     GatewayNet;     /* network number for first gateway */
        Host    GatewayHost;    /* host number for first gateway */
        uchar   HopCount;       /* hops to TargetNet */
        long    UpdateTime;     /* time this entry was last updated */
}
```

Note that the local host is effectively a gateway to its directly connected networks, those with a hop-count of zero. If the hop-count is zero, then the proper action is to send packets directly to the target host; otherwise, they should be sent to *GatewayHost* via *GatewayNet*.

There is a globally defined routing table *PupRoutingTable*.

ROUTINES

The routines which modify an existing routing table (*UpdateRoutingTable* and *PurgeRoutingTable*) return true (nonzero) *iff* a "real change" is made to the table, one that should be propagated to other hosts.

InitRoutingTable — This is a simple routine which requests a routing table by broadcasting a "Gateway Information Request" and taking the first reply. This is not perfect, but it's as good as can be done if we cannot asynchronously listen for broadcasts. The *devname* parameter specifies which network interface to use; if multiple interfaces

exist, this routine can be called in sequence on each one to create the most accurate routing table.

UpdateRoutingTable — This takes the Pup data buffer from a "Gateway Information Reply" Pup, as specified by *GWIPupData* and *GWILen*, and the Pup source address of this data, *GWport*, and updates the specified routing table. The algorithm used is defined by the Xerox memorandum on the Gateway Information Protocol cited below.

PurgeRoutingTable — This should be called for a routing table periodically (on the order of once every 15 seconds or so). It removes entries that are too old to be considered valid, also according to the algorithm given in the Xerox memorandum.

PrintRoutingTable — This is used in debugging; it prints the given routing table on *stdout* in a human-readable format.

SEE ALSO

pupopen(9), puproute(9), [Lassen]<Pup>GatewayInformation.Press

DIAGNOSTICS

None.

AUTHOR

Jeffrey Mogul @ Stanford

BUGS

NAME

pupsetbacklog -- set input queue backlog for pup channel

SYNOPSIS

```
#include <pup/puplib.h>

pupsetbacklog(Pchan,backlog)
struct PupChan *Pchan;
int backlog;

cc files ... -lpup
```

DESCRIPTION

Pupsetbacklog sets the input backlog for the given Pup channel. The *backlog* value specifies the maximum number of packets that will be queued in the kernel pending a read system call.

If *backlog* is zero, a default value will be used, and if it is larger than the maximum alloable value, the maximum will be used.

AUTHORS

Jeff Mogul

SEE ALSO

pupopen(9)

NAME

pupsetdfilt − set a default packet filter for a Pup channel

SYNOPSIS

```
# include <pup/puplib.h>
# include <pup/pupconstants.h>
# include <pup/pupstatus.h>

pupsetdfilt(Pchan,priority)
struct PupChan *Pchan;
unsigned char priority;

cc files ... -lpup
```

DESCRIPTION

Pupsetdfilt sets a packet filter for the given Pup channel. A packet filter is a mechanism for specifying which packets should be delivered to your Pup channel. *Pupsetdfilt* decides, on the basis of the Pup channel's mode, what fields of incoming pup packets should be checked for equality against given values, and if the packet matches *all* of the specified values, then *pupread*(9) will deliver it to you. Otherwise, it will be checked against the packet filters for other Pup channels and processes.

The routine is given the address of a Pup channel structure, and a priority for the filter. Higher priority filters will be checked first, in case more than one filter accepts a packet.

Normally, the filter created will check the destination socket field of the incoming packet, and if the channel mode (see *pupsetmode*(9)) does not include PCM_BROADCAST, the destination host field as well.

If you want more control over the filter, use *pupsetfilter*(9).

AUTHORS

Jeff Mogul

SEE ALSO

pupsetfilter(9), pupopen(9)

DIAGNOSTICS

Normally, this should return OK. However, it is possible that it might return FILTERTOOBIG if something about the implementation of filters changes drastically.

BUGS

NAME

pupsetfilter -- set the packet filter for a Pup channel

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/pupstatus.h>

pupsetfilter(Pchan,priority,Fpuptype, Fpupid,Fdstnet,Fdsthost,Fdstsock, Fsrcnet,Fsrchost,Fsrcsock)
struct PupChan *Pchan;
unsigned char priority;
unsigned char *Fpuptype;
unsigned long *Fpupid;
unsigned char *Fdstnet;
unsigned char *Fdsthost;
unsigned long *Fdstsock;
unsigned char *Fsrcnet;
unsigned char *Fsrchost;
unsigned long *Fsrcsock;

cc files ... -lpup
```

DESCRIPTION

Pupsetfilter sets a packet filter for the given Pup channel. A packet filter is a mechanism for specifying which packets should be delivered to your Pup channel. *Pupsetfilter* allows you to specify certain fields of pup packets to be checked for equality against given values, and if the packet matches *all* of the specified values, then *pupread*(9) will deliver it to you. Otherwise, it will be checked against the packet filters for other Pup channels and processes.

The routine is always given the address of a Pup channel structure, and a priority for the filter. Higher priority filters will be checked first, in case more than one filter accepts a packet. The rest of the parameters are optional, in the sense that you can either give the *address* of a value for the filter to check, or NULL. If the parameter is null, the filter will ignore the associated field of the incoming packet.

The parameters are associated with, respectively, the packet's Pup Type, Pup Id, Pup Source net/host/socket, and Pup Destination net/host/socket.

If this seems too complex, look at *pupsetdfilt*(9), which tries to figure out what filter you really want.

AUTHORS

Jeff Mogul

SEE ALSO

pupsetdfilt(9), pupopen(9)

DIAGNOSTICS

Normally, this should return OK. However, it is possible that it might return FILTERTOOBIG if you try to specify too many fields to check. In the current implementation, you should be able to specify all of the fields.

BUGS

NAME

pupsetmode, pupgetmode — set and read the mode for a Pup channel

SYNOPSIS

```
# include <pup/puplib.h>

pupsetmode(Pchan,mode)
struct PupChan *Pchan;
short mode;

short pupgetmode(Pchan)

cc files ... -lpup
```

DESCRIPTION

Pup channels (see *pupchan*(9)) can have a number of attributes that are used to control activity over the channel. Collectively, the attributes of a channel are known as its *mode*. *Pupsetmode* and *pupgetmode* are used to set and read the channel mode.

Available modes are:

PCM_RCHECKSUM	Check the Pup checksums of packets when reading them. If this attribute is not set, the checksum is always ignored.
PCM_WCHECKSUM	Insert a checksum into the Pup packets when writing them. If this attribute is not set, insert a constant which means "this packet has no checksum."
PCM_BROADCAST	If this attribute is set, *pupsetdfilt*(9) will insure that packets directed to host 0 (broadcast packets) are not received. Otherwise, packets directed to either the local host, or to host 0, will be received.
PCM_IGNBADCKS	Setting this causes *pupread*(9) to ignore bad Pup checksums. Otherwise, *pupread*(9) will not alter any of its reference parameters if the checksum is bad, and if PCM_RCHECKSUM is set.
PCM_WANTOSRC	Setting this causes *pupread*(9) to return the Pup Source port of a received packet without modifying it. By default, *pupread* will try to ensure a non-zero net number in the Pup Source ports it hands back to the caller.

For convenience, symbols are defined for the clear state of each attribute; e.g., the clear state of PCM_RCHECKSUM is PCM_NORCHECKSUM. Each attribute is encoded as one bit; therefore, multiple attributes should be or'ed together to form a channel mode. Note that the default mode is all attributes in the clear state.

AUTHORS

Jeff Mogul

SEE ALSO

pupread(9), pupwrite(9), pupsetdfilt(9), pupopen(9), pupchan(9)

NAME

pupsettimeout – set timeout for pup channel

SYNOPSIS

```
#include <pup/puplib.h>

pupsettimeout(Pchan,timeout)
struct PupChan *Pchan;
int timeout;

cc files ... -lpup
```

DESCRIPTION

Settimeout sets the read timeout for the given Pup channel. The timeout value is (currently) in clock ticks (1/60 second). The constant ONESEC may be used as a one second timeout, and is likely to be transportable.

AUTHORS

Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9)

BUGS

The 1/60th timeout quantum may not be too transportable.

The Vax implementation has a maximum timeout of about 18 minutes. Any larger value will produce unpredictable results.

NAME

(pupstring) GetBCPLstring, PutBCPLstring, GetMesastring, PutMesastring -- manipulate strings

SYNOPSIS

```
#include <pup/puplib.h>

unsigned short GetMesastring(from,to)
struct MesaString *from;
char *to;

GetBCPLString(from,to)
struct BCPLstring *from;
char *to;

PutBCPLString(from, to)
char *from;
struct BCPLstring *to;

cc files ... -lpup
```

DESCRIPTION

Pup objects often contain text strings in formats peculiar to either Mesa or BCPL. These routines allow you to convert them to and from C strings (null terminated arrays of char).

Read *byteorder*(9) before reading this. These routines are meant for dealing with Mesa or BCPL strings inside of packets, and C strings outside of packets.

GetBCPLstring — takes the address of a BCPLstring (don't worry about what this structure looks like, just pass the address of what you think is a BCPLstring), extracts it into a C string (null-terminated, of course), and returns the length in bytes of the BCPLString structure that you gave it (not the length that strlen(to) would return.) This value is useful if you have a lot of BCPLstrings packed together.

PutBCPLstring — moves a C string into a BCPLstring, and returns the length in bytes of the BCPLstring object. The buffer whose address you pass for the *to* argument should be longer than the C string by a few (i.e., at least 4) bytes.

GetMesastring — takes the address of a MesaString (don't worry about what this structure looks like, just pass the address of what you think is a MesaString), extracts it into a C string (null-terminated, of course), and returns the length in bytes of the MesaString structure that you gave it. This value is useful if you have a lot of MesaStrings packed together.

PutMesastring — To be specified.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz, Bill Nowicki

SEE ALSO

byteorder(9)

BUGS

PutMesaString isn't implemented.

NAME

pupwrite — write a packet to a pup channel

SYNOPSIS

```
#include <pup/puplib.h>
#include <pup/pupconstants.h>
#include <pup/papstatus.h>

pupwrite(Pchan, Ptype, ID, buf, buflen)
struct Chan *Pchan;
unsigned char Ptype;
unsigned long ID;
char *buf;
int buflen;

cc files ... -lpup
```

DESCRIPTION

Pupwrite is used to send a Pup packet over a pup channel opened using *pupopen(1)*. The parameters are:

Pchan	The address of a Pup channel (as initialized by *pupopen(9)*), over which the pup packet should be sent.
Ptype	The PupType for the packet to be sent. Registered Pup types are defined in <pup/pupconstants.h>.
ID	The Pup ID for the packet to be sent.
buf	The address of the data to be sent.
buflen	The number of bytes to send.

Buffers should always have an even number of bytes allocated, even if the *buflen* parameter is odd, since *pupwrite* will round up the buffer length to an even number before copying it into the packet. This conforms to Pup specifications, which state that packets must contain a garbage byte if necessary, to make the packet an even length.

AUTHORS

John Seamons, Jeff Mogul, Dan Kolkowitz

SEE ALSO

pupopen(9), pupclose(9), pupread(9), port(9), pupchan(9)

DIAGNOSTICS

Returns OK if things went right.

BUGS

NAME

ringbuf - - ring buffer package

SYNOPSIS

```
#include <pup/ringbuf.h>

struct   RingBuf rngbuf;

rngbufinit(&rngbuf, databuf, maxbuflen)
char *databuf;
int maxbuflen;

int rngbufsize(&rngbuf)

rngbufput(&rngbuf, c)
char c;

char rngbufget(&rngbuf)

rngbufwrite(&rngbuf, inputbuf, len)
char *inputbuf;
int len;

rngbufread(&rngbuf, outputbuf, len)
char *outputbuf;
int len;

rngbufpeek(&rngbuf, outputbuf, len)

cc files ... -lpup
```

DESCRIPTION

The ring buffer package was written to provide a general purpose, *high-speed* ring buffer facility for use in network software. The object code is part of the pup library **libpup**. A ring buffer provides first-in, first-out buffering, in this case of bytes. The buffer space itself is provided by the user, and thus may be of any convenient size. However, one must not provide less buffer space than one intends to use; this package normally performs no consistency checks (to keep efficiency high), and will gladly trash all of your data without complaining. [The package can be compile to print warning messages on overflow or underflow, for debugging purposes.] Routines are provided for either byte-by-byte transfers, or block transfers, and use of these two modes may be intermixed at will.

The ring buffer must be initialized *once* by calling *rngbufinit* with the address of a *RingBuf* structure, the address of a data buffer, and the maximum number of bytes that the buffer can hold. After initialization, this data buffer should not be referenced directly, and of course it cannot be deallocated until no further use is contemplated.

It is often useful to know the number of bytes contained in the ring buffer, especially in order to avoid overflow or underflow. The function *rngbufsize* returns the number of available byte of data in a given ring buffer.

To insert bytes into a ring buffer, either *rngbufput* or *rngbufwrite* may be used. *Rngbufput* adds one byte to the ring buffer; *rngbufwrite* takes the address of a data buffer, and the length of the data buffer, and copies this data into the ring buffer. [Horrible things will happen if this overflows the available buffer space!]

To retrieve bytes from a ring buffer, either *rngbufget* or *rngbufread* may be used. *Rngbufget* is a function that returns one byte from the ring buffer; *rngbufread* takes the address of a data buffer, and the number of bytes to read, and copies the requested information from the ring buffer to this data buffer.

Both *rngbufget* and *rngbufread* do *destructive* reads; that is, any data that they return is removed from the ring buffer. Sometimes it is necessary to find out what would be the next data returned from the buffer without actually removing it from the buffer; in order to make this possible, *rngbufpeek* works exactly like *rngbufread*, except that it does not remove the data from the buffer.

CAUTIONS

If the ring buffer *ever* underflows or overflows, it is unlikely that *any* future operations will work properly. Moreover, your program may crash mysteriously!

Any or all of these functions and procedures may be re-implemented as macros or in assembly code for efficiency; thus, it is unwise to write programs that would be affected by one or both of these re-implementations

SEE ALSO

malloc(3)

DIAGNOSTICS

If compiled with the symbol DEBUG defined, these routines can print warning messages on the standard error stream. Otherwise, diagnostics are not provided.

AUTHOR

Jeffrey Mogul

BUGS

NAME

UAtimecvt - conversions from Unix to Alto time and vice versa

SYNOPSIS

#include <pup/UAtimecvt.h>

AtoUtime(clock)
UtoAtime(clock)

DESCRIPTION

AtoUtime and *UtoAtime* are used to convert internal time formats between Unix and Alto systems. Both systems keep time as the number of seconds since midnight, January 1, GMT, of some year. The difference is that Unix starts from 1970, whereas Altos start from 1901. (Note that midnight *starts* a new day, so the time at noon on January 1, 1901 is positive, on an Alto.) Intervening leapyears are taken into account as well.

AtoUtime converts a longword representing an Alto internal time to a longword representing the corresponding Unix internal time. *UtoAtime* does the complementary conversion. Make sure you use longwords!

Be warned that these are macros.

AUTHOR

Jeffrey Mogul

SEE ALSO

time(3), ctime(3)

NAME

UniqueSocket -- create unique socket number

SYNOPSIS

unsigned long UniqueSocket()

cc files ... -lpup

DESCRIPTION

UniqueSocket is a function that returns a number which is guaranteed to be unique during the current incarnation of the operating system. Uniqueness is guaranteed neither across crashes, nor across hosts, altbough using a combination of this number and the time-of-year returned by *time(2)* should give reasonable results.

The value returned is useful for creating unique sockets, connection ids, etc. In particular, it is guaranteed not to be the same as any "well-known" Pup socket.

AUTHORS

Jeff Mogul

BUGS

The "guarantee" of uniqueness is only probabilistiely correct (it gets worse as the product of process creation rate and average connection lifetime goes up), and does not hold if a single process manages to call the function more than 255 times during one clock tick. In practice, though, it should work well enough.

The MC68000 implementation is even worse; there is a global variable which is incremented on each call. This will stop working when there are multiple processes involved.