1 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.

1.1 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 datatype 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 datatype 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.

1.2 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 INIT file 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 *OBJECT*) [Function]
Returns a list of all objects in the Clearinghouse database that are aliases of *OBJECT*.

(CH.RETRIEVE.ITEM *OBJECT PROPERTY INTERPRETATION*) [Function]
Retrieves the value of the *PROPERTY* property of *OBJECT*. 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 section X.XX) 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 *OBJECT PROPERTY* ---) [Function]
Retrieves the members of the group *OBJECT*, as a list of NSNAME's. *PROPERTY* is 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.

1.3 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. Possible properties are as follows:

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

DOCUMENT.CREATION.DATE The creation date to appear on the header page (a Lisp integer date, such as returned by IDATE). The default value is the creation date of the file.

SENDER.NAME The name of the sender to appear on the header page (a string). The default value is the name of the user.

RECIPIENT.NAME The name of the recipient to appear on the header page (a string). The default is none.

#COPIES The number of copies to be printed. The default value is 1.

MEDIUM The medium on which the master is to be printed. If omitted, this defaults to the value of NSPRINT.DEFAULT.MEDIUM , as follows: NIL means to use the printer's default; T means to use the first medium reported available by the printer; any other value must be a Courier value of type MEDIUM. The format of this type is a list (PAPER (KNOWN.SIZE *TYPE*)) or (PAPER (OTHER.SIZE (*WIDTH LENGTH*))). The paper *TYPE* is one of US.LETTER, US.LEGAL, A0 through A10, ISO.B0 through ISO.B10, and JIS.B0 through JIS.B10.

STAPLE? True if the document should be stapled.

#SIDES 1 or 2 to indicate that the document should be printed on one or two sides, respectively. The default is the value of EMPRESS#SIDES.

PRIORITY The priority of this print request, one of LOW, NORMAL, or HIGH. The default is the printer's default.

(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---type of paper loaded, whether it can print two-sided, etc.

1.4 NS Filing

Lisp accesses Xerox NS fileservers using the NS Filing Protocol. For most operations, the programmer simply treats the NS fileserver as any other host or device. OPENFILE, GETFILEINFO, DIRECTORY all work appropriately when the filename specifies an NS file server host name, e.g.,

{PHYLEX:PARC:XEROX}<LISP>LIBRARY>GRAPHER.DCOM;1 . Note that all NS File Server host names must contain a colon, even if the domain and organization fields are defaulted, in order that they be distinguishable from other types of host names (e.g., Pup server names). Note also that spaces are allowable characters in NS names. Thus, if an NS file name contains spaces (in the file name or the server name) and is presented as a litatom, the spaces must be quoted with %. However, all the standard file operations also work when the name is presented as a string, in which case there is no problem with spaces.

The following are features specific to NS fileservers.

(BREAK.NSFILING.CONNECTION *HOST*) [Function]
Closes any open connections to NS file server *HOST* .

FILING.ENUMERATION.DEPTH [Variable]
The full NS Filing Protocol supports a true hierarchical name space for files. This leaves some ambiguity about how deep the function DIRECTORY should enumerate files. The depth is controlled by the variable FILING.ENUMERATION.DEPTH , which is either a number, specifying the number of levels deep to enumerate, or T, meaning enumerate to all levels. In the former case, when the enumeration reaches the specified depth, only the subdirectory name rooted at that level is listed, and none of its descendants is listed. When FILING.ENUMERATION.DEPTH is T, all files are listed, and no subdirectory names are listed. FILING.ENUMERATION.DEPTH is initially T.

Independent of FILING.ENUMERATION.DEPTH , a request to enumerate the top-level of a file server's hierarchy lists only the top level, i.e., assumes a depth of 1. For example, (DIRECTORY '{PHYLEX:}) lists exactly the top-level directories of the server PHYLEX:.

1.5 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 WHENCLOSEDFN*) [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. *WHENCLOSEDFN* is an optional function or list of functions to call when the stream is closed, either by the user or the server.

(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.SENDEOM` *STREAM*) [Function]
Transmits the data buffered so far on this output stream, if any, with the End of Message bit set. If there is nothing buffered, sends a zero-length packet with End of Message bit set.

(`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.EOMP` *STREAM*) [Function]
This function returns `T` or `NIL` depending on whether or not an End of Message indication has been reached. This is only true after the last byte of data in the message has been read. By convention, `EOFP` is true of a stream that is at EOM.

(`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 true.

(`SPP.SENDATTENTION` *STREAM ATTENTIONBYTE*) [Function]
Sends an SPP "attention" packet, one with the Attention bit set and containing the single byte of data *ATTENTIONBYTE* .

1.6 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.

1.6.1 Defining Courier Programs

A Courier program definition is a file package type and command, `COURIERPROGRAMS`. Thus, you can use `GETDEF`, `PUTDEF`, and `EDITDEF` to manipulate them, or use the file package command `(COURIERPROGRAMS name1 name2 ...)` to save them. The function `COURIERPROGRAM` can be used to define a Courier program initially.

(`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.

1.6.2 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.

1.6.2.1 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` (section X.XX), and is encoded as six items of type `UNSPECIFIED`.

`NSNAME` Represents a three-part Clearinghouse name. The value is an object of type `NSNAME` (section X.XX), 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` (section X.XX), and is encoded as two items of type `STRING`.

1.6.2.2 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".

1.6.2.3 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 some of the functions described in section X.XX. *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 (section X.XX.

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.

1.6.3 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*) [Function]

Opens a Courier connection to the Courier socket on *HOST*, and returns an SPP stream that can be passed to COURIER.CALL. *HOST* 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. If *NOERRORFLG* is true, NIL is returned if a connection cannot be made, or the server supports the wrong version of Courier. The Courier connection process is named *NAME*, if specified. *WHENCLOSEDFN* is a function of one argument, the Courier stream, that will be called when the connection is closed, either by user or server.

(COURIER.CALL *STREAM PROGRAM PROCEDURE* $ARG_1 \ldots ARG_N$ *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 section X.XX.

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 . BIRTHDAY)
                              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).

1.6.3.1 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* $ARG_1 \ldots ARG_N$ *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.

1.6.3.2 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.

1.6.3.3 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* `)` [Function]
Reads from *STREAM* a "Stream of *TYPE* " for Courier program *PROGRAM*, and returns a list of the objects read.

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))`.

1.6.3.4 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 BASETYPE* `)` [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 *BASETYPE*)).

(COURIER.WRITE.SEQUENCE *STREAM ITEM PROGRAM BASETYPE*) [Function]
Equivalent to (COURIER.WRITE *STREAM ITEM PROGRAM* (SEQUENCE *BASETYPE*)).

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)).