31. ETHERNET

Interlisp was first developed on large timesharing machines which provided each user with access to large amounts of disk storage, printers, mail systems, etc. Interlisp-D, however, was designed to run on smaller, single-user machines without these facilities. In order to provide Interlisp-D users with access to all of these services, Interlisp-D supports the Ethernet communications network, which allows multiple Interlisp-D machines to share common printers, file servers, etc.

Interlisp-D supports the Experimental Ethernet (3 Megabits per second) and the Ethernet (10 Megabits per second) local communications networks. These networks may be used for accessing file servers, remote printers, mail servers, or other machines. This chapter is divided into three sections: First, an overview of the various Ethernet and Experimental Ethernet protocols is presented. Then follow sections documenting the functions used for implementing PUP and NS protocols at various levels.

31.1 Ethernet Protocols

The members of the Xerox 1100 family (1108, 1132), Xerox file servers and laser xerographic printers, along with machines made by other manufacturers (most notably DEC) have the capability of communicating over 3 Megabit per second Experimental Ethernets, 10 Megabit per second Ethernets and telephone lines.

Xerox pioneered its work with Ethernet using a set of protocols known as PARC Universal Packet (PUP) computer communication protocols. The architecture has evolved into the newer Network Systems (NS) protocols developed for use in Xerox office products. All of the members of the Xerox 1100 family can use both NS and PUP protocols.

31.1.1 Protocol Layering

The communication protocols used by the members of the Xerox 1100 family are implemented in a "layered" fashion, which means that different levels of communication are implemented

as different protocol layers. Protocol Layering allows implementations of specific layers to be changed without requiring changes to any other layers. The layering also allows use of the same higher level software with different lower levels of protocols. Protocol designers can implement new types of protocols at the correct protocol level for their specific application in a layered system.

At the bottom level, level zero, there is a need to physically transmit data from one point to another. This level is highly dependent on the particular transmission medium involved. There are many different level zero protocols, and some of them may contain several internal levels. At level one, there is a need to decide where the data should go. This level is concerned with how to address a source and destination, and how to choose the correct transmission medium to use in order to route the packet towards its destination. A level one packet is transmitted by *encapsulating* it in the level zero packet appropriate for the transmission medium selected. For each independent communication protocol system, a single level one protocol is defined. The rule for delivery of a level one packet is that the communication system must only make a best effort to deliver the packet. There is no guarantee that the packet is delivered, that the packet is not duplicated and delivered twice, or that the packets will be delivered in the same order as they were sent.

The addresses used in level zero and level one packets are not necessarily the same. Level zero packets are specific to a particular transmission medium. For example, the destination address of a level zero packet transmitted on one of the two kinds of Ethernet is the Ethernet address (host number) of a machine on the particular network. Level one packets specify addresses meaningful to the particular class of protocols being implemented. For the PUP and NS protocols, the destination address comprises a network number, host number (not necessarily the same as the level zero host number), and a socket number. The socket number is a higher-level protocol concept, used to multiplex packets arriving at a single machine destined for separate logical processes on the machine.

Protocols in level two add order and reliability to the level one facilities. They suppress duplicate packets, and are responsible for retransmission of packets for which acknowledgement has not been received. The protocol layers above level two add conventions for data structuring, and implement application specific protocols.

31.1.2 Level Zero Protocols

Level zero protocols are used to physically connect computers. The addresses used in level zero protocols are protocol specific.

The Ethernet and Experimental Ethernet level zero protocols use host numbers, but level zero phone line protocols contain less addressing information since there are only two hosts connected to the telephone line, one at each end. As noted above, a level zero protocol does not include network numbers.

The 3MB Experimental Ethernet [1] was developed at PARC. Each Experimental Ethernet packet includes a source and destination host address of eight bits. The Experimental Ethernet standard is used by any machine attached to an Experimental Ethernet.

The 10MB Ethernet [2] was jointly developed and standardized by Digital, Intel, and Xerox. Each Ethernet level zero packet includes a source and destination host address that is 48 bits long. The Ethernet standard is used by any machine attached to an Ethernet.

Both of the level one protocols described later (PUP and NS) can be transported on any of the level zero protocols described above.

The Ethernet and Experimental Ethernet protocols are broadcast mediums. Data packets can be sent on these networks to every host attached to the net. A packet directed at every host on a network is a broadcast packet.

Other Level 0 protocols in use in industry include X.25, broadband networks, and Chaosnet. In addition, by using the notion of "mutual encapsulation", it is possible to treat a higher-level protocol (e.g. ARPANET) as if it were a Level Zero Protocol.

31.1.3 Level One Protocols

Two Level One Protocols are used in the Xerox 1100 Family, the PUP and the NS protocols. With the proper software, computers attached to Ethernets or Experimental Ethernets can send PUPs and NS packets to other computers on the same network, and to computers attached to other Ethernets or Experimental Ethernets.

The PUP protocols [3] were designed by Xerox computer scientists at the Palo Alto Research Center. The destination and source addresses in a PUP packet are specified using an 8-bit network number, an 8-bit host number, and a 32-bit socket number. The 8-bit network number allows an absolute maximum of 256 PUP networks in an internet. The 8-bit host number is network relative. That is, there may be many host number "1"s, but only one per network. 8 bits for the host number limits the number of hosts per network to 256. The

socket number is used for further levels of addressing within a specific machine.

The Network Systems (NS) protocols [4, 5] were developed by the Xerox Office Products Division. Each NS packet address includes a 32-bit network number, a 48-bit host number, and a 16-bit socket number. The NS host and network numbers are unique through all space and time. A specific NS host number is generally assigned to a machine when it is manufactured, and is never changed. In the same fashion, all networks (including those sold by Xerox and those used within Xerox) use the same network numbering space---there is only one network "74".

31.1.4 Higher Level Protocols

The higher level PUP protocols include the File Transfer Protocol (FTP) and the Leaf Protocol used to send and retrieve files from Interim File Servers (IFSs) and DEC File Servers, the Telnet protocol implemented by "Chat" windows and servers, and the EFTP protocol used to communicate with the laser xerographic printers developed by PARC ("Dovers" and "Penguins").

The higher level NS protocols include the Filing Protocol which allows workstations to access the product File Services sold by Xerox, the Clearinghouse Protocol used to access product Clearinghouse Services, and the TelePress Protocol used to communicate with the Xerox model 8044 Print Server.

31.1.5 Connecting Networks: Routers and Gateways

When a level one packet is sent from one machine to another, and the two machines are not on the same network, the packet must be passed between networks. Computers that are connected to two or more level zero mediums are used for this function. In the PUP world, these machines have been historically called "Gateways." In the NS world these machines are called Internetwork Routers (Routers), and the function is packaged and sold by Xerox as the Internetwork Routing Service (IRS).

Every host that uses the PUP protocols requires a PUP address; NS Hosts require NS addresses. An address consists of two parts: the host number and the network number. A computer learns its network number by communicating with a Router or Gateway that is attached to the same network. Host number determination is dependent on the hardware and the type of host number, PUP or NS.

Note that there is absolutely no relationship between a host's NS host and net numbers and the same host's PUP host and net numbers.

31.1.6 Addressing Conflicts with Level Zero Mediums

For convenience in the respective protocols, a level one PUP (8-bit) host number is the same as a level zero Experimental Ethernet host number; i.e., when a PUP level one packet is transported by an Experimental Ethernet to another host on the same network, the level zero packet specifies the same host number as the level one packet. Similarly, a level one NS (48-bit) host number is the same as a level zero Ethernet host number.

When a PUP level one packet is transported by an Ethernet, or an NS level one packet is sent on Experimental Ethernet, the level one host number cannot be used as the level zero address, but rather some means must be provided to determine the correct level zero address. Xerox solved this problem by specifying another level-one protocol called *translation* to allow hosts on an Experimental Ethernet to announce their NS host numbers, or hosts on an Ethernet to announce their PUP host numbers. Thus, both the Ethernet and Experimental Ethernet Level Zero Protocols totally support both families of higher level protocols.

31.1.7 References

[1] Robert M. Metcalfe and David R. Boggs, Ethernet: Distributed Packet Switching for Local Computer Networks, *Communications of the ACM*, vol. 19 no. 7, July 1976.

[2] Digital Equipment Corporation, Intel Corporation, Xerox Corporation. The Ethernet, A Local Area Network: Data Link Layer and Physical Layer Specifications. September 30, 1980, Version 1.0

[3] D. R. Boggs, J. F. Shoch, E. A. Taft, and R. M. Metcalfe, PUP: An Internetwork Architecture, *IEEE Transactions on Communications*, com-28:4, April 1980.

[4] Xerox Corporation. Courier: The Remote Procedure Call Protocol. Xerox System Integration Standard. Stamford, Connecticut, December, 1981, XSIS 038112.

[5] Xerox Corporation. Internet Transport Protocols. Xerox System Integration Standard. Stamford, Connecticut, December, 1981, XSIS 028112.

31.2 Higher-level PUP Protocol Functions

This section describes some of the functions provided in Interlisp-D to perform protocols above Level One. Level One functions are described in a later section, for the benefit of those users who wish to program new protocols.

(ETHERHOSTNUMBER *NAME*) [Function]

Returns the number of the named host. The number is 16-bit quantity, the high 8 bits designating the net and the low 8 bits the host. If *NAME* is **NIL**, returns the number of the local host.

(ETHERPORT *NAME ERRORFLG MULTFLG*) [Function]

Returns a port corresponding to *NAME*. A "port" is a network address that represents (potentially) one end of a network connection, and includes a socket number in addition to the network and host numbers. Most network functions that take a port as argument allow the socket to be zero, in which case a well-known socket is supplied. A port is currently represented as a dotted pair (*NETHOST* . *SOCKET*).

NAME may be a litatom, in which case its address is looked up, or a port, which is just returned directly. If *ERRORFLG* is true, generates an error "host not found" if the address lookup fails, else it returns **NIL**. If *MULTFLG* is true, returns a list of alternative port specifications for *NAME*, rather than a single port (this is provided because it is possible for a single name in the name database to have multiple addresses). If *MULTFLG* is **NIL** and *NAME* has more than one address, the currently nearest one is returned. **ETHERPORT** caches its results.

The *SOCKET* of a port is usually zero, unless the name explicitly contains a socket designation, a number or symbolic name following a **+** in *NAME*, e.g., **PHYLUM + LEAF**. A port can also be specified in the form "*NET#HOST#SOCKET*", where each of *NET*, *HOST* and *SOCKET* is a sequence of octal digits; the socket, but not the terminating #, can be omitted, in which case the socket is zero.

(ETHERHOSTNAME *PORT USE.OCTAL.DEFAULT*) [Function]

Looks up the name of the host at address *PORT*. *PORT* may be a numeric address, a (*NETHOST* . *SOCKET*) pair returned from **ETHERPORT**, or a numeric designation in string form, "*NET#HOST#SOCKET*", as described above. In the first case, the net defaults to the local net. If *PORT* is **NIL**, returns the name of the local host. If there is no name for the given port, but *USE.OCTAL.DEFAULT* is true, the function returns a string specifying the port in octal digits, in the form "*NET#HOST#SOCKET*", with *SOCKET* omitted if it is zero. Most

functions that take a port argument will also accept ports in this octal format.

(EFTP *HOST FILE PRINTOPTIONS***)** [Function]

Transmits *FILE* to *HOST* using the **EFTP** protocol. The *FILE* need not be open on entry, but in any case is closed on exit. **EFTP** returns only on success; if *HOST* does not respond, it keeps trying.

The principal use of the **EFTP** protocol is for transmitting Press files to a printer. If *PRINTOPTIONS* is non-**NIL**, **EFTP** assumes that *HOST* is a printer and *FILE* is a Press file, and takes additional action: it calls **PRINTERSTATUS** (page 29.4) for *HOST* and prints this information to the prompt window; and it fills in the "printed-by" field on the last page of the press file with the value of **USERNAME** (page 24.40). Also, *PRINTOPTIONS* is interpreted as a list in property list format that controls details of the printing. Possible properties are as follows:

#COPIES Value is the number copies of the file to print. Default is one.

#SIDES If the value is 2, select two-sided printing (if the printer can print two-sided copies).

DOCUMENT.CREATION.DATE Value is the document creation date to appear on the header page (an integer date as returned by **IDATE**).

DOCUMENT.NAME Value is the document name to appear on the header page (as a string). Default is the full name of the file.

31.3 Higher-level NS Protocol Functions

The following is a description of the Interlisp-D facilities for using Xerox SPP and Courier protocols and the services based on them. The sections on naming conventions, Printing, and Filing are of general interest to users of Network Systems servers. The remaining sections describe interfaces of interest to those who wish to program other applications on top of either Courier or SPP.

31.3.1 Name and Address Conventions

Addresses of hosts in the NS world consist of three parts, a network number, a machine number, and a socket number. These three parts are embodied in the Interlisp-D data type **NSADDRESS**. Objects of type **NSADDRESS** print as "net#h1.h2.h3#socket", where all the numbers are printed in octal radix, and the 48-bit host number is broken into three

16-bit fields. Most functions that accept an address argument will accept either an **NSADDRESS** object or a string that is the printed representation of the address.

Higher-level functions accept host arguments in the form of a symbolic name for the host. The NS world has a hierarchical name space. Each object name is in three parts: the *Organization*, the *Domain*, and the *Object* parts. There can be many domains in a single organization, and many objects in a single domain. The name space is maintained by the *Clearinghouse*, a distributed network database service.

A Clearinghouse name is standardly notated as *object:domain:organization*. The parts *organization* or *domain:organization* may be omitted if they are the default (see below). Alphabetic case is not significant. Internally, names are represented as objects of data type **NSNAME**, but most functions accept the textual representation as well, either as a litatom or a string. Objects of type **NSNAME** print as *object:domain:organization*, with fields omitted when they are equal to the default. A *Domain* is standardly represented as an **NSNAME** in which the object part is null. If frequent use is to be made of an NS name, it is generally preferable to convert it to an **NSNAME** once, by calling **PARSE.NSNAME**, then passing the resultant object to all functions desiring it.

CH.DEFAULT.ORGANIZATION [Variable]

This is a string specifying the default Clearinghouse organization.

CH.DEFAULT.DOMAIN [Variable]

This is a string specifying the default Clearinghouse domain. If it or the variable **CH.DEFAULT.ORGANIZATION** is **NIL**, they are set by Lisp system code (when they are needed) to be the first domain served by the nearest Clearinghouse server.

In small organizations with just one domain, it is reasonable to just leave these variables **NIL** and have the system set them appropriately. In organizations with more than one domain, it is wise to set them in the site initialization file, so as not to be dependent on exactly which Clearinghouse servers are up at any time.

(PARSE.NSNAME *NAME #PARTS DEFAULTDOMAIN***)** [Function]

When *#PARTS* is 3 (or **NIL**), parses *NAME*, a litatom or string, into its three parts, returning an object of type **NSNAME**. If the domain or organization is omitted, defaults are supplied, either from *DEFAULTDOMAIN* (an **NSNAME** whose domain and

organization fields only are used) or from the variables **CH.DEFAULT.DOMAIN** and **CH.DEFAULT.ORGANIZATION**.

If *#PARTS* is 2, *NAME* is interpreted as a domain name, and an **NSNAME** with null object is returned. In this case, if *NAME* is a full 3-part name, the object part is stripped off.

If *#PARTS* is 1, *NAME* is interpreted as an organization name, and a simple string is returned. In this case, if *NAME* is a 2- or 3-part name, the organization is extracted from it.

If *NAME* is already an object of type **NSNAME**, then it is returned as is (if *#PARTS* is 3), or its domain and/or organization parts are extracted (if *#PARTS* is 1 or 2).

(NSNAME.TO.STRING *NSNAME FULLNAMEFLG***)** [Function]

Converts *NSNAME*, an object of type **NSNAME**, to its string representation. If *FULLNAMEFLG* is true, the full printed name is returned; otherwise, fields that are equal to the default are omitted.

Programmers who wish to manipulate **NSADDRESS** and **NSNAME** objects directly should load the Library package **ETHERRECORDS**.

31.3.2 Clearinghouse Functions

This section describes functions that may be used to access information in the Clearinghouse.

(START.CLEARINGHOUSE *RESTARTFLG***)** [Function]

Performs an expanding ring broadcast in order to find the nearest Clearinghouse server, whose address it returns. If a Clearinghouse has already been located, this function simply returns its address immediately, unless *RESTARTFLG* is true, in which case the cache of Clearinghouse information is invalidated and a new broadcast is performed. **START.CLEARINGHOUSE** is normally performed automatically by the system the first time it needs Clearinghouse information; however, it may be necessary to call it explicitly (with *RESTARTFLG* set) if the local Clearinghouse server goes down.

CH.NET.HINT [Variable]

A number or list of numbers, giving a hint as to which network the nearest Clearinghouse server is on. When **START.CLEARINGHOUSE** looks for a Clearinghouse server, it probes the network(s) given by **CH.NET.HINT** first, performing the expanding ring broadcast only if it fails there. If the nearest Clearinghouse server is not on the directly connected network,

setting **CH.NET.HINT** to the proper network number in the local site init file (page 12.1) can speed up **START.CLEARINGHOUSE** considerably.

(SHOW.CLEARINGHOUSE *ENTIRE.CLEARINGHOUSE? DONT.GRAPH***)** [Function]

This function displays the structure of the cached Clearinghouse information in a window. Once created, it will be redisplayed whenever the cache is updated, until the window is closed. The structure is shown using the Library package **GRAPHER**.

If *ENTIRE.CLEARINGHOUSE?* is true, then this function probes the Clearinghouse to discover the entire domain:organization structure of the Internet, and graphs the result. If *DONT.GRAPH* is true, the structure is not graphed, but rather the results are returned as a nested list indicating the structure.

(LOOKUP.NS.SERVER *NAME TYPE FULLFLG***)** [Function]

Returns the address, as an **NSADDRESS**, for the object *NAME*. *TYPE* is the property under which the address is stored, which defaults to **ADDRESS.LIST**. The information is cached so that it need not be recomputed on each call; the cache is cleared by restarting the Clearinghouse. If *FULLFLG* is true, returns a list whose first element is the canonical name of *NAME* and whose tail is the address list.

The following functions perform various sorts of retrieval operations on database entries in the Clearinghouse. Here, "The Clearinghouse" refers to the collective service offered by all the Clearinghouse servers on an internet; Lisp internally deals with which actual server(s) it needs to contact to obtain the desried information. The argument(s) describing the objects under consideration can be strings or **NSNAME**'s, and in most cases can contain the wild card "*", which matches a subsequence of zero or more characters. Wildcards are permitted only in the most specific field of a name (e.g., in the object part of a full three-part name). When an operation intended for a single object is instead given a pattern, the operation is usually performed on the first matching object in the database, which may or may not be interesting.

(CH.LOOKUP.OBJECT *OBJECTPATTERN***)** [Function]

Looks up *OBJECTPATTERN* in the Clearinghouse database, returning its canonical name (as an **NSNAME**) if found, **NIL** otherwise. If *OBJECTPATTERN* contains a "*", returns the first matching name.

(CH.LIST.ORGANIZATIONS *ORGANIZATIONPATTERN*) [Function]

Returns a list of organization names in the Clearinghouse database matching *ORGANIZATIONPATTERN*. The default pattern is "*", which matches anything.

(CH.LIST.DOMAINS *DOMAINPATTERN*) [Function]

Returns a list of domain names (two-part **NSNAME**'s) in the Clearinghouse database matching *DOMAINPATTERN*. The default pattern is "*", which matches anything in the default organization.

(CH.LIST.OBJECTS *OBJECTPATTERN PROPERTY*) [Function]

Returns a list of object names matching *OBJECTPATTERN* and having the property *PROPERTY*. *PROPERTY* is a number or a symbolic name for a Clearinghouse property; the latter include **USER**, **PRINT.SERVICE**, **FILE.SERVICE**, **MEMBERS**, **ADDRESS.LIST** and **ALL**.

For example,

(CH.LIST.OBJECTS "*:PARC:Xerox" (QUOTE USER))

returns a list of the names of users in the domain PARC:Xerox.

(CH.LIST.OBJECTS "*lisp*:PARC:Xerox" (QUOTE MEMBERS))

returns a list of all group names in PARC:Xerox containing the substring "lisp".

(CH.LIST.ALIASES *OBJECTNAMEPATTERN*) [Function]

Returns a list of all objects in the Clearinghouse database that are aliases and match *OBJECTNAMEPATTERN*.

(CH.LIST.ALIASES.OF *OBJECTPATTERN*) [Function]

Returns a list of all objects in the Clearinghouse database that are aliases of *OBJECTPATTERN*.

(CH.RETRIEVE.ITEM *OBJECTPATTERN PROPERTY INTERPRETATION*) [Function]

Retrieves the value of the *PROPERTY* property of *OBJECTPATTERN*. Returns a list of two elements, the canonical name of the object and the value. If *INTERPRETATION* is given, it is a Clearinghouse type (see page 31.19) with which to interpret the bits that come back; otherwise, the value is simply of the form **(SEQUENCE UNSPECIFIED)**, a list of 16-bit integers representing the value.

(CH.RETRIEVE.MEMBERS *OBJECTPATTERN PROPERTY —*) [Function]

Retrieves the members of the group *OBJECTPATTERN*, as a list of **NSNAME**s. *PROPERTY* is the Clearinghouse Group property

under which the members are stored; the usual property used for this purpose is **MEMBERS**.

(CH.ISMEMBER *GROUPNAME PROPERTY SECONDARYPROPERTY NAME***)** [Function]

Tests whether *NAME* is a member of *GROUPNAME*'s *PROPERTY* property. This is a potentially complex operation; see the description of procedure **IsMember** in the Clearinghouse Protocol documentation for details.

31.3.3 NS Printing

This section describes the facilities that are available for printing Interpress masters on NS Print servers.

(NSPRINT *PRINTER FILE OPTIONS***)** [Function]

This function prints an Interpress master on *PRINTER*, which is a Clearinghouse name represented as a string or **NSNAME**. If *PRINTER* is **NIL**, **NSPRINT** uses the first print server registered in the default domain. *FILE* is the name of an Interpress file to be printed. *OPTIONS* is a list in property list format that controls details of the printing (see **SEND.FILE.TO.PRINTER**, page 29.1).

(NSPRINTER.STATUS *PRINTER***)** [Function]

This function returns a list describing the printer's current status; whether it is available or busy, and what kind of paper is loaded.

(NSPRINTER.PROPERTIES *PRINTER***)** [Function]

This function returns a list describing the printer's capabilities at the moment; the type of paper loaded, whether it can print two-sided, etc.

31.3.4 SPP Stream Interface

This section describes the stream interface to the Sequenced Packet Protocol. SPP is the transport protocol for Courier, which in turn is the transport layer for Filing and Printing.

(SPP.OPEN *HOST SOCKET PROBEP NAME PROPS***)** [Function]

This function is used to open a bidirectional SPP stream. There are two cases: user and server.

User: If *HOST* is specified, an SPP connection is initiated to *HOST*, an **NSADDRESS** or string representing an NS address. If the socket part of the address is null (zero), it is defaulted to *SOCKET*. If both *HOST* and *PROBEP* are specified, then the connection is

probed for a response before returning the stream; **NIL** is returned if *HOST* doesn't respond.

Server: If *HOST* is **NIL**, a passive connection is created which listens for an incoming connection to local socket *SOCKET*.

SPP.OPEN returns the input side of the bidirectional stream; the function **SPPOUTPUTSTREAM** is used to obtain the output side. The standard stream operations **BIN**, **READP**, **EOFP** (on the input side), and **BOUT**, **FORCEOUTPUT** (on the output side), are defined on these streams, as is **CLOSEF**, which can be applied to either stream to close the connection.

NAME is a mnemonic name for the connection process, mainly useful for debugging.

PROPS is an optional property list, used to set the properties that determine the behavior of the SPP stream when certain events occur. The following properties can be specified:

CLOSEFN A function or list of functions called (with the stream as argument) when an SPP connection is closed.

ATTENTIONFN A function called (with the stream as argument) when an ATTENTION packet is received on the SPP connection.

ERRORHANDLER A function called (with the stream as argument) when an error (such as end-of-stream) occurs on the SPP connection.

OTHERXIPHANDLER A function called (with the stream as argument) when a non-SPP, non-error packet is received on the socket associated with the SPP connection.

EOM.ON.FORCEOUTPUT The value of this property should be either **T** or **NIL** (the default). If **T**, then the end-of-message bit is set when the current collection of bytes buffered for transmission is forcibly sent (e.g. by **FORCEOUTPUT**, page 25.10).

SERVER.FUNCTION This property can be used for creating SPP servers. Normally, when a connection is opened with the *HOST* argument set to **NIL**, a passive "listener" connection is created. **SPP.OPEN** will not return until some other host attempts to connect to socket specified in the **SPP.OPEN** call.

If the **SERVER.FUNCTION** property is specified, a new listener (and listener process) is created. **SPP.OPEN** will return immediately. Whenever another host attempts to connect to the specified socket, a new process and unique SPP connection are created. The function specified by the **SERVER.FUNCTION** property is run in the top level of the new process. The server function should be a function of two arguments: the first argument is the SPP input stream associated with the connection; the second argument is the SPP output stream associated with the connection.

(SPPOUTPUTSTREAM *STREAM***)** [Function]

Applied to the input stream of an SPP connection, this function returns the corresponding output stream.

SPP.USER.TIMEOUT [Variable]

Specifies the time, in milliseconds, to wait before deciding that a host isn't responding.

(SPP.DSTYPE *STREAM DSTYPE***)** [Function]

Accesses the current datastream type of the connection. If *DSTYPE* is **NIL**, returns the datastream type of the current packet being read. If *DSTYPE* is non-**NIL**, sets the datastream type of all subsequent packets sent on this connection, until the next call to **SPP.DSTYPE**. Since this affects the current partially-filled packet, the stream should probably be flushed (via **FORCEOUTPUT**) before this function is called.

(SPP.SENDEOM *STREAM***)** [Function]

Transmits the data buffered so far on the output stream *STREAM*, if any, with the End of Message bit set. If there is nothing buffered, sends a zero-length packet with the End of Message bit set.

(SPP.SENDATTENTION *STREAM ATTENTIONBYTE* **—)** [Function]

Sends an SPP "attention" packet on the output stream *STREAM*, with the Attention bit set and containing the single byte of data *ATTENTIONBYTE*.

Note: The appropriate way to determine whether an SPP stream is open, or whether an End of Message or Attention indication has been reached (for input streams) is to use the **EOFP** function (page 25.6). When **EOFP** is applied to an SPP stream, it returns one of the following values:

NIL The connection is open and readable or writable.

T The connection is closed.

EOM (Input streams only) The End of Message bit was set in the last packet received, and all bytes from the packet have been read. The function **SPP.CLEAREOM** (below) must be called to clear this condition.

ATTENTION (Input streams only) An attention packet is waiting. **SPP.CLEARATTENTION** (below) must be called before the single byte of data associated with the attention packet can be read.

(SPP.CLEAREOM *STREAM NOERRORFLG***)** [Function]

Clears the End of Message indication on *STREAM*. This is necessary in order to read beyond the EOM. Causes an error if the stream is not currently at the End of Message, unless *NOERRORFLG* is non-**NIL**.

(SPP.CLEARATTENTION *STREAM NOERRORFLG***)** [Function]

Clears the Attention packet indication on *STREAM*. This must be called before the single byte of data associated with the attention packet can be read. Causes an error if the stream does not have an attention packet waiting, unless *NOERRORFLG* is non-**NIL**.

31.3.5 Courier Remote Procedure Call Protocol

Courier is the Xerox Network Systems Remote Procedure Call protocol. It uses the Sequenced Packet Protocol for reliable transport. Courier uses procedure call as a metaphor for the exchange of a request from a user process and its positive reply from a server process; exceptions or error conditions are the metaphor for a negative reply. A family of remote procedures and the errors they can raise constitute a remote program. A remote program generally represents a complete service, such as the Filing or Printing programs described earlier in this chapter.

For more detail about Courier, the reader is referred to the published specification of the Courier protocol. The following documentation assumes some familiarity with the protocol. It describes how to define a Courier program and use it to communicate with a remote system element that implements a server for that program. This section does not discuss how to construct such a server.

31.3.5.1 Defining Courier Programs

A Courier program definition is accessed using the file package type **COURIERPROGRAMS**, so **GETDEF**, **PUTDEF**, and **EDITDEF** can be used to manipulate Courier programs. The file package command **COURIERPROGRAMS** (page 17.39) can be used to save Courier programs on files. Courier program are initially defined using the following function:

(COURIERPROGRAM *NAME ...***)** [NLambda NoSpread Function]

This function is used to define Courier programs. The syntax is

(COURIERPROGRAM *NAME*
  (PROGRAMNUMBER VERSIONNUMBER)
  . DEFINITIONS)

The tail *DEFINITIONS* is a property list where the properties are selected from **TYPES, PROCEDURES, ERRORS** and **INHERITS**; the values are lists of pairs of the form (*LABEL . DEFINITION*). These are described in more detail as follows:

The **TYPES** section lists the symbolically-defined types used to represent the arguments and results of procedures and errors in this Courier program. Each element in this section is of the form (*TYPENAME TYPEDEFINITION*), e.g., **(PRIORITY INTEGER)**. The *TYPEDEFINITION* can be a predefined type (see next section), another type defined in this **TYPES** section, or a qualified typename taken from another Courier program; these latter are written as a dotted pair (*PROGRAMNAME . TYPENAME*).

The **PROCEDURES** section lists the remote procedures defined by this Courier program. A procedure definition is a stylized reduction of the Courier definition syntax defined in the Courier Protocol specification:

(*PROCEDURENAME NUMBER ARGUMENTS*
RETURNS *RESULTTYPES* **REPORTS** *ERRORNAMES*)

ARGUMENTS is a list of type names, one per argument to the remote procedure, or **NIL** if the procedure takes no arguments. *RESULTTYPES* is a list of type names, one for each value to be returned. *ERRORNAMES* is a list of names of errors that can be raised by this procedure; each such error must be listed in the program's **ERRORS** section. The atoms **RETURNS** and **REPORTS** are noise words to aid readability.

The **ERRORS** section lists the errors that can be raised by procedures in this program. An error definition is of the form

(*ERRORNAME NUMBER ARGUMENTS*),

where *ARGUMENTS* is a list of type names, one for each argument, if any, reported by the error.

The **INHERITS** section is an optional list of other Courier programs, some of whose definitions are "inherited" by this program. More specifically, if a type, procedure or error referenced in the current program definition is not defined in this program, the system searches for a definition of it in each of the inherited programs in turn, and uses the first such definition found.

The **INHERITS** section is useful when defining variants of a given Courier program. For example, if one wanted to try out version 4 of Courier program **BAR**, and version 4 differed from version 3 of program **BAR** only in a small number of procedure or type definitions, one could define a program **NEWBAR** with an **INHERITS** section of **(BAR)** and only need to list the few changed definitions inside **NEWBAR**.

31.3.5.2 Courier Type Definitions

This section describes how the Courier types described in the Courier Protocol document are expressed in a Lisp Courier program definition, and how values of each type are represented. Each type in a Courier program's **TYPES** section must ultimately be defined in terms of one of the following "base" types, although the definition can be indirect through arbitrarily many levels. That is, a type can be defined in terms of any other type known by an extant Courier definition. The names of the base types are "global"; they need no qualification, nor do type names mentioned in the same Courier program. To refer to a type not defined in the same Courier program (or to any non-base type when there is no program context), one writes a *Qualified name*, in the form (*PROGRAM . TYPE*). In general, a Qualified name is legal in any place that calls for a Courier type.

31.3.5.2.1 Pre-defined Types

Pre-defined (atomic) types are expressed as uppercase litatoms from the following set:

BOOLEAN Values are represented by **T** and **NIL**.

INTEGER Values are represented as small integers in the range [-32768..32767].

CARDINAL Values are represented as small integers in the range [0..65535].

UNSPECIFIED Same as **CARDINAL**.

LONGINTEGER Values are represented as **FIXP**'s.

LONGCARDINAL Same as **LONGINTEGER**. Note that Interlisp-D does not (currently) have a datatype that truly represents a 32-bit *unsigned* integer.

STRING Values are represented as Lisp strings.

In addition, the following types not in the document have been added for convenience:

TIME Represents a date and time in accordance with the Network Time Standard. The value is a **FIXP** such as returned by the function **IDATE**, and is encoded as a **LONGCARDINAL**.

NSADDRESS Represents a network address. The value is an object of type **NSADDRESS** (page 31.7), and is encoded as six items of type **UNSPECIFIED**.

NSNAME Represents a three-part Clearinghouse name. The value is an object of type **NSNAME** (page 31.8), and is encoded as three items of type **STRING**.

NSNAME2 Represents a two-part Clearinghouse name, i.e., a domain. The value is an object of type **NSNAME** (page 31.8), and is encoded as two items of type **STRING**.

31.3.5.2.2 Constructed Types

Constructed Types are composite objects made up of elements of other types. They are all expressed as a list whose **CAR** names the type and whose remaining elements give details. The following are available:

(ENUMERATION (*NAME INDEX*) ... (*NAME INDEX*)**)**

Each *NAME* is an arbitrary litatom or string; the corresponding *INDEX* is its Courier encoding (a **CARDINAL**). Values of type **ENUMERATION** are represented as a *NAME* from the list of choices. For example, a value of type **(ENUMERATION (UNKNOWN 0) (RED 1) (BLUE 2))** might be the litatom **RED**.

(SEQUENCE *TYPE*) A **SEQUENCE** value is represented as a list, each element being of type *TYPE*. A **SEQUENCE** of length zero is represented as **NIL**. Note that there is no maximum length for a **SEQUENCE** in the Lisp implementation of Courier.

(ARRAY *LENGTH TYPE*) An **ARRAY** value is represented as a list of *LENGTH* elements, each of type *TYPE*.

(CHOICE (*NAME INDEX TYPE*) ... (*NAME INDEX TYPE*)**)**

The **CHOICE** type allows one to select among several different types at runtime; the *INDEX* is used in the encoding to distinguish the value types. A value of type **CHOICE** is represented in Lisp as a list of two elements, (*NAME VALUE*). For example, a value of type

(CHOICE (STATUS 0 (ENUMERATION (BUSY 0) (COMPLETE 1)))
(MESSAGE 1 STRING))

could be **(STATUS COMPLETE)** or **(MESSAGE "Out of paper.")**.

(RECORD (*FIELDNAME TYPE*) ... (*FIELDNAME TYPE*)**)**

Values of type **RECORD** are represented as lists, with one element for each field of the record. The field names are not part of the value, but are included for documentation purposes.

For programmer convenience, there are two macros that allow Courier records to be constructed and dissected in a manner similar to Lisp records. These compile into the appropriate composites of **CONS**, **CAR** and **CDR**.

(COURIER.CREATE *TYPE FIELDNAME* ← *VALUE* ... *FIELDNAME* ← *VALUE*) [Macro]

Creates a value of type *TYPE*, which should be a fully-qualified type name that designates a **RECORD** type, e.g., **(MAILTRANSPORT . POSTMARK)**. Each *FIELDNAME* should correspond to a field of the record, and all fields must be

included. Each *VALUE* is evaluated; all other arguments are not. The assignment arrows are for readability, and are optional.

(COURIER.FETCH *TYPE FIELD OBJECT***)** [Macro]

Analogous to the Record Package operator **fetch**. Argument *TYPE* is as with **COURIER.CREATE**; *FIELD* is the name of one of its fields. **COURIER.FETCH** extracts the indicated field from *OBJECT*. For readability, the noiseword "**of**" may be inserted between *FIELD* and *OBJECT*. Only the argument *OBJECT* is evaluated.

For example, if the program **CLEARINGHOUSE** has a type declaration

```
(USERDATA.VALUE (RECORD (LAST.NAME.INDEX CARDINAL)
        (FILE.SERVICE STRING))),
```

then the expression

```
(SETQ INFO (COURIER.CREATE
        (CLEARINGHOUSE . USERDATA.VALUE)
        LAST.NAME.INDEX ← 12
        FILE.SERVICE ← "Phylex:PARC:Xerox")
```

would set the variable **INFO** to the list (12 "**Phylex:PARC:Xerox**"). The expression

```
(COURIER.FETCH (CLEARINGHOUSE . USERDATA.VALUE)
  FILE.SERVICE of INFO)
```

would produce "**Phylex:PARC:Xerox**".

31.3.5.2.3 User Extensions to the Type Language

The programmer can add new base types to the Courier language by telling the system how to read and write values of that type. The programmer chooses a name for the type, and gives the name a **COURIERDEF** property. The new name can then be used anywhere that the type names listed in the previous sections, such as **CARDINAL**, can be used. Such extensions are useful for user-defined objects, such as datatypes, that are not naturally represented by any predefined or constructed type. The **NSADDRESS** and **NSNAME** Courier types are defined by this mechanism.

COURIERDEF [Property Name]

The format of the **COURIERDEF** property is a list of up to four elements, (*READFN WRITEFN LENGTHFN WRITEREPFN*). The first two elements are required; if the latter two are omitted, the system will simulate them as needed. The elements are as follows:

READFN This is a function of three arguments, (*STREAM PROGRAM TYPE*). The function is called by Courier when it needs to read a value of this type from *STREAM* as part of a Courier transaction. The function reads and returns the value from *STREAM*, possibly using functions such as **COURIER.READ** (page 31.25). *PROGRAM* and *TYPE* are the name of the Courier program and the type. In the case of atomic types, *TYPE* is a litatom, and is provided for type discrimination in case the programmer has supplied a single reading function for several different types. In the case of constructed types, *TYPE* is a list, **CAR** of which is the type name.

WRITEFN This is a function of four arguments, (*STREAM VALUE PROGRAM TYPE*). The function is called by Courier when it needs to write *VALUE* to *STREAM*. *PROGRAM* and *TYPE* are as with the reading function. The function should write *VALUE* on *STREAM*. The result returned from this function is ignored.

LENGTHFN This function is called when Courier wants to write a value of this type in the form **(SEQUENCE UNSPECIFIED)**, and then only if the *WRITEREPFN* is omitted. The function is of three arguments, (*VALUE PROGRAM TYPE*). It should return, as an integer, the number of 16-bit words that the *WRITEFN* would require to write out this value. If values of this type are all the same length, the *LENGTHFN* can be a simple integer instead of a function. See discussion of **COURIER.WRITE.SEQUENCE.UNSPECIFIED** (page 31.26.

WRITEREPFN This function is called when Courier wants to write a value of this type in the form **(SEQUENCE UNSPECIFIED)**. The function takes the same arguments as the *WRITEFN*, but must write the value to the stream preceded by its length. If this function is omitted, Courier invokes the *LENGTHFN* to find out how long the value is, and then invokes the *WRITEFN*. If the *LENGTHFN* is omitted, Courier invokes the *WRITEFN* on a scratch stream to find out how long the value is.

31.3.5.3 Performing Courier Transactions

The normal use of Courier is to open a connection with a remote system element using **COURIER.OPEN**, perform one or more remote procedure calls using **COURIER.CALL**, then close the connection with **CLOSEF**.

(COURIER.OPEN *HOSTNAME SERVERTYPE NOERRORFLG NAME WHENCLOSEDFN OTHERPROPS***)** [Function]

Opens a Courier connection to the Courier socket on *HOST*, and returns an SPP stream that can be passed to **COURIER.CALL**. *HOSTNAME* can be an NS address, or a symbolic Clearinghouse name in the form of a string, litatom or **NSNAME**. In the case of a symbolic name, *SERVERTYPE* specifies the Clearinghouse

property under which the server's address may be found; normally, this is **NIL**, in which case the **ADDRESS.LIST** property is used.

Normally, if a connection cannot be made, or the server supports the wrong version of Courier, an error occurs. If *NOERRORFLG* is non-**NIL**, **COURIER.OPEN** returns **NIL** in these cases.

If *NAME* is non-**NIL**, it is used as the name of the Courier connection process.

WHENCLOSEDFN is a function (or list of functions) of one argument, the Courier stream, that will be called when the connection is closed, either by user or server.

If *OTHERPROPS* is non-**NIL**, it should be a property list of SPP stream properties, as accepted by **SPP.OPEN** (page 31.12). Any **CLOSEFN** property on this list is overridden by the value of *WHENCLOSEDFN*.

(COURIER.CALL *STREAM PROGRAM PROCEDURE* ARG_1 ... ARG_N *NOERRORFLG***)** [NoSpread Function]

This function calls the remote procedure *PROCEDURE* of the Courier program *PROGRAM*. *STREAM* is the stream returned by **COURIER.OPEN**. The arguments should be Lisp values appropriate for the Courier types of the corresponding formal parameters of the procedure. There must be the same number of actual and formal arguments. If the procedure call is successful, Courier returns the result(s) of the call as specified in the **RETURNS** section of the procedure definition. If there is only a single result, it is returned directly, otherwise a list of results is returned.

Procedures that take a Bulk Data argument (source or sink) are treated specially; see page 31.24.

If the procedure call results in an error, one of three possible courses is available. The default behavior is to cause a Lisp error. To suppress the error, an optional keyword can be appended to the argument list, as if an extra argument. This *NOERRORFLG* argument can be the atom **NOERROR**, in which case **NIL** is returned as the result of the call. If *NOERRORFLG* is **RETURNERRORS**, the result of the call is a list **(ERROR** *ERRORNAME . ERRORARGS***)**. If the failure was a Courier Reject, rather than Error, then *ERRORNAME* is the atom **REJECT**.

Examples:

```
(COURIERPROGRAM PERSONNEL (17 1)
  TYPES
  ((PERSON.NAME (RECORD (FIRST.NAME STRING)
                        (MIDDLE MIDDLE.PART)
                        (LAST.NAME STRING)))
```

```
  (MIDDLE.PART (CHOICE (NAME 0 STRING)
                       (INITIAL 1 STRING)))
  (BIRTHDAY (RECORD (YEAR CARDINAL)
                    (MONTH STRING)
                    (DAY CARDINAL))))
  PROCEDURES
  ((GETBIRTHDAY 3 (PERSON.NAME)
      RETURNS (BIRTHDAY) REPORTS (NO.SUCH.PERSON)))
  ERRORS
  ((NO.SUCH.PERSON 1))
)
```

This expression defines **PERSONNEL** to be Courier program number 17, version number 1. The example defines three types, **PERSON.NAME**, **MIDDLE.PART** and **BIRTHDAY**, and one procedure, **GETBIRTHDAY**, whose procedure number is 3. The following code could be used to call the remote **GETBIRTHDAY** procedure on the host with address **HOSTADDRESS**.

```
(SETQ STREAM (COURIER.OPEN HOSTADDRESS))
(PROG1 (COURIER.CALL STREAM 'PERSONNEL 'GETBIRTHDAY
          (COURIER.CREATE (PERSONNEL . PERSON.NAME)
             FIRST.NAME ← "Eric"
             MIDDLE ← '(INITIAL "C")
             LAST.NAME ← "Cooper"))
   (CLOSEF STREAM))
```

COURIER.CALL in this example might return a value such as **(1959 "January" 10)**.

31.3.5.3.1 Expedited Procedure Call

Some Courier servers support "Expedited Procedure Call", which is a way of performing a single Courier transaction by a Packet Exchange protocol, rather than going to the expense of setting up a full Courier connection. Expedited calls must have no bulk data arguments, and their arguments and results must each fit into a single packet.

(COURIER.EXPEDITED.CALL *ADDRESS SOCKET# PROGRAM PROCEDURE* ARG_1 ... ARG_N *NOERRORFLG***)** [NoSpread Function]

Attempts to perform a Courier call using the Expedited Procedure Call. *ADDRESS* is the NS address of the remote host and *SOCKET#* is the socket on which it is known to listen for expedited calls. The remaining arguments are exactly as with **COURIER.CALL**. If the arguments to the procedure do not fit in one packet, or if there is no response to the call, or if the call returns the error **USE.COURIER** (which must be defined by

exactly that name in *PROGRAM*), then the call is attempted instead by the normal, non-expedited method—a Courier connection is opened with *ADDRESS*, and **COURIER.CALL** is invoked on the arguments given.

31.3.5.3.2 Expanding Ring Broadcast

"Expanding Ring Broadcast" is a method of locating a server of a particular type whose address is not known in advance. The system broadcasts some sort of request packet on the directly-connected network, then on networks one hop away, then on networks two hops away, etc., until a positive response is received.

For use in locating a server for a particular Courier program, a stylized form of Expanding Ring Broadcast is defined. The request packet is essentially the call portion of an Expedited Procedure Call for some procedure defined in the program. The response packet is a Courier response, and typically contains at least the server's address as the result of the call. The designer of the protocol must, of course, specify which procedure to use in the broadcast (usually it is procedure number zero) and on what socket the server should listen for broadcasts.

START.CLEARINGHOUSE uses this procedure to locate the nearest Clearinghouse server.

(COURIER.BROADCAST.CALL *DESTSOCKET# PROGRAM PROCEDURE ARGS RESULTFN NETHINT MESSAGE***)** [Function]

Performs an expanding ring broadcast for servers willing to implement *PROCEDURE* in Courier program *PROGRAM*. *DESTSOCKET#* is the socket on which such servers of this type are known to listen for broadcasts, typically the same socket on which they listen for expedited calls. *ARGS* is the argument list, if any, to the procedure (note that it is not spread, unlike with **COURIER.CALL**).

If a host responds positively, then the function *RESULTFN* is called with one argument, the Courier results of the procedure call. If *RESULTFN* returns a non-null value, the value is returned as the value of **COURIER.BROADCAST.CALL** and the search stops there; otherwise, the search for a responsive host continues. If *RESULTFN* is not supplied (or is **NIL**), then the results of the procedure call are returned directly from **COURIER.BROADCAST.CALL**; i.e., *RESULTFN* defaults to the identity function.

NETHINT, if supplied, is a net number or list of net numbers as a hint concerning which net(s) to try first before performing a pure expanding-ring broadcast. If *MESSAGE* is non-NIL, it is a description (string) of what the broadcast is looking for, to be

printed in the prompt window to inform the user of what is happening. For example, **START.CLEARINGHOUSE** passes in the message **"Clearinghouse servers"** and the hint **CH.NET.HINT**.

31.3.5.3.3 Using Bulk Data Transfer

When a Courier program needs to transfer an arbitrary amount of information as an argument or result of a Courier procedure, the procedure is usually defined to have one argument of type "Bulk Data". The argument is a "source" if it is information transferred from caller to server (as though a procedure argument), a "sink" if it is information transferred from server to caller (as though a procedure result). These two "types" are indicated in a Courier procedure's formal argument list as **BULK.DATA.SOURCE** and **BULK.DATA.SINK**, respectively. A Courier procedure may have at most one such argument.

In a Courier call, the bulk data is transmitted in a special way, between the arguments and the results. There are two basic ways to handle this in the call. The caller can specify how the bulk data is to be interpreted (how to read or write it), or the caller can request to be given a bulk data stream as the result of the Courier call. The former is the preferred way; both are described below.

In the first method, the caller passes as the actual argument to the Courier call (i.e., in the position in the argument list occupied by **BULK.DATA.SOURCE** or **BULK.DATA.SINK**) a function to perform the transfer. Courier sets up the transaction, then calls the supplied function with one argument, a stream on which to write (if a source argument) or read (if a sink) the bulk data. If the function returns normally, the Courier transaction proceeds as usual; if it errors out, Courier sends a Bulk Data Abort to abort the transaction.

In the case of a sink argument, if the value returned from the sink function is non-**NIL**, it is returned as the result of **COURIER.CALL**; otherwise, the result of **COURIER.CALL** is the usual procedure result, as declared in the Courier program.

For convenience, a Bulk Data sink argument to a Courier call can be specified as a fully qualified Courier type, e.g., **(CLEARINGHOUSE . NAME)**, in which case the Bulk Data stream is read as a "stream of" that type (see **COURIER.READ.BULKDATA**, below).

The second method for handling bulk data is to pass **NIL** as the bulk data "argument" to **COURIER.CALL**. In this case, Courier sets up the call, then returns a stream that is open for **OUTPUT** (if a source argument) or **INPUT** (if a sink). The caller is responsible for transferring the bulk data on the stream, then closing the stream to complete the transaction. The value returned from

CLOSEF is the Courier result. This method is required if the caller's control structure is open-ended in a way such that the bulk data cannot be transferred within the scope of the call to **COURIER.CALL**.

In either method, the stream on which the bulk data is transferred is a standard Interlisp stream, so **BIN**, **BOUT**, **COPYBYTES** are all appropriate.

Many Courier programs define a "Stream of <type>" as a means of transferring an arbitrary number of objects, all of the same type. Although this is typically specified formally in the printed Courier documentation as a recursive definition, the recursion is in practice unnecessary and unwieldy; instead, the following function should be used.

(COURIER.READ.BULKDATA *STREAM PROGRAM TYPE DONTCLOSE***)** [Function]

Reads from *STREAM* a "Stream of *TYPE*" for Courier program *PROGRAM*, and returns a list of the objects read. *STREAM* is closed on exit, unless *DONTCLOSE* is non-**NIL**.

Passing **(X . Y)** as the bulk argument to a Courier call is thus equivalent to passing the function **(LAMBDA (STREAM) (COURIER.READ.BULKDATA STREAM X Y))**.

31.3.5.3.4 Courier Subfunctions for Data Transfer

The following functions are of interest to those who transfer data in Courier representations, e.g., as part of a function to implement a user-defined Courier type.

(COURIER.READ *STREAM PROGRAM TYPE***)** [Function]

Reads from the stream *STREAM* a Courier value of type *TYPE* for program *PROGRAM*. If *TYPE* is a predefined type, then *PROGRAM* is irrelevant; otherwise, it is required in order to qualify *TYPE*.

(COURIER.WRITE *STREAM ITEM PROGRAM TYPE***)** [Function]

Writes *ITEM* to the stream *STREAM* as a Courier value of type *TYPE* for program *PROGRAM*.

(COURIER.READ.SEQUENCE *STREAM PROGRAM TYPE***)** [Function]

Reads from the stream *STREAM* a Courier value **SEQUENCE** of values of type *TYPE* for program *PROGRAM*. Equivalent to **(COURIER.READ** *STREAM PROGRAM* **(SEQUENCE** *TYPE***))**.

(COURIER.WRITE.SEQUENCE *STREAM ITEM PROGRAM TYPE***)** [Function]

Equivalent to **(COURIER.WRITE** *STREAM ITEM PROGRAM* **(SEQUENCE** *TYPE***))**.

Some Courier programs traffic in values whose interpretation is left up to the clients of the program; the values are transferred in Courier transactions as values of type **(SEQUENCE UNSPECIFIED)**. For example, the Clearinghouse program transfers the value of a database property as an uninterpreted sequence, leaving it up to the caller, who knows what type of value the particular property takes, to interpret the sequence of raw bits as some other Courier representation. The following functions are useful when dealing with such values.

(COURIER.WRITE.REP *VALUE PROGRAM TYPE***)** [Function]

Produces a list of 16-bit integers, i.e., a value of type **(SEQUENCE UNSPECIFIED)**, that represents *VALUE* when interpreted as a Courier value of type *TYPE* in *PROGRAM*. Examples:

(COURIER.WRITE.REP T NIL 'BOOLEAN) = > (1)

(COURIER.WRITE.REP "Thing" NIL 'STRING) = >
(5 52150Q 64556Q 63400Q)

(COURIER.WRITE.REP '(10 25) NIL '(SEQUENCE INTEGER)) = >
(2 10 25)

(COURIER.READ.REP *LIST.OF.WORDS PROGRAM TYPE***)** [Function]

Interprets *LIST.OF.WORDS*, a list of 16-bit integers, as a Courier object of type *TYPE* in the Courier program *PROGRAM*.

(COURIER.WRITE.SEQUENCE.UNSPECIFIED *STREAM ITEM PROGRAM TYPE***)** [Function]

Writes to the stream *STREAM* in the form **(SEQUENCE UNSPECIFIED)** the object *ITEM*, whose value is really a Courier value of type *TYPE* for program *PROGRAM*. Equivalent to, but usually much more efficient than, **(COURIER.WRITE** *STREAM* **(COURIER.WRITE.REP** *ITEM PROGRAM TYPE***) NIL '(SEQUENCE UNSPECIFIED))**.

31.4 Level One Ether Packet Format

The data type **ETHERPACKET** is the vehicle for all kinds of packets transmitted on an Ethernet or Experimental Ethernet. An **ETHERPACKET** contains several fields for use by the Ethernet drivers and a large, contiguous data area making up the data of the level zero packet. The first several words of the area are

reserved for the level one to zero encapsulation, and the remainder (starting at field **EPBODY**) make up the level one packet. Typically, each level one protocol defines a **BLOCKRECORD** (page 8.11) that overlays the **ETHERPACKET** starting at the **EPBODY** field, describing the format of a packet for that particular protocol. For example, the records **PUP** and **XIP** define the format of level one packets in the PUP and NS protocols.

The extra fields in the beginning of an **ETHERPACKET** have mostly a fixed interpretation over all protocols. Among the interesting ones are:

EPLINK A pointer used to link packets, used by the **SYSQUEUE** mechanism (page 31.41). Since this field is used by the system for maintaining the free packet queue and ether transmission queues, do not use this field unless you understand it.

EPFLAGS A byte field that can be used for any purpose by the user.

EPUSERFIELD A pointer field that can be used for any purpose by the user. It is set to **NIL** when a packet is released.

EPTRANSMITTING A flag that is true while the packet is "being transmitted", i.e., from the time that the user instructs the system to transmit the packet until the packet is gathered up from the transmitter's finished queue. While this flag is true, the user must *not* modify the packet.

EPREQUEUE A pointer field that specifies the desired disposition of the packet after transmission. The possible values are: **NIL** means no special treatment; **FREE** means the packet is to be released after transmission; an instance of a **SYSQUEUE** means the packet is to be enqueued on the specified queue (page 31.41).

The normal life of an outgoing Ether packet is that a program obtains a blank packet, fills it in according to protocol, then sends the packet over the Ethernet. If the packet needs to be retained for possible retransmission, the **EPREQUEUE** field is used to specify a queue to place the packet on after its transmission, or the caller hangs on to the packet explicitly.

There are redefinitions, or "overlays" of the **ETHERPACKET** record specifically for use with the PUP and NS protocols. The following sections describe those records and the handling of the PUP and NS level one protocols, how to add new level one protocols, and the queueing mechanism associated with the **EPREQUEUE** field.

31.5 PUP Level One Functions

The functions in this section are used to implement level two and higher PUP protocols. That is, they deal with sending and receiving PUP packets. It is assumed the reader is familiar with the format and use of pups, e.g., from reading reference [3] on page 31.5.

31.5.1 Creating and Managing Pups

There is a record **PUP** that overlays the data portion of an **ETHERPACKET** and describes the format of a pup. This record defines the following numeric fields: **PUPLENGTH** (16 bits), **TCONTROL** (transmit control, 8 bits, cleared when a PUP is transmitted), **PUPTYPE** (8 bits), **PUPID** (32 bits), **PUPIDHI** and **PUPIDLO** (16 bits each overlaying **PUPID**), **PUPDEST** (16 bits overlayed by 8-bit fields **PUPDESTNET** and **PUPDESTHOST**), **PUPDESTSOCKET** (32 bits, overlayed by 16-bit fields **PUPDESTSOCKETHI** and **PUPDESTSOCKETLO**), and **PUPSOURCE**, **PUPSOURCENET**, **PUPSOURCEHOST**, **PUPSOURCESOCKET**, **PUPSOURCESOCKETHI**, and **PUPSOURCESOCKETLO**, analagously. The field **PUPCONTENTS** is a pointer to the start of the data portion of the pup.

(ALLOCATE.PUP) [Function]

Returns a (possibly used) pup. Keeps a free pool, creating new pups only when necessary. The pup header fields of the pup returned are guaranteed to be zero, but there may be garbage in the data portion if the pup had been recycled, so the caller should clear the data if desired.

(CLEARPUP *PUP***)** [Function]

Clears *all* information from *PUP*, including the pointer fields of the **ETHERPACKET** and the pup data portion.

(RELEASE.PUP *PUP***)** [Function]

Releases *PUP* to the free pool.

31.5.2 Sockets

Pups are sent and received on a *socket*. Generally, for each "conversation" between one machine and another, there is a distinct socket. When a pup arrives at a machine, the low-level pup software examines the pup's destination socket number. If there is a socket on the machine with that number, the incoming pup is handed over to the socket; otherwise the incoming pup is

discarded. When a *user* process initiates a conversation, it generally selects a large, random socket number different from any other in use on the machine. A *server* process, on the other hand, provides a specific service at a "well-known" socket, usually a fairly small number. In the PUP world, advertised sockets are in the range 0 to 100Q.

(OPENPUPSOCKET *SKT# IFCLASH***)** [Function]

Opens a new pup socket. If *SKT#* is **NIL** (the normal case), a socket number is chosen automatically, guaranteed to be unique, and probably different from any socket opened this way in the last 18 hours (the low half of the time of day clock is sampled).

If a specific local socket is desired, as is typically the case when implementing a server, *SKT#* is given, and must be a (up to 32-bit) number. *IFCLASH* indicates what to do in the case that the designated socket is already in use: if **NIL**, an error is generated; if **ACCEPT**, the socket is quietly returned; if **FAIL**, then **OPENPUPSOCKET** returns **NIL** without causing an error. Note that "well-known" socket numbers should be avoided unless the caller is actually implementing one of the services advertised as provided at the socket.

(CLOSEPUPSOCKET *PUPSOC NOERRORFLG***)** [Function]

Closes and releases socket *PUPSOC*. If *PUPSOC* is **T**, closes all pup sockets (this must be used with caution, since it will also close system sockets!). If *PUPSOC* is already closed, an error is generated unless *NOERRORFLG* is true.

(PUPSOCKETNUMBER *PUPSOC***)** [Function]

Returns the socket number (a 32-bit integer) of *PUPSOC*.

(PUPSOCKETEVENT *PUPSOC***)** [Function]

Returns the **EVENT** of *PUPSOC* (page 23.7). This event is notified whenever a pup arrives on *PUPSOC*, so pup clients can perform an **AWAIT.EVENT** on this event if they have nothing else to do at the moment.

31.5.3 Sending and Receiving Pups

(SENDPUP *PUPSOC PUP***)** [Function]

Sends *PUP* on socket *PUPSOC*. If any of the **PUPSOURCESHOST**, **PUPSOURCENET**, or **PUPSOURCESOCKET** fields is zero, **SENDPUP** fills them in using the pup address of this machine and/or the socket number of *PUPSOC*, as needed.

(GETPUP *PUPSOC WAIT***)** [Function]

Returns the next pup that has arrived addressed to socket *PUPSOC*. If there are no pups waiting on *PUPSOC*, then **GETPUP** returns **NIL**, or waits for a pup to arrive if *WAIT* is **T**. If *WAIT* is an integer, **GETPUP** interprets it as a number of milliseconds to wait, finally returning **NIL** if a pup does not arrive within that time.

(DISCARDPUPS *SOC***)** [Function]

Discards without examination any pups that have arrived on *SOC* and not yet been read by a **GETPUP**.

(EXCHANGEPUPS *SOC OUTPUP DUMMY IDFILTER TIMEOUT***)** [Function]

Sends *OUTPUP* on *SOC*, then waits for a responding pup, which it returns. If *IDFILTER* is true, ignores pups whose **PUPID** is different from that of *OUTPUP*. *TIMEOUT* is the length of time (msecs) to wait for a response before giving up and returning **NIL**. *TIMEOUT* defaults to **\ETHERTIMEOUT**. **EXCHANGEPUPS** discards without examination any pups that are currently waiting on *SOC* before *OUTPUP* gets sent. (*DUMMY* is ignored; it exists for compatibility with an earlier implementation).

31.5.4 Pup Routing Information

Ordinarily, a program calls **SENDPUP** and does not worry at all about the route taken to get the pup to its destination. There is an internet routing process in Lisp whose job it is to maintain information about the best routes to networks of interest. However, there are some algorithms for which routing information and/or the topology of the net are explicitly desired. To this end, the following functions are supplied:

(PUPNET.DISTANCE *NET#***)** [Function]

Returns the "hop count" to network *NET#*, i.e., the number of gateways through which a pup must pass to reach *NET#*, according to the best routing information known at this point. The local (directly-connected) network is considered to be zero hops away. Current convention is that an inaccessible network is 16 hops away. **PUPNET.DISTANCE** may need to wait to obtain routing information from an Internetwork Router if *NET#* is not currently in its routing cache.

(SORT.PUPHOSTS.BY.DISTANCE *HOSTLIST***)** [Function]

Sorts *HOSTLIST* by increasing distance, in the sense of **PUPNET.DISTANCE**. *HOSTLIST* is a list of lists, the **CAR** of each list being a 16-bit Net/Host address, such as returned by

ETHERHOSTNUMBER. In particular, a list of ports ((nethost . socket) pairs) is in this format.

(PRINTROUTINGTABLE *TABLE SORT FILE*) [Function]

Prints to *FILE* the current routing cache. The table is sorted by network number if *SORT* is true. *TABLE* = **PUP** (the default) prints the PUP routing table; *TABLE* = **NS** prints the NS routing table.

31.5.5 Miscellaneous PUP Utilities

(SETUPPUP *PUP DESTHOST DESTSOCKET TYPE ID SOC REQUEUE*) [Function]

Fills in various fields in *PUP*'s header: its length (the header overhead length; assumes data length of zero), *TYPE*, *ID* (if *ID* is **NIL**, generates a new one itself from an internal 16-bit counter), destination host and socket (*DESTHOST* may be anything that **ETHERPORT** accepts; an explicit nonzero socket in *DESTHOST* overrides *DESTSOCKET*). If *SOC* is not supplied, a new socket is opened. *REQUEUE* fills the packets **EPREQUEUE** field (see above). Value of **SETUPPUP** is the socket.

(SWAPPUPPORTS *PUP*) [Function]

Swaps the source and destination addresses in *PUP*. This is useful in simple packet exchange protocols, where you want to respond to an input packet by diddling the data portion and then sending the pup back whence it came.

(GETPUPWORD *PUP WORD#*) [Function]

Returns as a 16-bit integer the contents of the *WORD#*th word of *PUP*'s data portion, counting the first word as word zero.

(PUTPUPWORD *PUP WORD# VALUE*) [Function]

Stores 16-bit integer *VALUE* in the *WORD#*th word of *PUP*'s data portion.

(GETPUPBYTE *PUP BYTE#*) [Function]

Returns as an integer the contents of the *BYTE#*th 8-bit byte of *PUP*'s data portion, counting the first byte as byte zero.

(PUTPUPBYTE *PUP BYTE# VALUE*) [Function]

Stores *VALUE* in the *BYTE#*th 8-bit byte of *PUP*'s data portion.

(GETPUPSTRING *PUP OFFSET*) [Function]

Returns a string consisting of the characters in *PUP*'s data portion starting at byte *OFFSET* (default zero) through the end of *PUP*.

(PUTPUPSTRING *PUP STR*) [Function]

Appends *STR* to the data portion of *PUP*, incrementing *PUP*'s length appropriately.

31.5.6 PUP Debugging Aids

Tracing facilities are provided to allow the user to see the pup traffic that passes through **SENDPUP** and **GETPUP**. The tracing can be verbose, displaying much information about each packet, or terse, which shows a concise "picture" of the traffic.

PUPTRACEFLG [Variable]

Controls tracing information provided by **SENDPUP** and **GETPUP**. Legal values:

NIL No tracing.

T Every **SENDPUP** and every successful **GETPUP** call **PRINTPUP** of the pup at hand (see below).

PEEK Allows a concise "picture" of the traffic. For normal, non-broadcast packets, **SENDPUP** prints "**!**", **GETPUP** prints "**+**". For broadcast packets, **SENDPUP** prints "↑", **GETPUP** prints "*****". In addition, for packets that arrive not addressed to any socket on this machine (e.g., broadcast packets for a service not implemented on this machine), a "**&**" is printed.

PUPIGNORETYPES [Variable]

A list of pup types (small integers). If the type of a pup is on this list, then **GETPUP** and **SENDPUP** will not print the pup verbosely, but treat it as though **PUPTRACEFLG** were **PEEK**. This allows the user to filter out "uninteresting" pups, e.g., routine routing information pups (type 201Q).

PUPONLYTYPES [Variable]

A list of pup types. If this variable is non-**NIL**, then **GETPUP** and **SENDPUP** print verbosely *only* pups whose types appear on the list, treating others as though **PUPTRACEFLG** were **PEEK**. This lets the tracing be confined to only a certain class of pup traffic.

PUPTRACEFILE [Variable]

The file to which pup tracing output is sent by default. The file must be open. **PUPTRACEFILE** is initially **T**.

PUPTRACETIME [Variable]

If this variable is true, then each printout of a pup is accompanied by a relative timestamp (in seconds, with 2 decimal places) of the current time (i.e., when the **SENDPUP** or **GETPUP** was called; for incoming pups, this is not the same as when the pup actually arrived).

(PUPTRACE *FLG REGION***)** [Function]

Creates a window for puptracing, and sets **PUPTRACEFILE** to it. If **PUPTRACEFILE** is currently a window and *FLG* is **NIL**, closes the window. Sets **PUPTRACEFLG** to be *FLG*. If *REGION* is supplied, the window is created with that region. The window's **BUTTONEVENTFN** is set to cycle **PUPTRACEFLG** through the values **NIL**, **T**, and **PEEK** when the mouse is clicked in the window.

(PRINTPUP *PACKET CALLER FILE PRE.NOTE DOFILTER***)** [Function]

Prints the information in the header and possibly data portions of pup *PACKET* to *FILE*. If *CALLER* is supplied, it identifies the direction of the pup (**GET** or **PUT**), and is printed in front of the header. *FILE* defaults to **PUPTRACEFILE**. If *PRE.NOTE* is non-**NIL**, it is **PRIN1**'ed first. If *DOFILTER* is true, then if *PUP*'s type fails the filtering criteria of **PUPIGNORETYPES** or **PUPONLYTYPES**, then *PUP* is printed "tersely", i.e., as a !, +, ↑, or *, as described above.

GETPUP and **SENDPUP**, when **PUPTRACEFLG** is non-**NIL**, call **(PRINTPUP** *PUP* {**'GET** or **'PUT}** **NIL NIL T)**.

The form of printing provided by **PRINTPUP** can be influenced by adding elements to **PUPPRINTMACROS**.

PUPPRINTMACROS [Variable]

An association list of elements (*PUPTYPE* . *MACRO*) for printing pups. The *MACRO* (**CDR** of each element) tells how to print the information in a pup of type *PUPTYPE* (**CAR** of the element). If *MACRO* is a litatom, then it is a function of two arguments (*PUP FILE*) that is applied to the pup to do the printing. Otherwise, *MACRO* is a list describing how to print the data portion of the pup (the header is printed in a standard way).

The list form of *MACRO* consists of "commands" that specify a "datatype" to interpret the data, and an indication of how far that datatype extends in the packet. Each element of *MACRO* is one of the following: (a) a byte offset (positive integer), indicating the byte at which the next element, if any, takes effect; (b) a negative integer, the absolute value of which is the number of bytes until the next element, if any, takes effect; or (c) an atom giving the format in which to print the data, one of the following:

BYTES Print the data as 8-bit bytes, enclosed in brackets. This is the default format to start with.

CHARS Print the data as (8-bit) characters. Non-printing characters are printed as if the format were **BYTES**, except that the sequence 15Q, 12Q is printed specially as [crlf].

WORDS Print the data as 16-bit integers, separated by commas (or the current **SEPR**).

INTEGERS Print the data as 32-bit integers, separated by commas (or the current **SEPR**). Note: the singular **BYTE, CHAR, WORD, INTEGER** are accepted as synonyms for these four commands.

SEPR Set the separator for **WORDS** and **INTEGERS** to be the next element of the macro. The separator is initially the two characters, comma, space.

IFSSTRING Interprets the data as a 16-bit length followed by that many 8-bit bytes or characters. If the current datatype is **BYTES**, leaves it alone; otherwise, sets it to be **CHARS**.

... If there is still data left in the packet by the time processing reaches this command, prints "..." and stops.

FINALLY The next element of the macro is printed when the end of the packet is reached (or printing stops because of a ...). This command does not alter the datatype, and can appear anywhere in the macro as long as it is encountered before the actual end of the packet.

T Perform a **TERPRI**.

REPEAT The remainder of the macro is itself treated as a macro to be applied over and over until the packet is exhausted. Note that the offsets specified in the macro must be in the relative form, i.e., negative integers. For example, the macro **(INTEGERS 4 REPEAT BYTES -2 WORDS -4)** says to print the first 4 bytes of the data as one 32-bit integer, then print the rest of the data as sets of 2 8-bit bytes and 2 16-bit words.

Only as much of the macro is processed as is needed to print the data in the given packet. The default macro for printing a pup is **(BYTES 12 ...)**, meaning to print the first up to 12 bytes as bytes, and then print "..." if there is anything left.

(PUP.ECHOUSER *HOST ECHOSTREAM INTERVAL NTIMES***)** [Function]

Sends dummy packets to be echoed by the host *HOST*. Can be used as a simple test of the functioning of the Ethernet and the host.

HOST is the pup host to send the packets to. *ECHOSTREAM* is the stream for printing status information. *INTERVAL* is the interval (in milliseconds) to wait for the packet to be echoed (default 1000). *NTIMES* is the number of packets to send (default 1000).

As each packet is sent and received, characters are printed to *ECHOSTREAM* as follows:

! Printed when a packet is sent.

+ Printed when an echo packet is sucessfully received.

. Printed when an echo packet has not been received after *INTERVAL* milliseconds.

? Printed when a packet is received, but it isn't an echo packet or an error packet.

(late) Printed when an error packet is received, after the echo request timed out.

The trace can be used to test the functioning of the ethernet and host. For example, if the trace is **!+!+!+!+!+**, the host is listening and echoing correctly. **!.!.!.!.!.** indicates that for some reason the host is not responding. **!+!.!.!(late).!(late)(late)+** indicates that the packets are being echoed, but not immediately.

The following functions are used by **PRINTPUP** and similar functions, and may be of interest in special cases.

(PORTSTRING *NETHOST SOCKET***)** [Function]

Converts the pup address *NETHOST, SOCKET* into octal string format as follows: *NET#HOST#SOCKET*. *NETHOST* may be a port (dotted pair of nethost and socket), in which case *SOCKET* is ignored, and the socket portion of *NETHOST* is omitted from the string if it is zero.

(PRINTPUPROUTE *PACKET CALLER FILE***)** [Function]

Prints the source and destination addresses of pup *PACKET* to *FILE* in the **PORTSTRING** format, preceded by *CALLER* (interpreted as with **PRINTPUP**).

(PRINTPACKETDATA *BASE OFFSET MACRO LENGTH FILE***)** [Function]

Prints data according to *MACRO*, which is a list interpreted as described under **PUPPRINTMACROS**, to *FILE*. The data starts at *BASE* and extends for *LENGTH* bytes. The actual printing starts at the *OFFSET*th byte, which defaults to zero. For example, **PRINTPUP** ordinarily calls **(PRINTPACKETDATA (fetch PUPCONTENTS of** *PUP***) 0** *MACRO* **(IDIFFERENCE (fetch PUPLENGTH of** *PUP***) 20)** *FILE***)**.

(PRINTCONSTANT *VAR CONSTANTLIST FILE PREFIX***)** [Function]

CONSTANTLIST is a list of pairs (*VARNAME VALUE*), of the form given to the **CONSTANTS** File Package Command. **PRINTCONSTANT** prints *VAR* to *FILE*, followed in parentheses by

the *VARNAME* out of *CONSTANTLIST* whose *VALUE* is **EQ** to *VAR*, or ? if it finds no such element. If *PREFIX* is non-**NIL** and is an initial substring of the selected *VARNAME*, then *VARNAME* is printed without the prefix.

For example, if **FOOCONSTANTS** is **((FOO.REQUEST 1) (FOO.ANSWER 2) (FOO.ERROR 3))**, then **(PRINTCONSTANT 2 FOOCONSTANTS T "FOO.")** produces "**2 (ANSWER)**".

(OCTALSTRING *N***)** [Function]

Returns a string of octal digits representing *N* in radix 8.

31.6 NS Level One Functions

The functions in this section are used to implement level two and higher NS protocols. The packets used in the NS protocol are termed Xerox Internet Packets (XIPs). The functions for manipulating XIPs are similar to those for managing PUPs, so will be described in less detail here. The major difference is that NS host addresses are 48-bit numbers. Since Interlisp-D cannot currently represent 48-bit numbers directly as integers, there is an interim form called **NSHOSTNUMBER**, which is defined as a **TYPERECORD** of three fields, each of them being a 16-bit portion of the 48-bit number.

31.6.1 Creating and Managing XIPs

There is a record **XIP** that overlays the data portion of an **ETHERPACKET** and describes the format of a XIP. This record defines the following fields: **XIPLENGTH** (16 bits), **XIPTCONTROL** (transmit control, 8 bits, cleared when a XIP is transmitted), **XIPTYPE** (8 bits), **XIPDESTNET** (32 bits), **XIPDESTHOST** (an **NSHOSTNUMBER**), **XIPDESTSOCKET** (16 bits), and **XIPSOURCENET**, **XIPSOURCEHOST**, and **XIPSOURCESOCKET**, analagously. The field **XIPCONTENTS** is a pointer to the start of the data portion of the XIP.

(ALLOCATE.XIP) [Function]

Returns a (possibly used) XIP. As with **ALLOCATE.PUP**, the header fields are guaranteed to be zero, but there may be garbage in the data portion if the pup had been recycled.

(RELEASE.XIP *XIP***)** [Function]

Releases *XIP* to the free pool.

31.6.2 NS Sockets

As with pups, XIPs are sent and received on a *socket*. The same comments apply as with pup sockets (page 31.29), except that NS socket numbers are only 16 bits.

(OPENNSOCKET *SKT# IFCLASH***)** [Function]

Opens a new NS socket. If *SKT#* is **NIL** (the normal case), a socket number is chosen automatically, guaranteed to be unique, and probably different from any socket opened this way in the last 18 hours. If a specific local socket is desired, as is typically the case when implementing a server, *SKT#* is given, and must be a (up to 16-bit) number. *IFCLASH* governs what to do if *SKT#* is already in use: if *IFCLASH* is **NIL**, an error is generated; if *IFCLASH* is **ACCEPT**, the socket is quietly returned; if *IFCLASH* is **FAIL**, then **OPENNSOCKET** returns **NIL** without causing an error.

(CLOSENSOCKET *NSOC NOERRORFLG***)** [Function]

Closes and releases socket *NSOC*. If *NSOC* is **T**, closes all NS sockets (this must be used with caution, since it will also close system sockets!). If *NSOC* is already closed, an error is generated unless *NOERRORFLG* is true.

(NSOCKETNUMBER *NSOC***)** [Function]

Returns the socket number (a 16-bit integer) of *NSOC*.

(NSOCKETEVENT *NSOC***)** [Function]

Returns the **EVENT** of *NSOC*. This event is notified whenever a XIP arrives on *NSOC*.

31.6.3 Sending and Receiving XIPs

(SENDXIP *NSOC XIP***)** [Function]

Sends *XIP* on socket *NSOC*. If any of the **XIPSOURCESHOST**, **XIPSOURCENET**, or **XIPSOURCESOCKET** fields is zero, **SENDXIP** fills them in using the NS address of this machine and/or the socket number of *NSOC*, as needed.

(GETXIP *NSOC WAIT***)** [Function]

Returns the next XIP that has arrived addressed to socket *NSOC*. If there are no XIPs waiting on *NSOC*, then **GETXIP** returns **NIL**, or waits for a XIP to arrive if *WAIT* is **T**. If *WAIT* is an integer, **GETXIP** interprets it as a number of milliseconds to wait, finally returning **NIL** if a XIP does not arrive within that time.

(DISCARDXIPS *NSOC***)** [Function]

Discards without examination any XIPs that have arrived on *NSOC* and not yet been read by a **GETXIP**.

(EXCHANGEXIPS *SOC OUTXIP IDFILTER TIMEOUT***)** [Function]

Useful for simple NS packet exchange protocls. Sends *OUTXIP* on *SOC*, then waits for a responding XIP, which it returns. If *IDFILTER* is true, ignores XIPs whose packet exchange ID (the first 32 bits of the data portion) is different from that of *OUTXIP*. *TIMEOUT* is the length of time (msecs) to wait for a response before giving up and returning **NIL**. *TIMEOUT* defaults to **\ETHERTIMEOUT**. **EXCHANGEXIPS** discards without examination any XIPs that are currently waiting on *SOC* before *OUTXIP* gets sent.

31.6.4 NS Debugging Aids

XIPs can be printed automatically by **SENDXIP** and **GETXIP** analogously to the way pups are. The following variables behave with respect to XIPs the same way that the corresponding PUP-named variables behave with respect to PUPs: **XIPTRACEFLG, XIPTRACEFILE, XIPIGNORETYPES, XIPONLYTYPES, XIPPRINTMACROS.** In addition, the functions **PRINTXIP, PRINTXIPROUTE, XIPTRACE,** and **NS.ECHOUSER** are directly analogous to **PRINTPUP, PRINTPUPROUTE, PUPTRACE,** and **PUP.ECHOUSER.** See page 31.32.

31.7 Support for Other Level One Protocols

Raw packets other than of type PUP or NS can also be sent and received. This section describes facilities to support such protocols. Many of these functions have a \ in their names to designate that they are system internal, not to be dealt with as casually as user-level functions.

(RESTART.ETHER) [Function]

This function is intended to be invoked from the executive on those rare occasions when the Ethernet appears completely unresponsive, due to Lisp having gotten into a bad state. **RESTART.ETHER** reinitializes Lisp's Ethernet driver(s), just as when the Lisp system is started up following a **LOGOUT, SYSOUT,** etc. This aborts any Ethernet activity and clears several internal caches, including the routing table.

(\ALLOCATE.ETHERPACKET) [Function]

Returns an **ETHERPACKET** datum. Enough of the packet is cleared so that if the packet represents a **PUP** or **NS** packet, that its header is all zeros; no guarantee is made about the remainder of the packet.

(\RELEASE.ETHERPACKET *EPKT***)** [Function]

Returns *EPKT* to the pool of free packets. This operation is dangerous if the caller actually is still holding on to *EPKT*, e.g., in some queue, since this packet could be returned to someone else (via **\ALLOCATE.ETHERPACKET**) and suffer the resulting contention.

From a logical standpoint, programs need never call **\RELEASE.ETHERPACKET**, since the packets are eventually garbage-collected after all pointers to them drop. However, since the packets are so large, normal garbage collections tend not to occur frequently enough. Thus, for best performance, a well-disciplined program should explicitly release packets when it knows it is finished with them.

A locally-connected network for the transmission and receipt of Ether packets is specified by a *network descriptor block*, an object of type **NDB**. There is one **NDB** for each directly-connected network; ordinarily there is only one. The **NDB** contains information specific to the network, e.g., its **PUP** and **NS** network numbers, and information about how to send and receive packets on it.

\LOCALNDBS [Variable]

The first **NDB** connected to this machine, or **NIL** if there is no network. Any other **NDB**s are linked to this first one via the **NDBNEXT** field of the **NDB**.

In order to transmit an Ether packet, a program must specify the packet's type and its immediate destination. The type is a 16-bit integer identifying the packet's protocol. There are preassigned types for **PUP** and **NS**. The destination is a host address on the local network, in whatever form the local network uses for addressing; it is not necessarily related to the logical ultimate destination of the packet. Determining the immediate destination of a packet is the task of *routing*. The functions **SENDPUP** and **SENDXIP** take care of this for the **PUP** and **NS** protocols, routing a packet directly to its destination if that host is on the local network, or routing it to a gateway if the host is on some other network accessible via the gateway. Of course, a gateway must know about the type (protocol) of a packet in order to be able to forward it.

(ENCAPSULATE.ETHERPACKET *NDB PACKET PDH NBYTES ETYPE***)** [Function]

Encapsulates *PACKET* for transmission on network *NDB*. *PDH* is the physical destination host (e.g., an 8-bit pup host number or a 48-bit NS host number); *NBYTES* is the length of the packet in bytes; *ETYPE* is the packet's encapsulation type (an integer).

(TRANSMIT.ETHERPACKET *NDB PACKET***)** [Function]

Transmits *PACKET*, which must already have been encapsulated, on network *NDB*. Disposition of the packet after transmission is complete is determined by the value of *PACKET*'s **EPREQUEUE** field.

In order to receive Ether packets of type other than **PUP** or **NS**, the programmer must specify what to do with incoming packets. Lisp maintains a set of *packet filters*, functions whose job it is to appropriately dispose of incoming packets of the kind they want. When a packet arrives, the Ethernet driver calls each filter function in turn until it finds one that accepts the packet. The filter function is called with two arguments: (*PACKET TYPE*), where *PACKET* is the actual packet, and *TYPE* is its Ethernet encapsulation type (a number). If a filter function accepts the packet, it should do what it wants to with it, and return **T**; else it should return **NIL**, allowing other packet filters to see the packet.

Since the filter function is run at interrupt level, it should keep its computation to a minimum. For example, if there is a lot to be done with the packet, the filter function can place it on a queue and notify another process of its arrival.

The system already supplies packet filters for packets of type **PUP** and **NS**; these filters enqueue the incoming packet on the input queue of the socket to which the packet is addressed, after checking that the packet is well-formed and indeed addressed to an existing socket on this machine.

Incoming packets have their **EPNETWORK** field filled in with the **NDB** of the network on which the packet arrived.

(\ADD.PACKET.FILTER *FILTER***)** [Function]

Adds function *FILTER* to the list of packet filters if it is not already there.

(\DEL.PACKET.FILTER *FILTER***)** [Function]

Removes *FILTER* from the list of packet filters.

(\CHECKSUM *BASE NWORDS INITSUM***)** [Function]

Computes the one's complement add and cycle checksum for the *NWORDS* words starting at address *BASE*. If *INITSUM* is supplied, it is treated as the accumulated checksum for some set of words

preceding *BASE*; normally *INITSUM* is omitted (and thus treated as zero).

(PRINTPACKET *PACKET CALLER FILE PRE.NOTE DOFILTER***)** [Function]

Prints *PACKET* by invoking a function appropriate to *PACKET*'s type. See **PRINTPUP** for the intended meaning of the other arguments. In order for **PRINTPACKET** to work on a non-standard packet, there must be information on the list **\PACKET.PRINTERS**.

\PACKET.PRINTERS [Variable]

An association list mapping packet type into the name of a function for printing that type of packet.

31.8 The SYSQUEUE mechanism

The **SYSQUEUE** facility provides a low-level queueing facility. The functions described herein are all system internal: they can cause much confusion if misused.

A **SYSQUEUE** is a datum containing a pointer to the first element of the queue and a pointer to the last; each item in the queue points to the next via a pointer field located at offset 0 in the item (its **QLINK** field in the **QABLEITEM** record). A **SYSQUEUE** can be created by calling **(NCREATE 'SYSQUEUE)**.

(\ENQUEUE *Q ITEM***)** [Function]

Enqueues *ITEM* on *Q*, i.e., links it to the tail of the queue, updating *Q*'s tail pointer appropriately.

(\DEQUEUE *Q***)** [Function]

Removes the first item from *Q* and returns it, or returns **NIL** if *Q* is empty.

(\UNQUEUE *Q ITEM NOERRORFLG***)** [Function]

Removes the *ITEM* from *Q*, wherever it is located in the queue, and returns it. If *ITEM* is not in *Q*, causes an error, unless *NOERRORFLG* is true, in which case it returns **NIL**.

(\QUEUELENGTH *Q***)** [Function]

Returns the number of elements in *Q*.

(\ONQUEUE *ITEM Q***)** [Function]

True if *ITEM* is an element of *Q*.