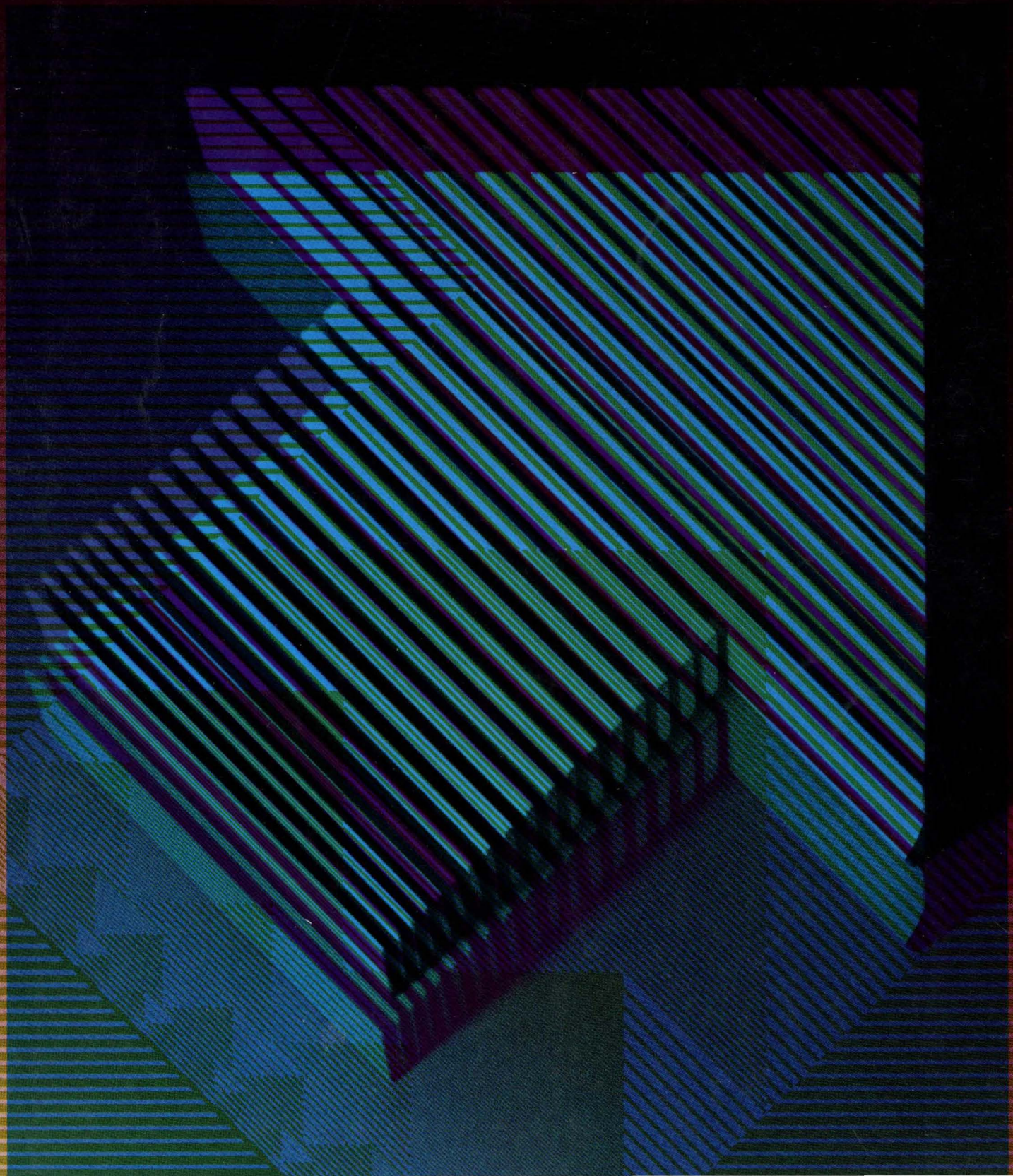
# 9 Networks

symbolics



# 9 Networks

symbolics

Cambridge, Massachusetts

# Networks

**# 996095**

**March 1985**

**This document corresponds to Release 6.0 and later releases.**



Text written and produced on Symbolics 3600-family computers by the Documentation Group of Symbolics, Inc.

Text typography: Century Schoolbook and Helvetica produced on Symbolics 3600-family computers from Bitstream, Inc., outlines; text masters printed on Symbolics LGP-1 Laser Graphics Printers.
Cover design: Schafer/LaCasse
Cover printer: W.E. Andrews Co., Inc.
Text printer: ZBR Publications, Inc.

Printed in the USA.

Printing year and number: 87 86 85 9 8 7 6 5 4 3 2 1

# Table of Contents

# List of Tables

# PART I.

# General Information on Networks

# 1. Namespace System

## 1.1 Introduction to the Namespace System

When computers are connected by means of networks to form a distributed computing environment, the computers should all be able to share information that describes that environment. For example, all the computers need to know or be able to find out about the names and addresses of the other computers to which they can communicate. A personal workstation computer might want to know what printers are available on which server computers. A computer trying to send mail to a particular user might want to know on which computer that user's mailbox resides on.

The Symbolics system has a convention by which such information can be maintained and shared in a simple database. The database is maintained by a *namespace server*. Other systems can query or make changes to the database by communicating over the network with the server. This database is the *namespace database*, a specific example of a distributed network database. Both the more specific term *namespace database* and the generic term *network database* are used to refer to it. However, in general, *namespace database* refers to the Symbolics implementation of network databases and *namespace system* refers to the namespace database and the tools to use it.

The database is structured to understand that there can be many different networks in a distributed environment, and so there are network objects to represent different networks. Hosts can be on more than one network, and some hosts that are on two networks can serve as gateways from one network to the other. One of the purposes of the database is to let a user host find a path to a server host, using whichever networks and gateways are necessary.

The database is designed so that it is not specific to the Symbolics computer; in theory, any computer system could be made to use the database, and act as a user or server.

The database consists of a collection of *objects*. Each object has

- A class. See the section "Namespace System Classes", page 4.
- Attributes. See the section "Namespace System Attributes", page 4.
- A name. See the section "Names and Namespaces", page 5.

All objects except namespaces themselves are added to the namespace database by using the namespace editor, which is invoked with Edit Namespace Object (or from Lisp with **tv:edit-namespace-object**). See the section "Updating the Namespace Database" in *User's Guide to Symbolics Computers*.

### 1.1.1 Namespace System Classes

Every object has a *class*, which tells what kind of object it is. Every class is identified by a global-name.

The following classes are especially important to the Symbolics system:

**host** A host object represents any computer, usually connected to a network.

**user** A user object represents a person who uses any of the hosts, or a daemon user, for example, a Symbolics computer.

**network** A network object represents a computer network, to which some hosts are attached.

**printer** A printer object represents a device for producing hardcopy.

**site** A site object represents a collection of hosts, printers, and networks that are grouped together in one physical location.

**namespace** A namespace object represents a mapping from names of objects to objects.

### 1.1.2 Namespace System Attributes

Attributes represent characteristics of the object. Each attribute has an *indicator* (the name of the attribute) and a *value;* they work like property lists in Lisp. For example, every host has a system-type (saying which operating system it runs), every printer has a type (saying what type of printer it is), and every user has a personal-name.

Each object class has one or more required attributes. However, most attributes are optional; for example, hosts can optionally have a pretty-name, printers can have a default-font, and a user can have a home-address. Some attributes can occur more than once for a given object; for example, a host object can have multiple addresses if it is attached to more than one network.

Each object has a fixed set of attributes: you cannot create additional attributes.

### 1.1.3 Data Types of Namespace System Attributes

Each class has attributes that are defined to have specific data types. Since the actual representation of the various types of data represented in the database varies from system to system, the namespace system uses the following system-independent types:

| *Data type* | *Value* |
|---|---|
| *object-class* | An object in the database, for example, a site object. See the section "Namespace System Classes", page 4. |
| name | A name in some namespace; name is not shared by all namespaces. |
| global-name | A name which is not specific to a particular namespace but is shared by all namespaces. |
| token | An arbitrary character string. |
| set | An ordered set of elements of the same data type. For example, a value can be a set of names or a set of triples. |
| pair | A list of two elements of specific data types; each element can be of a different data type. |
| triple | A list of three elements; each element can be of a different data type. |

Name, global-name, and token require simple values, whereas set, pair, and triples require compound values.

*Note*: The namespace data types specific to the Symbolics computer are described elsewhere: See the section "Namespace System Lisp Data Types", page 29.

### 1.1.4 Names and Namespaces

Every object has a *name*, which is a character string. Two objects of different classes can have the same name; for example, there can be a printer named george and a user named george; the two are unrelated. An object is identified by its class and its name; if you want to look up an object in the database and you know its name, you have to say "Find the printer named george" or "Find the user named george", not just "Find george".

When long-distance networks are used to link together different sites, however, the possibility of name conflicts arises; that is, two sites may use the same name in the same class for conflicting purposes. For example, suppose you had a host named orange, and you wanted to connect your site over a long-distance network to some other site that happens to have picked the name orange for one of its own hosts. Neither site is forced to change its host names just because it wants to connect to the other site.

To avoid these naming conflicts, the database can include more than one namespace. A *namespace* is a mapping from names to objects, and names in one namespace are unrelated to names in another namespace. (More strictly, it is a mapping from [name, class] pairs to objects, since an object is identified by its class and its name.) Normally each site has one namespace, and the names of all the objects at that site are in that namespace. An object in some other namespace than your own namespace can be referred to using a *qualified name*, which consists of the name of the namespace, a vertical bar, and the name of the object in that namespace.

For example, suppose both Harvard and Yale had computer centers. Harvard has three hosts named yellow, orange, and blue, and Yale has three hosts named apple, orange, and banana. Each computer center would have its own namespace, one named harvard and one named yale. At Harvard, the Harvard computers would be referred to by their unqualified names (yellow, orange, and blue), whereas the Yale computers would be referred to (by users at Harvard) by qualified names (yale|apple, yale|orange, and yale|banana). At Yale it would all work the other way around.

Each namespace also has a list of namespaces called *search rules*. When a name is looked up, each of the namespaces in the search rules list is consulted in turn, until an object of that name is found in one of the namespaces. If you have some other namespace in your search list, it is easier to refer to objects in that namespace, because you do not have to use qualified names unless a name conflict exists.

For example, in the scenario above, the search list for the harvard namespace could have the harvard namespace first and the yale namespace second. Then users at Harvard could refer to Yale's computers as apple, yale|orange, and banana. The qualified name is only necessary because of the name conflict.

Actually, only some classes of objects have names that are in namespaces; other classes of objects are *globally* named, which means that the names are universal, and conflicts are not permitted. In particular, classes, namespaces, and sites are globally named; networks, hosts, printers, and users are named within namespaces. There is never a need for multiply-qualified names; the names of namespaces are global and never need to be qualified themselves.

Some namespaces do not correspond to any local site. Most large nationwide or worldwide networks have their own host naming convention. For example, the Department of Defense Arpanet has its own set of host names, and this is considered a namespace. If a local site includes some hosts that are on the Arpanet, it might want to put the Arpanet namespace into its search list, and install gateways on its Arpanet machine so that other machines on the local network can access the Arpanet.

Some objects can also have *nicknames*. In particular, networks and hosts can have nicknames; objects of other classes cannot. A nickname serves as an alternative name by which the object can be referred. Sometimes you give an object a nickname because its full name is too long to type conveniently, like some host whose name you type frequently. However, each object always has one primary name, which is used when the object is printed.

It is possible for an object to be in several namespaces at once. For example, a host which is on both the Arpanet and a local network at some site might be in both the Arpanet namespace and the local namespace. In this case, each namespace maintains its own separate information on the object. The information from each namespace is merged before being presented to the user.

Note: Search lists are not followed recursively. If a user at Harvard looks up a

name and Yale's namespace is in Harvard's search list, Yale's search list is *not* relevant.

## 1.2 Namespace System Object Definitions

This chapter provides a description of the attributes and values for the following classes:

host
user
network
printer
site
namespace

### 1.2.1 Namespace System Host Objects

A host object represents any computer connected to a network. The database lists what networks a host is connected to, and at what addresses. It also says what high-level services the host provides to the network community.

The following is a list of all attributes that hosts can have together with examples of what those attributes can look like. The **name** and **system-type** attributes are required; all others are optional. Some of the attributes are useful only for Symbolics computer hosts.

**name: Host Object Attribute**

Specifies the primary name of the host; a name (required).

```
yellow
```

**site: Host Object Attribute**

Specifies the site at which this host is located; a site object.

```
harvard
```

**nickname: Host Object Attribute**

Specifies alternate names for the host; a set of names.

```
yel
```

**short-name: Host Object Attribute**

Specifies additional nicknames; a set of names. A short-name is used when a program wants to display a host's name without using up too much space. It is also used in the printed representation of pathnames.

```
¥
```

**user-property: Object Attribute**

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

**machine-type: Host Object Attribute**

Specifies the kind of machine out of which this host is made; a global-name. Common values are:

```
lispm
pdp10
pdp11
vax
honeywell-dps-8m
alto
ibmpc
pe3230
3600
cadr
```

**system-type: Host Object Attribute**

Specifies the operating system run on the host; a global-name (required). The Symbolics system uses this information to figure out how to parse pathnames for a given host; be sure to enter this information correctly. Common values are:

```
lispm
unix
vms
tops-20
alto
its
multics
minits
magicsix
mos
ms-dos
```

**address: Host Object Attribute**

Specifies the network addresses of this host; a set of pairs or triples. Each triple is of the form (*network address interface*), where *network* is a network object, *address* is a token, and

*interface* (optionally) identifies the interface for which this address is valid. Addresses are always represented as tokens because each kind of network has a different kind of address; the individual network types and their corresponding address conventions are discussed later in this document. An example of a pair:

```
(chaos "401")
```

**pretty-name: Host Object Attribute**

Specifies a "pretty" version of the name of the host; a token. Unlike the real name, the nicknames, and the short name, this does not count as a name as far as the database system is concerned (you cannot use it to find the host).

```
"Yellow Computer"
```

**finger-location: Host Object Attribute**

Specifies a description of the physical location of the host; a token. This is used by the **lispm-finger** and **show-users** services.

```
"The Out-of-Town News kiosk in Harvard Square"
```

**location: Host Object Attribute**

Specifies a description of the physical location suitable for programs to understand; a pair. The first element is a token that identifies the building the machine is in. The second element is a token that says what floor of the building the machine is on.

```
Kiosk 1
```

**printer: Host Object Attribute**

Specifies the preferred printer for this host; a printer object. This printer is used by default when files are hardcopied from this host. If this attribute is not provided, the site's **default-printer** attribute is used.

**bitmap-printer: Host Object Attribute**

Specifies the preferred bitmap printer for this host; a printer object. This printer is used by default when screen images are hardcopied from this host. If this attribute is not provided, site's **default-bitmap-printer** attribute is used.

### print-spooler-options: Host Object Attribute

Specifies options for any print spoolers running on this host; a set of pairs of global-names and tokens. A keyword for print-spooler-options is:

| | |
|---|---|
| home-directory | Directory where hardcopy requests are stored. The default for Symbolics computers is local:>print-spooler>. |

### spooled-printer: Host Object Attribute

Specifies printers for which this host provides a spooling service; a printer object followed by a set of pairs of global-names and tokens describing the spooling service for that printer. Allowed global-names are:

| | |
|---|---|
| spool-via | Method of spooling, for example, "network" (the default) or "file". |
| file-name | Name of spooling file of **spool-via** file. |
| protocol | Special spooling protocol, when **spool-via** is "network". If not present, the generic hardcopy service to the host is used. |
| home-directory | Directory where hardcopy requests are stored. The default for Symbolics computers is local:>print-spooler>. |

For Symbolics spoolers no keywords are normally necessary.

Example:

```
caspian-sea spool-via file file-name yellowunix:print;request
```

### service: Host Object Attribute

Specifies services and protocols supported by this host; a list of triples of the form *service medium protocol*. Each triple specifies that the host is capable of providing *service* when you connect to it using *medium* and *protocol*. Services, media, and protocols are discussed elsewhere: See the section "The Lisp Machine Generic Network System", page 37.

```
((file chaos qfile)
 (lispm-finger chaos-simple lispm-finger)
 (namespace-timestamp chaos-simple namespace-timestamp)
 (namespace chaos namespace)
 (mail-to-user chaos chaos-mail)
 (send chaos send)
 (login chaos telnet)
 (uptime chaos-simple uptime-simple)
 (time chaos-simple time-simple)
 (show-users chaos name)
 (chaos-status chaos-simple chaos-status))
```

**server-machine: Host Object Attribute**

Specifies whether the object described is a server machine; a token. If the value is the string **"yes"**, then this host is a server machine; if the attribute is not present, the host is not a server machine. Values other than "yes" are undefined and should not be used. This attribute only applies to Symbolics computers. Server machines do not automatically enable their services when they are booted. This is to prevent premature creation of servers before the machine has completely initialized. See the function **sys:enable-services**.

**file-control-lifetime: Host Object Attribute**

Specifies the lifetime of a file control connection; a token. The value is a string representing a number in decimal. When a Symbolics computer connects to this host as a user of the **file** service, it will automatically close its control connection if that connection has been idle for this number of sixtieths of a second.

```
"108000"
```

**peripheral: Host Object Attribute**

Specifies a peripheral device; a set of pairs. The first element is a peripheral type and is a global-name. The second element describes the device and is a set of pairs of global-names and tokens.

```
kanji-tablet unit 2 baud 2400
```

### 1.2.2 Namespace System User Objects

A user object represents either a person who uses any of the hosts, or a daemon pseudouser. Each person who uses Symbolics computers should be *registered* in the database; this means that there is a corresponding user object. Daemon users are

what Symbolics computers log in as when they need to conduct operations even though there is no particular person identifiable as the user. This typically happens when the Symbolics computer is acting as a file server or a mail server, or when it is performing maintenance functions such as saving a band with new patches loaded.

The following is a list of all attributes that users can have. The **name**, **lispm-name**, **personal-name**, **home-host**, and **mail-address** attributes are required; all others are optional.

### name: User Object Attribute

Specifies the name of the user. It can be used as an argument to the **login** command on a Symbolics computer.

```
george
```

### login-name: User Object Attribute

Specifies the appropriate login name for each of several hosts, a set of pairs. The first element of each pair is a token giving the login name, and the second element is a host object.

The Symbolics computer uses these login names when it connects to a host to log in a file server or a tape server. **login-name** is not required, but lack of this attribute causes the Lisp Machine to ask for the name to use for each server, which is likely to be inconvenient. Passwords are not stored in the database because it is not secure; the Symbolics computer prompts the user for a password interactively when one is required. Generally, you have one element on this list for every account that you have on a host on the network.

```
(("George" blue)
 ("Washington.States" mit|multics)
 ("GW" mit|mc))
```

### lispm-name: User Object Attribute

Specifies the name displayed in the status line; a token. Used in the **lispm-finger** service as the user name. The Lisp variable **user-id** is set from this attribute. Typically it is similar to the actual name of the user object, but uses upper- and lower-case.

```
"George"
```

### personal-name: User Object Attribute

Specifies the user's personal name; a token. Place the last name first.

```
"Washington, George"
```

### home-host: User Object Attribute

Specifies the user's host machine; a token.

```
"Vixen"
```

### nickname: User Object Attribute

Specifies a personal nickname; a token. Unlike host nicknames, user nicknames cannot be used to look up the user.

```
"Georgie"
```

### user-property: Object Attribute

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

### work-address: User Object Attribute

Specifies a work (business) address; a token.

```
"The White House, Washington D.C., 10001"
```

### work-phone: User Object Attribute

Specifies the work (business) phone number; a token.

```
"202-555-1212"
```

### home-address: User Object Attribute

Specifies the home address; a token.

```
"Mount Vernon VA"
```

### home-phone: User Object Attribute

Specifies the home phone number; a token.

```
"202-999-1234"
```

### mail-address: User Object Attribute

Specifies the network mailbox at which the user wants to receive mail. The value is a pair; the first element is the mailbox name (a token), and the second is a host object. Defaults to name@home-host

```
("George" blue)
```

**birthday: User Object Attribute**

Specifies the user's birthday; a token.

```
"Feb 22"
```

**project: User Object Attribute**

Specifies a description of what the user is working on; a token.

```
"being President of the United States"
```

**supervisor: User Object Attribute**

Specifies who the user is working for; a token.

```
"the People"
```

**affiliation: User Object Attribute**

Specifies the user's group affiliation; a single-character token. The letter is arbitrary and can refer to different sets of users at different sites.

```
"p"
```

**remarks: User Object Attribute**

Specifies other relevant information; a token.

```
"I cannot tell a lie."
```

### 1.2.3 Namespace System Network Objects

A network object represents a computer network, to which some hosts are attached. The **name** and **type** attributes are required; all others are optional.

**name: Network Object Attribute**

Specifies the name of the network; a name object (required).

```
harvnet
```

**nickname: Network Object Attribute**

Specifies alternate names for the network; a set of names. The network may be found by these names.

**site: Network Object Attribute**

Specifies the site at which this network is located; a site object.

```
harvard
```

### type: Network Object Attribute

Specifies the type of network; a global-name. Common network types include:

**chaos** A network using the MIT Chaos protocols. Addresses are 16-bit numbers represented in octal.

```
17006
```

**internet** A network using the DOD Internet protocols. Addresses are the 32-bit Internet addresses as four octets represented in decimal separated by periods.

```
10.0.0.6
```

**dial** A direct-dial telephone network. Usually there is only one of these, called **dial** by convention. Addresses are telephone numbers relative to the dialing conventions of the installation.

```
15551212
```

**x25** A packet-switching network with a CCITT Recommendation X.25 interface. Addresses are X.121 addresses.

```
311061700138
```

**gateway-pseudonet** A network actually implemented by direct connection of a gateway to a terminal line. Address is service-name = contact name on gateway host, for example, `tty-login=prime`.

### user-property: Object Attribute

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

### subnet: Network Object Attribute

Specifies characteristics of a subnetwork for this network; a pair. The first element is a token naming the subnet in that network's conventions. The second element is a set of pairs of global-names and tokens that provide extra information on that subnet.

Example:

```
(("81" ((cable-start "Rick's office" cable-end "Jane's office"))))
```

### 1.2.4 Namespace System Printer Objects

A printer object represents a hardcopy output device. The **name**, **type**, and **host** attributes are required; the rest are optional.

#### name: Printer Object Attribute

Specifies the name of the printer (required); a name.

```
caspian-sea
```

#### type: Printer Object Attribute

Specifies the device type of the printer; a global-name (required). This attribute implies some data formats that are interpreted by the device. Common values are:

**lgp**
**ascii**
**press**
**xgp**

#### site: Printer Object Attribute

The site where the printer is located; a site object. Generally all printers at a site are offered in menus of potential output devices for the destination of a hardcopy request.

```
HARVARD
```

#### pretty-name: Printer Object Attribute

Specifies a name for the printer; a token.

```
"Caspian Sea"
```

#### user-property: Object Attribute

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

**format: Printer Object Attribute**

Specifies the print formats supported by the device; a set of global-names. These are in addition to those implied by the **type** attribute. Common print formats are:

**lgp**
**press**
**xgp**
**ascii**
**tektronics**

**interface: Printer Object Attribute**

Specifies the type of interface by which this printer is attached to its host; a global-name. Possible values are:

**serial**
**drll-c**

**interface-options: Printer Object Attribute**

Specifies parameters of the hardware interface; a set of pairs of global-names and tokens. For each interface type attribute, give the permissible option names and their default values.

**host: Printer Object Attribute**

Specifies the host to which the printer is directly connected; a host object (required).

**protocol: Printer Object Attribute**

Specifies the protocols to use for direct unspooled printing; a set of global-names. If not specified, the HARDCOPY service is invoked on the host to which the printer is directly connected.

**default-font: Printer Object Attribute**

Specifies the name of the font that should normally be used for this printer; a token. If not specified, the default-font is usually determined by the type of printer.

**header-font: Printer Object Attribute**

Specifies the name of the header font that should normally be used; a token. If not specified, the header-font is usually determined by the type of printer.

**dplt-logo: Printer Object Attribute**

Specifies the name of the logo printed by DPLT; a global-name.

```
Symbolics
```

**character-size: Printer Object Attribute**

Specifies the size of a character in micas; a pair of width and height, in decimal. (A *mica* is 10 microns, or 1/2540 of an inch.)

**page-size: Printer Object Attribute**

Specifies the size of the page in device units; a pair of width and height, in decimal.

```
"135" "80"
```

**font-widths-file: Printer Object Attribute**

Specifies the name of the fonts.widths file for this printer; a token. It is best if this is a fully qualified physical pathname instead of a logical pathname, for example: SCRC|ARGUS:>sys>stats>lgp-1>fonts.widths.

### 1.2.5 Namespace System Site Objects

A site object represents a collection of hosts, printers, and networks that are grouped together in one physical location, within one timezone. The **name**, **local-namespace**, **site-directory**, **host-for-bug-reports**, and **timezone** attributes are required; the rest are optional.

**pretty-name: Site Object Attribute**

Specifies a version of the name suitable for people to read; a token.

```
"Harvard University"
```

**user-property: Object Attribute**

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

### local-namespace: Site Object Attribute

Specifies the site's local namespace; a namespace object (required). This is the namespace that will be used at that site. Normally, there is exactly one namespace for each site.

```
harvard
```

### site-directory: Site Object Attribute

Specifies the file name of the directory that holds the Symbolics computer system's site-specific files at this site; a token (required). This is used only to find the files that define the logical hosts, such as sys:. All other site-specific pathnames are managed by logical pathname translations or by the descriptor file attribute of a namespace.

```
blue:>sys>site>
```

### site-system: Site Object Attribute

Specifies the name of a system (in the **defsystem** sense) to be loaded automatically into Symbolics computer worlds at this site; a token.

```
"HARV-SPECIFIC"
```

### default-printer: Site Object Attribute

Specifies the default printer to use for printing text files at this site; a printer object. This will be used by hosts that do not have their own **printer** attribute.

### default-bitmap-printer: Site Object Attribute

Specifies the default printer to use for printing screen images at this site; a printer object. This attribute is for hosts that do not have their own **bitmap-printer** attribute.

### host-for-bug-reports: Site Object Attribute

Specifies the host to which bug reports should be sent, by the Debugger c-M command, the Bug (m-X) command, and the Zmail bug commands (required); a host object.

```
blue
```

### host-protocol-desirability: Site Object Attribute

Specifies a tuning factor to be used in the Generic Network System's cost estimates when trying to construct a path to a

service; a triple of the form (host protocol desirability), in which host represents a host, protocol names some protocol that host supports, and desirability is a token expressing a floating-point factor for the cost calculations. Services and protocols are discussed elsewhere: See the section "The Lisp Machine Generic Network System", page 37. Desirability is discussed elsewhere: See the section "Interfacing to the Network System", page 201.

```
(yukon chaos-mail "0.75")
```

**timezone: Site Object Attribute**

The timezone at this site; a global-name (required).

```
est
```

**secure-subnets: Site Object Attribute**

Specifies an association of networks and secure subnet numbers; a set of pairs. The first element of each pair is a network; it must be of type chaos. The second element of each pair is a set of subnet numbers, the interpretation of which depends on the type of the network. For a chaos network, the set is represented as octal character strings.

This attribute controls the subnet security feature of the Symbolics file server as well as other servers which use the **:trusted-p** or **:reject-unless-trusted** keywords to **net:define-server**. Hosts on these subnets are considered trustworthy.

**dont-reply-to-mailing-lists: Site Object Attribute**

Specifies a set of names of mailing lists to which Zmail does not reply by default; tokens. This attribute is useful only to those who have not set the PEOPLE NOT TO REPLY TO option in their Zmail init files.

**other-sites-ignored-in-zmail-summary: Site Object Attribute**

Specifies a set of site objects. Zmail does not display the host names of hosts from the specified sites in its summary window as well as not doing so for this site.

**standalone: Site Object Attribute**

Specifies whether the host at this site is a standalone machine; a token. If the value is the string "yes", then only one host exists at this site and no response to the *who-am-I* network broadcast request at boot time is expected. If the attribute is not present

or the value is not "yes", then multiple Symbolics computer hosts exist at this site; when one host is booted, another host answers its who-am-I query.

**validate-lmfs-dump-tapes: Site Object Attribute**

Specifies whether the LMFS backup dumper attempts to validate backup tapes. If the value is "yes", then the LMFS backup dumper validates backup tapes. If the value is not "yes" or if the attribute is not provided, no validation is done.

**terminal-f-argument: Site Object Attribute**

An associate set of specifications for what the various arguments to the FUNCTION F key should do. Each component is a triple consisting of a number (a string of the decimal number) or the string "none", a global name, and a set of hosts. The keywords can be:

**:login** The login file computer.

**:local-lisp-machines**
All Symbolics computers at this site.

**:all-lisp-machines**
All Symbolics computers on the local network.

**:host** The hosts in the third element of the triple.

```
(("none" :login nil)
  ("0" :read nil)
  ("1" :local-lisp-machines nil)
  ("2" :host (yellow blue read)))
```

### 1.2.6 Namespace Objects

A namespace object represents a mapping from names of objects to objects. The **name**, **search-rules**, and **descriptor-file** attributes are required; the rest are optional. Normally, no or one site is contained in each namespace.

**search-rules: Namespace Object Attribute**

Specifies the search rules, expressed as a set of namespaces (required).

```
(harvard yale)
```

**descriptor-file: Namespace Object Attribute**

Specifies the descriptor file for the namespace; a token (required). See the section "Namespace Database Descriptor Files", page 25.

```
blue:>sys>site>harvard-namespace.text
```

**primary-name-server: Namespace Object Attribute**

Specifies those hosts that are primary namespace servers for this namespace; a set of host objects. A primary server is an authority regarding its namespace. The namespace data are stored in files controlled by the primary namespace server.

```
(blue)
```

**secondary-name-server: Namespace Object Attribute**

A set of host objects, representing those hosts that are secondary namespace servers for this namespace. A secondary server is not an authority on a namespace, but can provide a backup in case the primary server is temporarily unavailable. It attempts to keep a copy of the namespace information current by querying the primary server more often than a nonserver machine would.

```
(orange pink)
```

**internet-domain-name: Namespace Object Attribute**

The Internet Domain Name associated with the namespace; a token. See the section "Dialnet and Internet Domain Names", page 85.

```
SCRC.Symbolics.COM
```

**user-property: Object Attribute**

Specifies a user-chosen property for this object; a pair whose first element is an indicator (by analogy with property lists) and whose second element is a token denoting whatever the user chooses to associate with that indicator.

```
(next-door-neighbor "Mr. Rogers")
```

## 1.3 User Interface to the Namespace System

To add new objects to the namespace database or edit existing ones, use the namespace editor. You can invoke the namespace editor by using the Edit Namespace Object command, clicking on [Namespace] in the System Menu, or calling **tv:edit-namespace-object**.

When editing an object that lives in more than one namespace (for example, a host that is on both the local Chaosnet and the Arpanet), a pop-up window appears that lists the name of each namespace and asks which "view" (that is, which namespace) you want to edit.

It lists each attribute (shown in Roman type) for the user object and its permissible data type(s). Attributes followed by an asterisk (*) are required; all others are optional. Data types are shown in regular italics and bold italics. By clicking on the data types shown in italics, you can enter values for the attribute. The value replaces the data type in the display. Data types shown in bold italics indicate compound values and remain in the display, as in `Login Name:` *Pair:* `sr VIXEN`. Existing values are also mouse-sensitive; click right on a value to edit it.

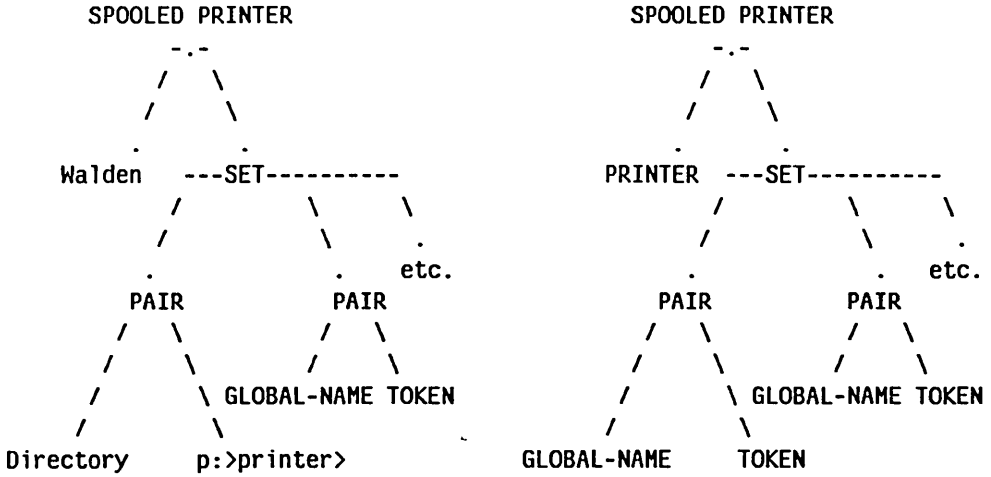
Some attributes can occur more than once for the same object (for example, login name); each attribute-value appears on its own line. An attribute with multiple values is not the same as an attribute having its value be a set.

Example: The spooled printer attribute entries for a particular host object look like this:

`Spooled Printer:` *Pair:* `WALDEN` *Set: Pair:* `DIRECTORY` *p:>printer> Pair:*
`Spooled Printer:` *Pair: Printer Set: Pair: Global-name Token*

The two entry lines for spooled printer indicate that the host object can have more than one such attribute. The second line, as yet not filled in, shows that the value of the spooled printer attribute is a pair, whose first element is a printer object and whose second element is a set of one or more pairs. Each pair consists of a global-name and a token. *Pair:* at the end of the first line indicates that you can continue to enter pair values for the printer object called Walden.

The following diagram illustrates the above entries.



## 1.4 Managing the Namespace Database

To run with the namespace database you must designate one machine as the primary namespace server for your namespace. This selection is made when software supporting the database is first installed at your site. The primary namespace server maintains permanent copies of the namespace database in some file system, usually its own, and distributes the information to other systems across the network.

### 1.4.1 Lisp Machine Namespace Server Files

The Namespace Server maintains four kinds of files to store the namespace information.

- Namespace database descriptor files
- Object files
- Log files
- Changes files

All of these are text files. Characters from a semicolon to the end of a line are considered to be comments.

#### 1.4.1.1 Record Format

The printed representation of an object and its attributes in files and over a network byte stream is in *records*. A record is a set of lines followed by a blank line. Each line is a set of *tokens* separated by spaces. A token is a sequence of characters except space, newline, semicolon, and double quote, or any sequence of characters between double quotes. Quoting within the double-quoted case is via the backslash character. Double quotes and backslashes must be quoted inside of double quotes.

For example,

```
SIZE EXTRA-LARGE
COLORS RED WHITE BLUE
MANUFACTURER "Symbolics, Inc."
SLOGAN "\"Yow!\", he said."
```

Due to the similarity to a property list, the first token in a line is called the *indicator* and the other tokens the *value*.

### 1.4.2 Namespace Database Descriptor Files

Each namespace has one *descriptor* file. Its pathname is stored as the **descriptor-file** attribute of the namespace. This file gives the locations of the other files which make up the namespace.

Each line of the file is either a comment or an indicator followed by a pathname.

Valid indicators are the names of the classes and the special indicators **version**, **changes**, and *.

| *Indicator* | *Value* |
|---|---|
| class name | The pathname of the *object* file for that class. See the section "Namespace Database Object Files", page 25. |
| **version** | The pathname of the *log* file. |
| **changes** | The pathname of the *changes* file. |
| * | The pathname of the *object* file for all classes that have not been explicitly named. See the section "Namespace Database Object Files", page 25. |

```
;-*-Text-*-
VERSION BLUE:>SYS>SITE>HARVARD-NAMESPACE-LOG.TEXT
CHANGES BLUE:>SYS>SITE>HARVARD-NAMESPACE-CHANGES.TEXT
HOST BLUE:>SYS>SITE>HARVARD-HOSTS.TEXT
USER BLUE:>SYS>SITE>HARVARD-USERS.TEXT
* BLUE:>SYS>SITE>HARVARD-OTHERS.TEXT
```

#### 1.4.2.1 Namespace Database Object Files

*Object* files, the heart of the namespace database, contain the stored attributes of all database objects. An object file contains the information for some subset of the classes in a namespace, as specified in that namespace's descriptor file. It begins with an file attribute list which specifies the namespace to which it belongs with the **network-namespace** attribute. This is followed by a series of records separated by blank lines.

Each record describes one object. The first line of the record consists of the class name and the primary name of the object. Each following line contains an indicator and a value for that attribute. Indicators defined as *elements* in the class definition may occur several times. The values are gathered together into a set.

A sample of HARVARD-USERS.TEXT, which contains only one record:

```
;-*- Mode: Text; Network-Namespace: Harvard -*-
USER GEORGE
LISPM-NAME George
PERSONAL-NAME "Washington, George"
HOME-HOST BLUE
MAIL-ADDRESS George BLUE
LOGIN-NAME George BLUE
LOGIN-NAME Washington.States MIT|MULTICS
LOGIN-NAME GW MIT|MC
NICKNAME Georgie
WORK-ADDRESS "The White House, Washington D.C., 10001"
WORK-PHONE 202-555-1212
HOME-ADDRESS "Mount Vernon VA"
HOME-PHONE 202-999-1234
PROJECT "being President of the United States"
SUPERVISOR "the People"
REMARKS "I cannot tell a lie."
```

#### 1.4.2.2 Namespace Database Log Files

The *log* file for a namespace is a text file containing all changes to the database. In addition, its file system version number is used as a timestamp for the change which resulted in that version being written out. This timestamp is used by the database system to identify obsolete data. For file systems that lack version numbers, such as UNIX, the creation date is used as a substitute for a version number.

An example from HARVARD-NAMESPACE-LOG.TEXT:

```
10/24/83 16:39:22 USER GEORGE by George.  Old timestamp was 607.
10/24/83 22:09:10 HOST BLUE by JAdams.  Old timestamp was 608.
10/26/83 07:23:45 HOST GREEN deleted by JAdams.
```

#### 1.4.2.3 Namespace Database Changes Files

The *changes* file for a namespace is a chronological record of all changes to the namespace. It is kept so that systems need only process changes since the last time they contacted the namespace server, rather than the entire database.

Each entry in the changes file consists of:

- a *timestamp* line an optional series of deleted object lines a blank line an optional series of changed or added object records

The timestamp line consists of the word TIMESTAMP followed by the version number of the log file before the change was made. Deleted objects are identified by their class name and primary name. Changed objects appear just as they do in the object file.

As changes are made, old entries in the changes file are winnowed. Thus, if an object is changed twice, only the newest record for it will appear. Older entries in the file are thus likely to consist of just a timestamp line and a blank line. Run **neti:prune-namespace-changes-file**.

An example from HARVARD-NAMESPACE-CHANGES.TEXT:

```
TIMESTAMP 607

USER GEORGE
LISPM-NAME George
PERSONAL-NAME "Washington, George"
HOME-HOST BLUE
MAIL-ADDRESS George BLUE
LOGIN-NAME George BLUE
LOGIN-NAME Washington.States MIT|MULTICS
LOGIN-NAME GW MIT|MC
NICKNAME Georgie
WORK-ADDRESS "The White House, Washington D.C., 10001"
WORK-PHONE 202-555-1212
HOME-ADDRESS "Mount Vernon VA"
HOME-PHONE 202-999-1234
PROJECT "being President of the United States"
SUPERVISOR "the People"
REMARKS "I cannot tell a lie."

TIMESTAMP 608

HOST BLUE
SYSTEM-TYPE LISPM
SERVICE CHAOS-STATUS CHAOS-SIMPLE CHAOS-STATUS
SERVICE SHOW-USERS CHAOS NAME
SERVICE TIME CHAOS-SIMPLE TIME-SIMPLE
SERVICE UPTIME CHAOS-SIMPLE UPTIME-SIMPLE
SERVICE LOGIN CHAOS TELNET
SERVICE SEND CHAOS SEND
SERVICE MAIL-TO-USER CHAOS CHAOS-MAIL
SERVICE NAMESPACE CHAOS NAMESPACE
SERVICE NAMESPACE-TIMESTAMP CHAOS-SIMPLE NAMESPACE-TIMESTAMP
SERVICE LISPM-FINGER CHAOS-SIMPLE LISPM-FINGER
SERVICE FILE CHAOS QFILE
LOCATION Kiosk 1
FINGER-LOCATION "Harvard Square Kiosk"
PRETTY-NAME Yellow
ADDRESS CHAOS 24412
MACHINE-TYPE LISPM
NICKNAME YEL
SHORT-NAME Y
SITE HARVARD

TIMESTAMP 609
HOST GREEN

TIMESTAMP 610
```

### 1.4.3 Namespace System Administrative Functions

**neti:read-object-file-and-update** *namespace class-name* *Function*

Update the namespace database from an object file. *namespace* can be a namespace object or the name of one. This function is used for namespaces which are maintained outside of the Symbolics namespace database, but which should be accessible to it. It reads an object file (usually generated from some external source of information) and makes the namespace database agree with it by adding, changing, and deleting objects. The changes and log files are updated. It can be invoked only on the primary namespace server for the namespace to be updated.

```
(neti:read-object-file-and-update
   :arpanet :host)
```

**neti:prune-namespace-changes-file** *namespace starting-timestamp* *Function*

Eliminate the record of changes to *namespace* before *starting-timestamp*. This reduces the amount of information which must be processed by the primary namespace server when it is booted. The changes file is best pruned only when there are no world load files that were saved before the earliest remaining change; they will take quite awhile to boot.

**neti:translate-hosts.text-file** &key *input-host-file input-lmlocs-file output-file allowed-prefixes destination-namespace short-names file-control-lifetimes server-machines tape-lisp-machines hosts-with-printers hosts-with-kanji-tablets* *Function*

Convert the M. I. T. files HOSTS.TEXT and LMLOCS into the namespace database format. Do only the Chaosnet part of hosts.text. Additionally, if the host is on the Arpanet, its primary name is given as a nickname to the corresponding name in the **arpanet** namespace, so that the object is properly shared. *allowed-prefixes* is a list of strings that can start the primary name. Give ("") to get everything, normally this would be ("MIT-"). *destination-namespace* is put in the file attribute list. The other arguments provide information that was not present in the M. I. T. files. Use of this function should generally be followed by using **neti:read-object-file-and-update** to update the database.

**neti:write-hosts.text-file** *output-file* &rest *args* *Function*

Writes an ITS-style HOSTS.TEXT file from the namespace database. This file can be used to initialize the the host table on a timesharing system from the database being maintained on the Symbolics computer.

## 1.5 Software Interface to the Namespace System

Symbolics computer programmers who want to use the capabilities provided by the network database should read this section. It describes the Lisp data types, variables, and functions for interacting with the network facilities.

### 1.5.1 Namespace System Lisp Data Types

The various database data types are implemented on the Lisp Machine as follows:

| | |
|---|---|
| object | An instance of some flavor based on **net:object**. |
| name | An instance of flavor **neti:name**. |
| global-name | A symbol in the keyword package. |
| token | A string. |
| set | A list. |
| pair | A list of two elements. |
| triple | A list of three elements. |

### 1.5.2 Namespace System Variables

**net:*local-site*** *Variable*

Specifies the site object representing the local site, that is, the value of this variable answers the question "What site am I at?"

**net:*local-host*** *Variable*

Specifies the host object representing the local host, that is, the value of this variable answers the question "What host am I?"

**si:*user*** *Variable*

Specifies the user object representing the user logged in to the machine, that is, the value of this variable answers the question "What user am I?"

**net:*namespace*** *Variable*

Specifies the current namespace object.

**net:*namespace-search-list*** *Variable*

Specifies the search rules, represented as a list of namespace objects.

### 1.5.3 Namespace System Functions

**net:find-object-named** *class name* &optional (*error-p* **t**) *Function*

Returns the object of the given *class* named *name*. *class* is a keyword symbol; *name* is a string. This function searches through all namespaces in

the search rules in order. If no object is found, the action taken depends on *error-p*:

| | |
|---|---|
| **t** | Signal a **neti:object-not-found-in-search-list** error. This is the default. |
| **nil** | Return **nil**. |

**net:find-object-named** also returns a second value, which is **t** if the object is valid and **nil** if it is not.

```
(net:find-object-named :host "apple")  =>  #<HOST APPLE>
(net:find-object-named :host "yale|orange")  => #<HOST YALE|ORANGE>
```

**net:find-object-from-property-list** *class property-list...* *Function*

Returns the first object of *class* that matches all of the properties in *property-list*. *class* is a keyword symbol; *property-list* is an alternating list of keywords and values. If no object is found, the function returns **nil**. If many objects are found, returns one of them. This function searches through all namespaces in the search rules in order.

For example, to find one UNIX host:

```
(net:find-object-from-property-list
  :host
  :system-type :unix)
```

**net:find-objects-from-property-list** *class property-list...* *Function*

Returns a list of all objects of *class* that match all of the properties in *property-list*. *class* is a keyword symbol; *property-list* is an alternating list of keywords and values. If no objects are found, it returns **nil**. Objects from all namespaces in the search rules are accumulated.

Example: To get a list of all Symbolics computers at the local site:

```
(net:find-objects-from-property-list
  :host
  :system-type :lispm
  :site net:*local-site*)
```

A property value from an object matches a pattern from the arguments to this function if one of the following conditions holds:

- The Lisp function **equal** returns true.
- The attribute is of the element or pair type and each element of the pattern list matches some element of the value; wildcards in the elements of a pattern are considered to match anything.

  A *wildcard* is the keyword symbol **:*** or the string **"*"**. (*Note:* The symbol ***** is not a wildcard.)

Example: To find a user who has an account on the blue host, use the :* to match any login name.

```
(net:find-objects-from-property-list
  :user
  :login-name '((:* ,(net:parse-host "blue"))))
```

**site-name** *Variable*

The value is a keyword, the name of the site at which this machine is located. **site-name** can be used to conditionalize programs. For example:

```
(when (eq site-name :acme)
  (load "apricot:>smith>cerebrum-server"))
```

Site names are described in more detail: See the section "Namespace System Site Objects", page 18.

**si:get-site-option** *keyword* *Function*

Finds out the value of a site option. *keyword* is the keyword symbol naming the option. This function returns the value of the option.

```
(si:get-site-option :timezone)
:EST
```

**si:parse-host** *host* &optional *no-error-p ignore* *Function*

*host* is a string representing the name of a host. The namespace database is searched for a host object corresponding to the name supplied. If the host is not found, an error is signalled unless *no-error-p* is supplied and is non-null.

### 1.5.4 Messages to Namespace Names and Objects

#### 1.5.4.1 Messages to neti:name

**:namespace** *Message*

Returns the namespace for the name.

```
(send (send si:*user* :name) :namespace) => #<NAMESPACE HARVARD>
```

**:qualified-string** *Message*

Returns a qualified character string representation of a name.

```
(send (send si:*user* :name) :qualified-string) => "HARVARD|GEORGE"
```

**:string** *Message*

Returns an unqualified character string representation of a name.

```
(send (send si:*user* :name) :string) => "GEORGE"
```

**:possibly-qualified-string** *Message*

Returns the qualified name if shadowed. The single argument is a class name. See the section "Shadowing Symbols" in *Reference Guide to Symbolics-lisp.*

```
(send (send si:*user* :name) :possibly-qualified-string :user) =>
"GEORGE" or "HARVARD|GEORGE"
```

### 1.5.4.2 Messages to neti:object

**:class** *Message*

Returns the name of the class of the object, as a keyword symbol.

```
(send net:*local-host* :class) => :host
```

**:get** *indicator* *Message*

Gets the value of this object's *indicator* attribute. *indicator* is a keyword symbol. If there is no such attribute, returns **nil**.

```
(send net:*local-host* :get :system-type) => :lispm
```

**:name** *Message*

Returns the primary name of the object, as a character string.

```
(send si:*user* :name) => #<NAME HARVARD|GEORGE 2346253>
```

**Note:** For compatibility with older releases, the **:name** message to a host returns a character string, rather than a name instance as would be expected. Use the **:primary-name** message to be sure of getting a name object.

**:primary-name** *Message*

Returns the primary name of the object, as a name instance.

```
(send net:*local-host* :primary-name)
#<NAME SCRC|TAGUS 36747263>

(send net:*local-host* :name)
"TAGUS"
```

**:names** *Message*

Returns a list of all of the names by which an object can be found.

```
(send net:*local-host* :names) => (#<NAME HARVARD|YELLOW 2346253>
                                   #<NAME HARVARD|Y 2346267>
                                   #<NAME HARVARD|YEL 2346303>)
```

**:user-get** *indicator* *Message*

Gets the value of this object's particular **user-property** attribute as indicated by *indicator*. *indicator* is a keyword symbol. If no such **user-property** attribute exists, **:user-get** returns **nil**.

```
(send si:*user* :user-get :favorite-color) => "Dusty Plum"
```

## 1.6 Implementation of the Namespace System

This chapter is for programmers who need to implement the namespace database on another system.

### 1.6.1 Network Namespace Protocol

Queries and updates to the network database are done over a byte stream with the *namespace protocol*. The general format of a request is a single record. The response is a series of records followed by a blank line. Queries can be serviced by a primary or secondary namespace server or by a non-server Symbolics computer; but in case of a secondary namespace server, the information in the response might be incomplete or out-of-date. Updates can be serviced by the primary namespace server only.

In the case of a query, you send a record which must at least specify a namespace and a class. Any additional attributes in the record are matched against objects in that namespace of that class. The response records describe those objects. Here the name of the object is given by the **name** attribute, rather than the value of the class name attribute. For attribute values that are pairs or elements, the special token * matches anything. Actually, * matches anything at any level, but putting it in as a value with a simple indicator is equivalent to leaving out that attribute entirely.

For example, the query

```
NAMESPACE MIT
CLASS HOST
NAME AI
```

might elicit the response

```
HOST MIT-AI
NICKNAME AI
SYSTEM-TYPE ITS
MACHINE-TYPE KA-10
ADDRESS CHAOS 2026
```

(Note the two blank lines at the end; the first ends the record describing MIT-AI. The second ends the blank record that marks the end of the response.)

Or the query

```
NAMESPACE MIT
CLASS HOST
SYSTEM-TYPE ITS
ADDRESS CHAOS *
```

might elicit

```
HOST MIT-AI
NICKNAME AI
SYSTEM-TYPE ITS
MACHINE-TYPE KA-10
ADDRESS CHAOS 2026

HOST MIT-MC
NICKNAME MC
SYSTEM-TYPE ITS
MACHINE-TYPE KL-10
ADDRESS CHAOS 1440
```

The format of an update is the same as that of a query, except that the additional **update-by** attribute is included. The value of this attribute is the user name of the person changing the information, for logging purposes. Additional tokens might be required by some servers for a password if security of the database is important.

A database deletion request has the special indicator **delete** in addition to **update-by**. The value of this attribute is the name of the object to be deleted from the database.

Incremental updates are accomplished in two ways. Any attribute list can have a **timestamp** indicator in addition to the match requests. The server reply lists only objects that have changed after that timestamp. In other words, the timestamp corresponds to the user's idea of when encached information was last valid.

A user can also request an incremental update of the database by supplying the **incremental** indicator. The value of this indicator is one of the special tokens **brief, full,** or **complete**. In this case, the **timestamp** indicator is mandatory and indicates from when the user is requesting an update. A brief incremental update starts with a record that is one of these:

- The word **current** if the timestamp supplied is still the correct timestamp for the namespace.
- A record with just a **too-old** attribute whose value is the current timestamp.
- A record that starts with a **timestamp** attribute whose value is the current timestamp and is followed by the class and name of each object that has been deleted from the namespace since the given timestamp. This last case is then followed by a record with a line giving the class and name of each object that has been changed or added to the namespace.

A **full** update has the same format as a changes file. See the section "Namespace Database Changes Files", page 26.

Finally, an **incremental complete** update results in one record containing a timestamp attribute for the namespace, followed by all the objects in the namespace.

### 1.6.2 Namespace Timestamp Protocol

A simple protocol is provided for determining whether any information in a namespace has changed. On the Chaosnet, this is implemented via a RFC/ANS transaction. The RFC specifies the name of the namespace and the corresponding ANS contains the timestamp as characters representing a decimal number.

### 1.6.3 Defining Namespace Classes

New namespace classes can be defined with the special operator **neti:define-class**. The definitions for the classes used in the system can be found in SYS:NETWORK;CLASS-DEFINITIONS.LISP.

# 2. The Lisp Machine Generic Network System

## 2.1 Networks and Addresses

The term *network* in this document means an entity that connects together a set of hosts. Any two hosts on the same network can communicate directly with each other. Each host has at least one *address* on the network; in order to talk to another host on the same network, you must know the address of that host.

The real world contains many distinct networks. The network database model takes account of this and provides abstract network objects to represent individual networks.

The Symbolics computer supports several different types of networks, including:

- Chaosnet
- Internet

The network type is determined by the level of protocol that (1) decides how addresses are represented and (2) figures out how to get a packet from a source address to a destination address. (This is the "network" level of the ISO OSI model.) Bear in mind that "type of network" does not refer to hardware. In particular, Ethernet is *not* a type of network; Ethernet cable can be used to transport Chaosnet packets or Internet packets (or both!).

Each type of network can support many actual networks. For example, two different Chaosnet networks can exist in the world, even though both are Chaosnets. The network database can represent this distinction. The network database also has a clear idea of what makes up "two different Chaosnets", as opposed to what it considers one. Two *subnets* (pieces of cable, as defined in *Chaosnet*) connected by a *bridge*, which automatically copies packets from one subnet to the next, make up a single network, not two networks. It is only one network because both subnets are using the same mapping of addresses into hosts; the same address means the same thing on each subnet. Bridges that connect subnets are invisible to the software described in this document, which considers a whole network to be one big "bus" configuration, with everything directly connected to everything else. The job of getting data from one host to another directly over a single network is below this level of modularity and therefore is not discussed herein.

However, this document does discuss the problem of getting information between two hosts that do not have any network in common. This is done with *gateways*: hosts that are on more than one network. If you are only on a network called jane and want to talk to a host that is only on network sarah, then you need to find a third gateway host that is on both networks jane and sarah. Then, by using a gateway service provided by that host, you might be able to communicate.

Sometimes it might even be necessary to have more than one intermediate host; a path to a destination might "hop" through many gateways.

In order to deal with getting information between two hosts that do not share a network, the local host needs to know what networks various hosts are on and what their addresses on those networks are. This information is stored in the network database, in the **:address** attribute of each host. When the local host wants to connect to some target host, it looks at the host objects for itself and for the target, compares the **:address** attributes, and looks for a network in common. If it does not find one, then it must find a more complex path. It searches through the database in search of a gateway host that shares some network with the local host and shares some other network with the target host. The details of the path of the communication, including the choice of gateway host, are determined entirely by the local host, on the basis of its examination of the database, and some information that it builds dynamically to estimate relative costs of different paths.

Each type of network has its own representation for addresses; the database just stores a string to represent an address. Each individual network is given those strings, and most, but not all, parse them into an internal numerical representation. For each type of network understood by the Symbolics computer, a set of procedures exists to do network-specific things such as address parsing and operating the network communications protocols.

Adding a new type of network to the Symbolics computer can be done in a straightforward way; the system is designed to be extensible. However, it requires a reasonable amount of programming and a good understanding of the modularity of the network system. Adding a new network of an existing type, at your site, is relatively easy and requires no programming; it is just an update to the database. Generally, user sites create new networks rather than network types.

## 2.2 Protocols and Services

The term *protocol* refers to a particular high-level network protocol, supported by server hosts, that provides some particular *service* to user hosts. (This definition is based on the concept of generic services.) Typical services include remote login service, mail delivery service, and file transfer service. When an application program needs to use networks, it calls a function in the network system, saying "Please get me this particular service on that particular host", or "Please get me this particular service on any host that provides it". Because many different ways exist to provide such service, the job of the network system is to figure out how to provide it in the most efficient way.

Each of the many different kinds of networks has its own protocols for providing the various services. For example, the service of getting mail to a user is provided by the **chaos-mail** protocol on Chaosnet, but by the **smtp** protocol on Internet. The

network system understands both of these protocols and uses the appropriate protocol to provide the service, based on how the local host is connected to the target host; for example, if both hosts are on an Internet together, the network system would decide to use the **smtp** protocol. This process is invisible to the application program.

The network system's selection of which protocol to use is actually more complex than the above example suggests, for the following reasons.

- Some protocols do not require a particular kind of network. For example, some protocols can operate over any network that is capable of providing a reliable byte-stream.

  The network system has to be aware of just what the need of each protocol is, and it also must understand when and how it can fulfill that need over different networks.

- Sometimes the two hosts are not connected by any kind of network at all but by a gateway. The network system must find a gateway host that provides the appropriate gateway service and then determine how communication can proceed over that gated connection.

  Note that provision of gateway service to another network is considered a generic service just like any other, and many different protocols can be used to implement gateway service.

When the network system needs to provide a service, how does it really select which protocol to use? Each service has a list of all protocols capable of providing that service. Each protocol has a demand: The protocol can provide the service only if an appropriate connection can be made to the target host. A connection is appropriate only if the appropriate *medium* of communication can be established between the local host and the target host. A medium represents the demand that a protocol makes.

In the database, each host has a **:service** attribute. The value is a list of all the services provided by this host. Each element of the list is a triple, of the form:

(*service medium protocol*)

Each element says that the host is capable of providing the given service, using the given protocol, if the network system can form a connection to it using the given medium. For example, suppose one element of a host's **:service** attribute were:

```
(:mail-to-user :chaos :chaos-mail)
```

This means that the host can provide the **:mail-to-user** service by using the **:chaos-mail** protocol, provided you can form a connection to it that supports the **:chaos** medium.

Note that services, media, and protocols are all represented as Lisp keyword symbols (*global names* in the network database).

Every host that provides services to other hosts has a list of entries of this sort. Usually, every host running a certain operating system and on the same types of network has the same entries. However, hosts can differ under certain circumstances; for example, one host might not offer a service even though it could, or one host has a special-purpose server that the others do not. Generally, though, the set of services that a host can provide, and the protocols that it provides those services with, are a function of what software runs on that host.

A single host might have two entries for the same service. This means that the host knows two different ways to provide a particular service, based on different media. If a host is on two different kinds of network, it might provide the same service in different ways on each network. For example, a host that was on both a Chaosnet and an Internet might provide **:mail-to-user** service on both networks, using different media and different protocols on each one. The network system picks which one to use, based on what kind of network connection it can establish.

The network system takes these steps to find you a path to service *s* on target host *h*.

1. It looks up *h* in the database and gets its **:service** attribute.

2. It looks for all of the entries for service *s*. Every time it finds one, it looks at the medium over which the service is provided and tries to find a path to *h* using that medium. The network system ignores any entries that mention protocols or media it does not understand.

3. It collects all the entries it finds during this search, and then at the end, chooses from the list of entries the most desirable one based on some cost criteria accumulated during the search.

## 2.3 Invoking Services: Lisp Machine Generic Network System

The simplest usage for invoking services requires the use of **net:invoke-service-on-host**.

**net:invoke-service-on-host** *service host* &rest *service-args* *Function*

*service* is a keyword symbol, *host* a host object. *service-args* and the values returned are service dependent. For example, the following invocation prints host MIT-MC's idea of the current time.

```
(time:print-universal-time
  (net:invoke-service-on-host :time (net:parse-host "MIT-MC")))
```

Whether or not **net:invoke-service-on-host** automatically tries all paths depends on the value of the variable **neti:*invoke-service-automatic-retry***.

**neti:*invoke-service-automatic-retry*** *Variable*

If the value of this variable is not **nil**, **net:invoke-service-on-host** automatically tries all paths. The default is **nil**.

### 2.3.1 Service Descriptions: Lisp Machine Generic Network System

Often more than one path to a service exists on a particular host, or more than one host provides a service. A *service access path* is a structure representing one of these paths, including the particular protocol to be used to get the service and how to establish any network connections needed.

Some protocols implement a service better or more efficiently than others. For these reasons, service access paths include a *desirability*, which is a number between 0 and 1 that represents how good a job this service access path does.

**net:find-paths-to-service** *service* *Function*

Returns a list of service access paths for the particular service and only one service access path for any given host. The list is sorted by decreasing desirability.

**net:find-paths-to-service-on-host** *service host* *Function*

Returns a list of all possible paths to a particular service on a given host. The list is sorted by decreasing desirability.

**net:find-path-to-service-on-host** *service host* *Function*

Returns a single access path or signals an error if none can be found.

**net:find-paths-to-protocol-on-host** *protocol host* *Function*

Similar to **net:find-paths-to-service-on-host**, except that the actual protocol is specified and only the network path is computed by the system. It is preferable to specify a service rather than a specific protocol in order to allow future transparent extension to a new protocol.

**net:find-path-to-protocol-on-host** *protocol host* *Function*

Similar to **net:find-path-to-service-on-host**, except that the actual protocol is specified and only the network path is computed by the system. It is preferable to specify a service rather than a specific protocol in order to allow future transparent extension to a new protocol.

**net:invoke-service-access-path** *service-access-path service-args* *Function*

Takes a service access path and returns the service dependent values, as **net:invoke-service-on-host** would.

Note that **net:find-paths-to-service** is not given *service-args*. This is because the service-path-finding mechanism does not implement a very fine weeding-out process. Network protocols and hosts are only known to implement a service, not that service under some restricted set of arguments, since that information is not present in the

network database. A higher-level mechanism must handle this filtering. For example, the network namespace service is sought only among hosts known to be namespace servers for the particular namespace desired.

**neti:most-desirable-service-access-path** *service-access-path-list* *Function*
Takes a list of service access paths sorted by desirability, as returned by **net:find-paths-to-service** or **net:find-paths-to-service-on-host**, and randomly chooses one from the equally desirable subset at the front. Since most paths to a service are equally desirable (such as a service provided by all Symbolics computers at the local site), this function should be used in preference to **first** for selection, since it distributes the server load evenly in the long run.

### 2.3.2 Service Futures: Lisp Machine Generic Network System

A *service future* is a request for a service whose connection establishment is outstanding. For simple services, like time, this allows you to have requests outstanding to more than one host at the same time. You can then pick the first or best of several responses without a long waiting period.

**net:start-service-access-path-future** *service-access-path* &rest *service-args* *Function*
Initiates the request for service. *service-access-path* and *service-args* are as for **net:invoke-service-access-path**. If the service is not implemented over the network, or the connection medium does not support asynchronous connections, the values **nil** and the values normally returned by this service are returned. Otherwise, the value **t** is returned.

**net:service-access-path-future-connected-p** *service-access-path* *Function*
Takes a service access path previously given to **net:start-service-access-path-future** and returns **t** if the connection is now complete. This can mean either successful or unsuccessful completion. This is useful for constructing wait predicates.

**net:continue-service-access-path-future** *service-access-path* *Function*
Takes a service access path which is connected (or which you have timed out on) and returns the values that invoking the service would. If the connection was not completed successfully, an error is signalled. If you are starting up several services but only looking for one answer, that means you must be prepared to intercept the error **sys:network-error** and go on to the next one. This is in practice necessary anyway, since byte-stream-oriented protocols can crash in the middle, datagram-oriented protocols can return malformatted answers that are not detected by the NCP itself, and so on. The **net:invoke-multiple-services** macro aids in writing programs that do this.

**net:abort-service-access-path-future** *service-access-path* *Function*

Takes a service access path previously given to **net:start-service-access-path-future** and cancels the outstanding connection. Useful for cleanup handlers.

**net:invoke-multiple-services** *(services timeout* &optional *whostate service-variable) (host* &rest *service-results)* &body *clauses* *Special Form*

A useful macro for dealing with multiple paths to a service at once. It starts up futures for multiple hosts, running the specified code when each is finished.

| | |
|---|---|
| *services* | A form that will return a list of service access paths, for example, a call to **net:find-paths-to-service**. |
| *timeout* | The maximum time to wait for any one host to respond, in sixtieths of a second. |
| *whostate* | Optional; the state to put in the status line while waiting for a future to complete. Defaults to **"service wait"**. |
| *service-variable* | Optional; the name of a variable to be bound to the service access path describing the service. |
| *host* | A variable name to be bound to the host on which the service was invoked. |
| *service-results* | Variables to be bound to the results of invoking the service. |
| *clauses* | Clauses as for **condition-case**. Actually, that means that the *service-results* variables are bound inside the **condition-case** form, so that the first of *service-results* would be the error object if an error were generated. |

For example:

```
(defun all-hosts-time ()
  (net:invoke-multiple-services
      ((net:find-paths-to-service :time) (* 60. 10.) "Time")
      (host time)
    (sys:network-error
      (format t "~&~A: ~A" host time))
    (:no-error
      (format t "~&~A: ~:[unknown~;~\TIME\~]"
              (if (eq host net:*local-host*) "local" host)
              time time))))
```

## 2.4 Finding Paths to Hosts: Lisp Machine Generic Network System

The description of how the network system decides what protocol to use leaves open a question: How does the network system find a path to a host using a particular medium? Each medium is defined by a special form called **net:define-medium**. The definition of a medium includes a set of alternative *implementations* of the medium, each of which describes a way to form a network connection using that medium.

Each implementation contains one or more *steps*. A one-step implementation is a way to connect directly to the target host. A two-step implementation is a way to connect first to a *gateway* (a host on more than one network), which then connects, in turn, to the target. (A three-step implementation would be a way to go through two levels of gateway; none of the defined media actually do this, but it could be done to any number of levels.)

Steps are of the following three types:

**:network**
**:medium**
**:service**

The last step of any implementation must be either **:network** or **:medium**; steps other than the last step must be **:service**. This means that a one-step path must be either **:network** or **:medium**.

Steps and implementations are represented as lists. An implementation is a list of steps. A step is a two-element list whose first element is the type of step (either **:network**, **:medium**, or **:service**).

The three types of steps are defined as follows:

**(:network** *network-type***)**
: Succeeds if the local host and the target host are both on some network of type *network-type*. The connection can be formed directly over that network. When a function is associated with the step, it is a function to actually open the connection over the network.

**(:medium** *medium***)**
: Succeeds if a connection can be formed over *medium*. When a function is associated with the step, it is a function to create the encapsulated connection. See examples below.

**(:service** *service***)**
: Succeeds if a connection can be formed to a server providing *service*, and that server can connect to the next step. Functions are never associated with **:service** steps.

The body of a **net:define-medium** form is made of implementations, or implementations associated with functions.

**net:define-medium** *medium ([built-on-medium]) . body* *Special Form*

Defines a medium named *medium*, which is built on *built-on-medium*, if *built-on-medium* is supplied. An element of the body can either be an implementation, or a list of the following form:

(*implementation lambda-list . body*)

The latter syntax provides a function associated with the last step of the implementation. Note that in a multi-step implementation, steps before the last must be **:service** steps, which cannot have an associated function.

The following form defines a medium called **:tcp** (the Transmission Control Protocol of the Department of Defense) and provides two implementations. The first one says that you can establish a **:tcp** connection with a host if you are on the same **:internet** as it. The second one says that you can establish a **:tcp** connection by finding a path to any gateway host that provides the **:tcp-gateway** service, and that can, itself, form a **:tcp** connection to the target host.

Note that the last step is a **:medium** step. This allows many levels of gateway to be used.

```
(define-medium :tcp (:byte-stream)
  ((:network :internet))
  ((:service :tcp-gateway) (:medium :tcp)))
```

To establish a **:chaos** or **:chaos-simple** connection to a target host, you both have to be on a **:chaos** network together. There is an associated Lisp function provided to open the connection. Note that the keyword **:chaos** is being used in two independent ways here: as a medium, and as a network type.

```
(define-medium :chaos (:byte-stream)
  (((:network :chaos)) (service-access-path &rest connection-args)
   body))

(define-medium :chaos-simple (:datagram)
  (((:network :chaos)) (service-access-path &rest connection-args)
   body))
```

There was no Lisp function in the TCP example because on the assumption that there is no actual network control program in the machine for TCP protocols. The **(:network :internet)** clause is still useful even though Symbolics computers cannot connect directly, because it tells the network system that a gateway and a target can connect using medium **:tcp** if they are both on an **:internet** network.

The following **:dial** example is just like **:chaos**, and the **:x25** example is just like **:tcp**. Again, note that the symbols **:dial** and **:x25** are being used in two distinct ways: as media, and as network types.

```
(define-medium :dial (:byte-stream)
  (((:network :dial)) (service-access-path &rest connection-args)
   body))
```

```
(define-medium :x25 (:byte-stream)
  ((:network :x25))
  ((:service :x25-gateway) (:medium :x25)))
```

The **:mmdf** example has something new: a Lisp function associated with a **:medium** step. This is an *encapsulated* medium. MMDF is an error-correcting "protocol" that creates a reliable byte-stream, building it out of an unreliable byte-stream. The Lisp procedure takes the connection opened by the **:dial** or **:x25** medium, and returns a new connection that uses the first connection but also supplies MMDF error-correction.

```
(define-medium :mmdf (:byte-stream)
  (((:medium :byte-stream))
       (service-access-path stream &rest connection-args)
   body))
```

The **:pseudonet** medium always uses a gateway to access a network of type **:gateway-pseudonet**. This is explained later in the documentation of the **:gateway-pseudonet** network type; it is used for accessing hosts that are not really on a network but are connected to some other host via something weaker, like serial lines.

```
(define-medium :pseudonet (:byte-stream)
  ((:service :pseudonet-gateway) (:network :gateway-pseudonet)))
```

In order to explain the following two forms, we have to explain the *built-on-medium* argument.

```
(define-medium :byte-stream ())

(define-medium :datagram ())
```

Whenever a medium **:x** uses *built-on-medium*, it is as if the medium *built-on-medium* is given a new implementation, namely **(:medium :x)**. So, it is as if the definition of these media really were

```
(define-medium :byte-stream ()
  ((:medium :tcp))
  ((:medium :chaos))
  ((:medium :dial))
  ((:medium :x25))
  ((:medium :mmdf))
  ((:medium :pseudonet)))

(define-medium :datagram ()
  ((:medium :chaos-simple)))
```

In other words, the **:byte-stream** medium can be implemented out of any of **:tcp**, **:chaos**, and so on, and the **:datagram** medium can be implemented out of **:chaos-simple**.

These are useful because some protocols are written in such a way that they only need generic byte streams, or generic datagrams, and do not care about the details of how those things are implemented. Such generic protocols are more useful than nongeneric protocols, because they can operate over many different kinds of network.

The network system allows you to have both generic and nongeneric protocols. The generic ones are more flexible, because they can operate over many kinds of network, but the nongeneric ones can sometimes take advantage of the features peculiar to a specific network in order to provide higher performance or special services. Of course, a dualistic distinction between generic and nongeneric does not really exist; one medium can be implemented out of another, which is implemented out of a third, and so on. The structure is really a directed graph rather than a pair of layers.

Generic media rarely appear in the **:service** attributes of host objects. If a host claimed to provide some service over the **:byte-stream** medium, it would have to support *every* kind of medium that is built on **:byte-stream**, which is unlikely. Generic media often appear in protocol definitions, however, and when the protocol is used, a specific nongeneric medium is chosen based on what is found in the **:service** attribute of the target host.

As the following example indicates, there is actually a fourth kind of step, called **:local**, which is only used by the medium named **:local**. The **t** is just a placeholder. Some network services can be satisfied locally, without actually using the network. For example, some computers have their own built-in time-of-day clocks, and so a server can be provided for the time-of-day service that is implemented out of this medium.

```
(define-medium :local ()
  ((:local t)))
```

Here is an extensive example to show how all this works. In this example, there are three hosts, named: Muddy, Collie, Boston-PAD.

Muddy and Collie are Symbolics computers, and Muddy is trying send mail to Collie. Boston-PAD is a computer owned by GTE-Telenet to which you can dial up to connect to hosts on Telenet (GTE-Telenet provides an X.25 long-haul network as a common carrier). Muddy has a dial-out unit, and Collie is connected to another Telenet PAD over a serial line. For purposes of this example, Collie and the Telenet PAD to which it is connected are considered a single host; the Telenet PAD is considered a peripheral device.

Three networks are involved.

- A Chaosnet that connects Muddy directly to Collie; its name is Chaos and its network type is **:chaos**.

- A **:dial** network that Muddy and Boston-PAD are both on; its name is Dial and its network type is **:dial**. The only network of type **:dial**, it is physically

implemented out of the international phone network and any host with appropriate modems is on it.

- GTE-Telenet's long-haul network; its name is Telenet and its type is **:x25.**

All of this information is in the network database. The database includes three host objects and three network objects. The state of which host is on which network is expressed in the **:address** attributes of the host objects. Muddy's **:address** attribute has two elements, one for Chaos (giving its octal Chaosnet address) and one for Dial (giving its phone number as the address). Boston-PAD has two addresses also, one for Dial and one for Telenet. Collie has two addresses, one for Chaos and one for Telenet.

The host objects also have **:service** attributes. Collie's **:service** attribute includes two entries of interest to us:

```
(:mail-to-user :chaos :chaos-mail)
(:mail-to-user :mmdf :chaos-mail)
```

This says that Collie can provide **:mail-to-user** service, using either the **:chaos** medium or the **:mmdf** medium, in either case using **:chaos-mail** protocol.

Boston-PAD's **:service** attribute has one interesting entry:

```
(:x25-gateway :dial :telenet-pad)
```

This says that Boston-PAD can provide **:x25-gateway** service, using the **:dial** medium and the **:telenet-pad** protocol.

Since Muddy wants to send mail to Collie, it calls the network system, asking it to find a path to Collie that provides the **:mail-to-user** service. Muddy calls **neti:find-paths-to-service-on-host**. See the section "Service Descriptions: Lisp Machine Generic Network System", page 41. The network system looks at the **:service** attribute on Collie and finds two entries for the **:mail-to-user** service. It considers each in turn.

The first entry says that Collie can provide the service if Muddy can connect using the **:chaos** medium. To figure out whether it can or not, the network system looks at the **net:define-medium** special form for the **:chaos** medium and finds that success can be achieved if Muddy and Collie are both on some network of type **:chaos**. So it examines the **:address** attribute of both host objects together and finds that, indeed, both are connected to such a network, namely Chaos. The network system has found the first path.

Next, the network system tries the second **:service** entry, which says that Collie can provide the service if Muddy can connect to it using the **:mmdf** medium. So the network system looks at the definition of **:mmdf** to see how this might be done. There are two possible implementations, and it will consider each of them.

The first implementation of **:mmdf** requires a connection using the **:dial** medium; this becomes a new subgoal. So the network system examines the definition of the

**:dial** medium and finds that the two hosts need to be on the same **:dial** network. It examines the **:address** attributes of Muddy and Collie and does not find such a network; indeed, Collie is not on any **:dial** network (it has no modem). So this subgoal fails, and the network system moves on to the next possible implementation of **:mmdf**.

The second implementation of **:mmdf** requires a connection using the **:x25** medium; this becomes a new subgoal. So the network system examines the definition of the **:x25** medium and finds two possibilities. First, a connection can be formed if the two hosts are on the same **:x25** network. Unfortunately, the **:address** attributes show that they are not (Muddy is not on Telenet). This has failed, so the network system tries the other possibility: a two-step path, which becomes a new subgoal.

In order to achieve this new subgoal, two conditions must be satisfied, namely the two steps of the path. The first step is to find a host that provides **:x25-gateway** service, and the second step is to find a way to connect that host to the target host using the **:x25** medium. So the network system searches the network database for hosts that provide **:x25-gateway** service. This is a recursive call; originally we were looking for **:mail-to-user** service, but now we are looking for **:x25-gateway** service. A difference is that this time we are not asking for the service on a specific host but on any host; the function **net:find-path-to-service** is used. See the section "Service Descriptions".

The network system searches the database for **:x25-gateway** servers, prepared to check each one to see whether it can connect via **:x25** to the target host. It finds Boston-PAD, whose service attribute says that it will provide **:x25-gateway** service if Muddy can connect to it using the **:dial** medium. **:dial**'s definition says that the two hosts both have to be on the same **:dial** network; fortunately, they are, namely the Dial network.

Now it has to see whether Boston-PAD can connect to Collie using **:x25**. The first implementation in the definition of the **:x25** medium says that you can connect if the two hosts are on the same **:x25** network; fortunately, they are, namely Telenet.

The network system has now succeeded a second time in its top-level goal. The answer is a list of two paths. One is a one-step path, going directly through the Chaos network. The other is a two-step path, going through the Dial network to the Boston-PAD host, and from there through the Telenet network to Collie.

## 2.5 Defining Protocols: Lisp Machine Generic Network System

Each network protocol has two implementations, a *server* and a *user*. A *server* runs in the system on which the service was invoked, and actually performs the service. A *user* runs in the system which requested the invocation and uses the network to cause the server to run. In the case of a locally provided service, the server is called as a subroutine by the user.

### 2.5.1 Users: Defining Protocols: Lisp Machine Generic Network System

**net:define-protocol** *name (service base-medium)* &body *options* *Special Form*
Defines the protocol *name*, a keyword symbol, which provides the service *service*. *base-medium* is the minimum medium needed for this protocol; it can be a specific medium, such as **:chaos** for protocols that require those features, or **:datagram** or **:byte-stream**, for more generic protocols. Also, specially, it can be **:local**, meaning that the protocol is not implemented in the network at all, but via some functions running on the local machine.

*options* are each a list whose first element is a keyword. Defined keywords are:

(**:desirability** *number*)
*number* is a number between 0 and 1 that describes how well this protocol provides the service. The default is 1.

(**:property** *indicator property*)
Used for higher level protocol-defining macros that save their own information.

(**:invoke** *function*) When the service is invoked, *function* is called with the service access path as an argument.

(**:invoke-with-stream** *function*)
Similar to **:invoke**, except that a network stream is gotten first via the appropriate medium, using **net:get-connection-for-service**, and it is the argument to *function*. The first argument to *function* is the stream, and the remaining arguments are the arguments to the service invocation.

(**:invoke-with-stream-and-close** *function*)
Similar to **:invoke-with-stream**, except that the stream is closed when *function* returns.

In **:invoke**, **:invoke-with-stream**, and **:invoke-with-stream-and-close**, *function* can either be a symbol, which is the name of a function, or the rest of the list can be a lambda-list and body for the function. For **:invoke-with-stream** and **:invoke-with-stream-and-close** the first element of the lambda-list is the stream variable, which will be bound to the stream returned by **net:get-connection-for-service**; the other elements are arguments to the service invocation. See the function **net:get-connection-for-service**, page 51. If you want to pass *connection-args* to **net:get-connection-for-service**, the first element of the lambda-list should *not* be a stream variable, but rather a list whose first element is the stream variable and whose other elements are the *connection-args*.

For example, the following defines a local version of the time service. Note that **nil** is returned if the time is not known locally. In general, how a protocol indicates that it cannot provide a service after all is defined by the service itself. For some services, such as time, this is done via the returned value. For others, an error would be signalled. This error could then be caught by the **net:invoke-multiple-services** macro.

```
(net:define-protocol :local-time (:time :local)
  (:invoke (ignore)
   (and time:*time-is-known-p*
        (time:get-universal-time))))
```

The following example defines the Chaosnet RFC/ANS version of the time protocol. **time-simple** is a function that just takes the bytes from the **:read-input-buffer** message to **stream** and deposits them together into a 32-bit time, returning **nil** if the datagram is malformatted (for example, does not contain exactly four data bytes).

```
(define-protocol :time-simple (:time :datagram)
  (:desirability .75)
  (:invoke-with-stream-and-close (stream)
   (time-simple stream nil)))
```

**net:get-connection-for-service** *service-access-path* &rest *connection-args* *Function*

Can be used inside of a **:invoke** clause to get a network stream to the service on the correct medium. *connection-args* are passed on to the stream creator; normally they would be keyword pairs such as **:ascii-translation t**, specifying that the ASCII character set was to be used over the network.

This gets the contact identifier from the protocol field of the service access path, over the medium given by the medium field.

Higher-level protocols such as login and file provide their own mechanisms for informing the service system of implementation of new protocols. These are macros that expand into a **net:define-protocol** form with suitable options.

### 2.5.2 Servers: Defining Protocols: Lisp Machine Generic Network System

A network *server* does not actually provide a generic service but rather implements a specific protocol. This protocol works over some specific medium, which may or may not be generic.

**net:define-server** *protocol-name options* &body *body* *Special Form*

Define the top-level function of a network server. *protocol-name* is a keyword, the same as for **net:define-protocol**. *options* is an alternating list of keywords and values. Some of these keyword-value pairs specify the names of variables which are bound inside *body*, which is the server itself. How the names are extracted is explained presently. This is in fact

implemented by the system's defining a function whose arguments are those variables and whose body is *body*.

The main keyword in the options list is **:medium**, whose value is a keyword specifying the medium type over which this protocol operates. Normally, this is a generic medium, such as **:byte-stream** or **:datagram**. Sometimes it is a specific medium, such as **:chaos**. It is usually preferable to use the generic medium, when possible, even if the protocol is only used over some particular type of network.

The following other keywords are recognized for all values of the **:medium** keyword.

**:address** The value of this keyword is the name of a variable that is bound to the parsed address of the host that is the user of the server.

**:error-disposition**
A keyword that determines what should happen if an error occurs in the server. Valid error dispositions are:

**nil** or **:notify** A notification is given when any error occurs and the server exits (abnormally because of the error). **:notify** is the default.

For finer control of error notification, you can specify the **:notify** keyword with one or more error flavors, as follows: (**:notify** *error-flavor-1 error-flavor-2* ...). For example, `:error-disposition (:notify` sys:remote-network-error) means send notifications of errors of the **sys:remote-network-error** flavor and ignore all others.

**:ignore** The server exits but no notification is given. As with **:notify** you can exercise finer control over error notification by specifying one or more error flavors with the **:ignore** keyword. For example, `:error-disposition (:ignore` sys:remote-network-error) means ignore errors of the **sys:remote-network-error** flavor but notify for all others.

**:debugger** The server process enters the debugger when an error occurs.

**:host** The value of this keyword is the name of a variable that is bound to the host object that is the user of the server.

**:network** The value of this keyword is the name of a variable that is bound to network object through which user is connected.

**:process-name** A string, defaulting to /"*protocol-name* server/", which is the name of the process created to run the server.

**:reject-unless-trusted**
The value of this keyword is **t** by convention. It causes the server request to be rejected if the host wanting the service is not trusted.

**:trusted-p** The value of this keyword is the name of a variable that is bound to **t** if the host using the service is /"trusted/".

**:who-line** The value of this keyword is **t** by convention. It causes a message to be displayed in the status line while the server is active. It also causes the server to appear in the Peek active server display.

The following keywords are recognized for the **:byte-stream** medium type.

**:stream** The value of this keyword is either a symbol, which is the name of a variable that is bound to a bidirectional stream, or a list of such a variable name and alternating keyword and value options that specify how the stream is made. Keywords at this level are:

**:ascii-translation** The protocol uses the ASCII character set rather than the Symbolics character set.

**:accept-p** If **nil**, **:accept-p** says that the stream should not be fully opened, but the *body* is allowed to decide whether to accept, by sending the **:accept** message, or reject the service by sending the stream a **:reject** message along with a reason for rejection.

**:direction** **:input** or **:output** if server needs only one direction. Note that the connection itself is bidirectional, but the stream accepts only one class of messages. Default is a bidirectional stream.

**:no-close** The value of this keyword is **t** by convention. It causes the network stream to be left untouched when the *body* returns, rather than closed or aborted. This is used for some protocols in which closing the stream is part of the protocol.

**:no-eof** The value of this keyword is **t** by convention. It causes the

network stream to be aborted when the *body* returns, rather than closed. This is used for some protocols in which closing the stream is part of the protocol.

The following keywords are recognized for the **:datagram** medium type.

**:request-array** The value of this keyword is a list of three variable names, which are bound to an array, its starting index, and its ending index. If any of the variable names is **nil**, or the list is not long enough to include it, no such variable is bound. The array within the given bounds contains any arguments to the service that the user specified. On the Chaos network, that means that it points to the portion of the RFC packet after the space following the contact name.

**:response-array** A list of variable names like **:request-array**. The server fills in the array with the response data and returns two values; the first is **t**, if the service is successful, or **nil**, if the request should be rejected. The second value is the byte index after the last byte stored in the array. Alternatively, the *body* can return a second value that is a string, which the system will store as the contents of the array itself. In that case, it is not necessary to specify the **:response-array** keyword.

The **:chaos** medium is provided for compatibility with older Chaos network protocols that are inconvenient to implement over a generalized medium, or for people who do not wish to spend time completely updating a program that used to use **chaos:server-alist**. The **:conn** keyword is used with this medium; the value is a variable to be bound to the Chaos connection, which will be in RFC-Received state. It is not necessary to do a **chaos:listen**. It is still necessary to do **chaos:accept** or **chaos:reject** as appropriate, and to do **chaos:remove-conn** when done.

**chaos:add-contact-name-for-protocol** *protocol* &optional (*contact-name* **(string** *protocol*)) *Function*

Creates an association between a protocol and a Chaosnet contact name when opening connections. *protocol* is a keyword that identifies the protocol. *contact-name* is a string that the Chaosnet uses when opening a connection (sending an RFC or listening for a request). *contact-name* defaults to **(string** *protocol*).

Examples:

```
(chaos:add-contact-name-for-protocol ':11load)
(chaos:add-contact-name-for-protocol ':chaos-status "STATUS")
```

**neti:with-server-error-disposition** *server* &body *body* *Macro*

Creates an environment for handling errors within a server. Using the server's **error-disposition** property, this macro sets up a **condition-case-if** to handle any errors not caught by the server itself.

A server's **error-disposition** property is set in one of two ways: by explicit specification when the server is defined (using the **:error-disposition** keyword argument to **define-server**) or by explicitly changing the **error-disposition** of a defined server with the **neti:change-server-error-disposition** function.

A server's **error-disposition** property is ignored when **neti:*server-debug-flag*** evaluates to something other than **nil**; if this is the case, the server process always enters the Debugger on an error not caught by the server itself.

Note that the environment for error disposition is set up when the server is started, and subsequent use of **neti:change-server-error-disposition** or binding of **neti:*server-debug-flag*** has no effect on that server.

**neti:change-server-error-disposition** *protocol-name new-disposition* *Function*
Changes the error disposition for the server handling *protocol-name*. Valid dispositions are the same as those used in **net:define-server**.

### 2.5.3 File Users: Defining Protocols: Lisp Machine Generic Network System

The Symbolics pathname system provides generic file access regardless of where a file resides, what operating system runs on the file system, or how files are named on that operating system. An instance of a pathname cannot contain specific information on how to access the file system, however, since that information might change as a world load containing the pathname is moved from one host to another or from one site to another. For this reason, the network system provides *file access paths*. Generic pathname operations on most pathnames pass the message on to the file access path for the file system.

Usually, a file access path is a user implementation of some file transfer and manipulation protocol. A file access path is found via the normal service lookup mechanism. Invoking a service access path for **:file** service returns a file access path. The **fs:define-file-protocol** macro is used to interface the service and file access path subsystems.

## 2.6 Defined Media: Lisp Machine Generic Network System

### 2.6.1 Byte Stream Media: Lisp Machine Generic Network System

| | |
|---|---|
| **byte-stream** | medium |
| **chaos** | medium |
| **dial** | medium |
| **tcp** | medium |

**pseudo-net** medium

### 2.6.2 Datagram Media: Lisp Machine Generic Network System

**chaos-simple** medium

## 2.7 Defined Services and Protocols: Lisp Machine Generic Network System

### 2.7.1 band-transfer Service

**band-transfer** service

**band-transfer** protocol

### 2.7.2 file Service

**file** service

| | |
|---|---|
| Returns | **file-access-path**. See the section "File Users: Defining Protocols: Lisp Machine Generic Network System", page 55. |
| User-protocols | **local-file (local)** - access to the local file system directly. |
| | **qfile (chaos)** - See the section "Chaosnet File Protocol", page 159. |
| | **tcp-ftp (byte-stream)** - IEN149 |
| Server-protocols | **qfile** |

**local-file** protocol

**qfile** protocol

**tcp-ftp** protocol

### 2.7.3 hardcopy Service

**hardcopy** service

| | |
|---|---|
| Arguments | Printer-object Options |
| | Options as to **si:make-hardcopy-stream** |
| Returns | A stream that outputs the hardcopy device eventually, possibly through a spooler. The |

characters to send to the stream are format specific. Programs should almost always use **si:make-hardcopy-stream** instead, which returns a stream that accepts device independent messages. See the function **si:make-hardcopy-stream** in *Text Editing and Processing*.

User-protocols **local-hardcopy (local)** - connects directly to a spooler running on this machine or a directly attached hardware device.

**lgp (chaos)** - contact name is

chaos <*filename*><line><*printer-name*><line><*user-name*><line><*user-personal-name*><line><*date*>

(where <*date*> is a decimal universal time.)

**dover(chaos)** - via **ai-chaos-11**.

Server-protocols **lgp**

**dover** protocol

**eftp** protocol

**lgp** protocol

**local-hardcopy** protocol

### 2.7.4 hardcopy-device-status Service

**hardcopy-device-status**
service

Prints some descriptions of the status of the device on standard-output.

Arguments printer-object

**user-protocols** lgp-status **(datagram)**

**rfc** is **lgp-status <printer>**.

**ans** is status information.

**ears-status (datagram)**

**dover-status (datagram)** - Amber 5.10. **ai-chaos-11** version of above.

Server-protocols **lgp-status**

**dover-status** protocol

**lgp-status** protocol

### 2.7.5 hardcopy-status Service

**hardcopy-status** service

Prints some descriptions of the status of the device and its spooler on standard-output.

| | |
|---|---|
| Arguments | printer-object |
| **user-protocols** | lgp-queue (**byte-stream**) |
| | Contact name is **lgp-queue <printer>**. |
| Server-protocols | **lgp-queue** |

**ears-status** protocol

**lgp-queue** protocol

### 2.7.6 lispm-finger Service

**lispm-finger** service

| | |
|---|---|
| Returns | list of (user-name machine-location idle-time personal-name affiliation-letter) |
| User-protocols | **lispm-finger** (**chaos**) - Amber 5.7. |
| Server-protocols | same |

**lispm-finger** protocol

### 2.7.7 login Service

**login** service

Returns four value:

- A stream to the remote virtual terminal.
- A set of input side filters.
- A set of output side filters.
- A string describing the route used, suitable for a window's label.

The middle two are specific to the Terminal program. The **si:define-login-protocol** special form is described elsewhere: See the section "File Users: Defining Protocols: Lisp Machine Generic Network System", page 55.

User-protocols **supdup (byte-stream)** - RFC734

**telnet (byte-stream)** - IEN148

**telsup (byte-stream)** - input side is like **supdup**, output side is like **telnet** with Imlac simulation enabled.

**tty-login (byte-stream)** - protocol suitable for talking ascii to a normal serial terminal line.

Server-protocols **telnet**

**chat** protocol

**eval** protocol

**supdup** protocol

**telnet** protocol

**telsup** protocol

**tty-login** protocol

### 2.7.8 mail-to-user Service

**mail-to-user** service

Sends mail via some host. **mail-to-user** sends mail to a mailbox on that host. The format of **mail-to-user** is Zmail specific. See the section "**store-and-forward-mail** Service", page 62.

Arguments recipients template-expansion

User-protocols **smtp (byte-stream)** - RFC821

**chaos-mail (chaos)** - Amber 5.5.

**dummy-mailer (local)** - Implements store-and-forward-mail in terms of mail-to-user. Takes the place of having the real mailer loaded.

Server-protocols **chaos-mail.**

**chaos-mail** protocol

**smtp** protocol

### 2.7.9 namespace Service

**namespace** service

| | |
|---|---|
| Returns | byte-stream to namespace server talking network namespace protocol. See the section "Protocols and Services", page 38. |
| User-protocols | **namespace (byte-stream)**, IEN-811 **(byte-stream)** |
| Server-protocols | **namespace.** |

**ien-811** protocol

**namespace** protocol

### 2.7.10 namespace-timestamp Service

**namespace-timestamp** service

| | |
|---|---|
| Arguments | Namespace-object |
| Returns | Timestamp for that namespace. |
| User-protocols | **namespace-timestamp (datagram)** See the section "Datagram Media: Lisp Machine Generic Network System", page 56. |
| Server-protocols | same |

**namespace-timestamp** protocol

### 2.7.11 notify Service

**notify** service

**notify** protocol

### 2.7.12 packet-gateway Service

**packet-gateway** service

Performs packet forwarding

### 2.7.13 print-disk-label Service

**print-disk-label** service

Prints a description of the host's disk label to standard-output.

User-protocols **local-print-disk-label (local)** - interface for printing local label.

**print-disk-label (byte-stream)**

Server-protocols **print-disk-label.**

**local-print-disk-label**
protocol

**network-print-disk-label**
protocol

### 2.7.14 pseudonet-gateway Service

**pseudonet-gateway**
service

Connects to another host through a gateway attached to its serial line, etc.

User-protocols **pseudonet-gateway (chaos)** - Contact name encoded in "address".

**pseudonet-gateway**
protocol

### 2.7.15 screen-spy Service

**screen-spy** service

**chaos-screen-spy** protocol

### 2.7.16 send Service

**send** service

Sends an interactive message containing text to the user on the particular host.

Arguments &key *date from to text*

User-protocols **smtp (byte-stream)** - RFC821

**send (chaos)** - Amber 5.6.

Server-protocols same

**send** protocol

**smtp** protocol

### 2.7.17 show-users Service

**show-users** service

Prints "finger" information on standard-output, optionally for user, optionally with' "whois" information.

Arguments &key *user whois*

User-protocols **ascii-name (byte-stream)** - RFC742

**name (chaos)** - Amber 5.7.

Server-protocols same

**ascii-name** protocol

**name** protocol

### 2.7.18 status Service

**status** service

**chaos-status** protocol

### 2.7.19 store-and-forward-mail Service

**store-and-forward-Mail**
service

Sends mail via some host. **store-and-forward** send mail (possibly) to some user on another host, to which this host is presumably closer. The format of **store-and-forward-mail** is Zmail specific.

Arguments recipients template-expansion

User-protocols **smtp (byte-stream)** - RFC821

**chaos-mail (chaos)** - Amber 5.5.

**dummy-mailer (local)** - Implements store-and-forward-mail in terms of mail-to-user. Takes the place of having the real mailer loaded.

Server-protocols **chaos-mail.**

| | | |
|---|---|---|
| **chaos-mail** | protocol | |
| **dummy-mailer** | protocol | |
| **smtp** | protocol | |

### 2.7.20 tape Service

| | | |
|---|---|---|
| **tape** | service | |
| **rtape** | protocol | |

### 2.7.21 tcp-gateway Service

| | | |
|---|---|---|
| **tcp-gateway** | service | |
| **tcp-gateway** | protocol | |

### 2.7.22 time Service

| | | |
|---|---|---|
| **time** | service | |
| | Returns | the current universal time. |
| | User-protocols | **local-time (local)** - interface to local idea of the time. |
| | | **time-msb (byte-stream)** - IEN142 |
| | | **time-simple-msb (byte-stream)** - IEN142 |
| | | **time-simple (chaos)** - Amber 5.8. |
| | Server-protocols | **time** |
| **local-time** | protocol | |
| **time-msb** | protocol | |
| **time-simple** | protocol | |
| **time-simple-msb** | protocol | |

### 2.7.23 uptime Service

| | | |
|---|---|---|
| **uptime** | service | |
| | Returns | the host's uptime in sixtieths of a second. |
| | User-protocols | **uptime-simple (datagram)** - **ans** with four bytes as for time. |
| | Server-protocols | **uptime-simple** |

**uptime-simple** protocol

### 2.7.24 who-am-i Service

**who-am-i** service

| | |
|---|---|
| Returns | three values: namespace name (a keyword), host name (or **:unknown**), and host who gave the answer. |
| User-protocols | **who-am-i (datagram) - ans** with **namespace\|host-name. *unknown*** for a host you do not know about. |
| Server-protocols | same. |

**who-am-i** protocol

# PART II.

# Remote Login

# 3. Overview of Remote Login Capability

The remote login facilities allow up to three ASCII terminals to be connected directly via the Symbolics computer's serial ports. Any number of terminals can be connected via the network. If a modem is connected to the machine, it is also possible to dial up the machine from an ASCII terminal or from another Symbolics computer. Video operations are supported only on ASCII terminals that support ANSI X3.64 display codes (Ann Arbor Ambassador, Digital Equipment VT100, and so forth).

Network servers are available for the remote login protocols TELNET, SUPDUP, TTYLINK, and 3600-LOGIN. TELNET and SUPDUP are standard protocols used on the Arpanet. TTYLINK is a raw byte-stream. 3600-LOGIN is used only in communication between two Symbolics computers.

The following programs can be run from terminals connected via a network, a serial port, or a modem:

- Lisp Listener
- Input editor
- Debugger (not the Window Debugger)
- Command processor

Zmacs, Zmail, and other programs that use the window system or the mouse cannot be used.

The remote login facility is useful for applications such as the following:

- Examining the status of a physically distant machine, such as a file server.
- Monitoring the status of a long computation from home.
- Simple data-entry or query-and-answer applications.

Note that the remote login feature cannot support several programmers on the same machine, because program-development tools, such as Zmacs, cannot be used remotely.

For further information on remote login: See the section "Using the Remote Login Facilities", page 69.

For information on the new functions dealing with remote login:

See the function **neti:ask-terminal-parameters**, page 69.
See the function **neti:set-terminal-parameters**, page 69.
See the function **neti:enable-serial-terminal**, page 69.
See the function **net:remote-login-on**, page 70.

# 4. Using the Remote Login Facilities

To use the 3600-LOGIN protocol, add the service attribute "LOGIN *medium* 3600-LOGIN" to the host object, where *medium* is the link-level network protocol (such as CHAOS or TCP) being used for communication between Symbolics computers.

The SUPDUP protocol is used in cases where ANSI X3.64 terminals are connected to Symbolics computers or where they are connected to foreign hosts that are using SUPDUP servers to connect to Symbolics computers. When the SUPDUP protocol is being used, terminal information is communicated automatically.

When using TTYLINK, TELNET, or the serial line, you should use the function **neti:ask-terminal-parameters** or the function **neti:set-terminal-parameters** to describe the terminal.

**neti:ask-terminal-parameters** *Function*

Asks you for information about the ASCII terminal currently associated with **terminal-io**. You are asked whether the terminal supports ANSI x3.64 escape sequences, whether it has a META key, and for its height and width in characters. Your answers are used to set or change the terminal's parameters. If you supply **nil** for height and width, the current settings are unchanged.

**neti:set-terminal-parameters** *x3.64 meta-key? width height* *Function*

Sets the parameters of the terminal associated with **terminal-io**. The argument *x3.64* specifies whether the terminal supports escape sequences meeting this ANSI standard; *meta-key?* says whether the terminal has a Meta key; *width* and *height* are the terminal's width and height in characters, respectively. If you supply **nil** for height and width, the current settings are unchanged.

To use a terminal connected via a serial line, invoke the function **neti:enable-serial-terminal**.

**neti:enable-serial-terminal** &key (*top-level* **'si:lisp-top-level1**) (*herald* **t**) (*x3.64* **nil**) (*width* **79**) (*height* **1073741824.**) (*unit* **1**) (*share-kill-history* **nil**) &allow-other-keys *Function*

The function **neti:enable-serial-terminal** allows an ASCII terminal to communicate with a Symbolics computer process through one of the machine's serial ports (specified by the *unit* argument).

The argument *x3.64* specifies whether the terminal supports escape sequences meeting this ANSI standard; *meta-key?* says whether the terminal has a Meta key; *width* and *height* are the terminal's width and height in

characters, respectively. If you supply **nil** for height and width, the current settings are unchanged.

The keyword argument **:top-level** specifies the process, and **:herald** specifies whether the herald is displayed on the terminal.

Sample use:

```
(neti:enable-serial-terminal :X3.64 T :HEIGHT 48.
 :WIDTH 80. :UNIT 3 :BAUD 9600.)
```

This creates a Lisp Listener process to communicate with the terminal. If you wish to have some other program communicating with the terminal, either invoke the program from the Lisp Listener, or use the **:top-level** keyword argument. The value of this keyword should be a function of one argument, which is the stream going to the terminal.

To kill the process, use **neti:disable-serial-terminal**; its single argument is the unit number.

If the terminal automatically echoes a newline when a character is printed in the rightmost column, then decrement the width by one.

If you are logging in from one Symbolics computer to another, the keyboard operation is identical except that these keys do not work:

- FUNCTION
- SELECT
- c-ABORT
- c-m-ABORT
- c-SUSPEND
- c-m-SUSPEND

If you are logging in from an ASCII keyboard, a translation scheme exists to allow you to refer to Symbolics computer keys that do not exist on an ASCII keyboard. From the logged-in ASCII keyboard, type c-_ H for online documentation.

If no user is logged into the Symbolics computer, there are no restrictions on logging into it from a remote terminal. If a user is logged in, remote login connections are rejected by default. The function **neti:remote-login-on** can be used to change this.

**net:remote-login-on** &optional (*mode* **t**) *Function*

The function **neti:remote-login-on** controls the acceptance or rejection of remote login requests to a Symbolics computer that has a user logged in at the main console. The *mode* argument specifies the treatment of remote login requests, as follows:

| | |
|---|---|
| **t** or unspecified | Allow remote login connections even when the main console is in use. |
| **nil** | Reject remote login requests. |

**:notify** Allow remote login requests but send the main-console user a notification.

Additional notes:

- The SUPDUP server works only if the terminal supports character insertion and deletion.
- There are no asynchronous characters. If your program starts looping, it must be aborted from the main console.
- Only one interactive process is allowed per remote terminal.

# PART III.

# Symbolics Dialnet

# 5. Introduction to Dialnet

Symbolics Dialnet is the component of the generic network system that supports the international dial network. The function of Dialnet is to provide a reliable transport medium over possibly unreliable common carrier facilities. The primary uses of this transport medium are mail transfer and remote login. Mail transfer is handled by the Symbolics mail reading and sending program (Zmail) and by the Symbolics store-and-forward mailer. Remote login is handled by the Terminal program.

# 6. Physical Connection to the Dial Network

The first step in the connection of your Symbolics computer to the dial network is to find a Symbolics machine at your site that has a modem. At least one machine at the site must have a modem; this is the machine that was designated the "site support system" in your initial Symbolics purchase. Other machines may also have modems, probably ordered from Symbolics as item COMM-MOD2.

The mechanics of the physical connection of your host to the dial network depend on the model of the processor:

- Symbolics 3600 processors contain a Vadic 3450 modem mounted inside the processor cabinet. Both a modular jack and a male EIA connector are brought out to the I/O bulkhead. The modular jack (labeled **MODEM TELCO**) accepts a standard modular plug from the data circuit provided by the telephone company. The male EIA connector (labeled EIA 4) should be connected (via a short cable, see below) to any one of the three female EIA connectors that provide access to the serial lines (labeled EIA 1, EIA 2, and EIA 3). EIA 1 corresponds to serial unit 1, EIA 2 to serial unit 2, and EIA 3 to serial unit 3.

  The cable between EIA 4 and EIA 1, EIA 2, or EIA 3 should pass the following signals on the pins given below:

  - Pin 2 (TXD ; Transmitted Data)
  - Pin 3 (RXD ; Received Data)
  - Pin 4 (RTS ; Request To Send)
  - Pin 5 (CTS ; Clear To Send)
  - Pin 6 (DSR ; Data Set Ready)
  - Pin 7 (SG ; Signal Ground)
  - Pin 8 (CXR ; Carrier Detect)
  - Pin 20 (DTR ; Data Terminal Ready)

  This cable should be terminated with one male and female connector. If you would prefer not to build such a cable yourself, it can be ordered from Symbolics.

  If your 3600 will be upgraded to support audio and phase-encoded video [via UPGR-SY70], then the gender of the serial ports will be male. The cable

should be constructed as above except that both connectors on the cable will be female. Again, should you prefer not to build such a cable yourself, it can be ordered from Symbolics.

Earlier versions of the FEP proms installed in 3600 processors do not support all the features of the 3600 serial lines. FEP version 22 or higher is required if Dialnet is to be used. The easiest way to determine what version of FEP proms are installed is the FEP Show Version command.

- Symbolics 3640 and 3670 processors have no internal modem, but instead expect an external Vadic 3451 or CDS-224 modem to be connected to one of the three serial ports.

  The modem should be connected to the data circuit provided by the telephone company via the modem's modular jack, and to one of the 3640 or 3670's serial lines. These serial lines are brought out to the I/O bulkhead and terminate in male EIA connectors. On the 3640 processor, these connectors are labeled SERIAL 1, SERIAL 2, and SERIAL 3. On the 3670 processor, these connectors are labeled EIA 1, EIA 2, and EIA 3.

  The cable used to connect the modem to the serial port should convey all the signals described above for 3600 cabling; only the gender of the serial port connector will differ. If you would prefer not to build this cable yourself, it can be ordered from Symbolics.

# 7. Dialnet Representation in the Namespace Database

The physical connection of your machine to the dial network must be recorded in the namespace database so that the generic network system can decide how connections over the dial network can best be established. The connection is represented by adding a **peripheral** attribute to the local view of the host object in the network database that corresponds to the host with the physical connection to the dial network.

The term "local view" is used here in contrast to the dial namespace view of the object. If you are editing a host that has *already* been identified as being in both the local and the dial namespace view namespaces, you will be asked to choose (via a small pop-up menu) between the local and the dial namespace view of the object before you begin to actually edit the object. Choose the local view; there are no servers for the dial namespace, so you would have no way to save your changes.

The Dialnet registry must also be informed that the Dialnet host corresponds to the host object. See the section "Dialnet Registries", page 81.

Finally, an Internet Domain Name should be specified for the local namespace object, to identify your hosts in the Internet Domain hierarchy. See the section "Dialnet and Internet Domain Names", page 85.

The **peripheral** attribute is used here to represent the modem that connects this host to the dial network. There are four relevant indicators for the peripheral attribute: **unit**, **model**, **phone-number**, and **autoanswer**.

**unit** Corresponds to the serial port number on the I/O bulkhead of the machine. The value should be a number between 1 and 3.

**model** Corresponds to the type of modem attached to this serial port. The value should be a one of the following: va212, va3450, va3451, or cds-224.

**phone-number** Corresponds to the telephone number of the telephone trunk to which this modem is attached. The phone-number associates this particular modem with a particular dial network address. (The address, in turn, associates this peripheral with a host in the dial namespace. This latter mapping is done via the Dialnet registry.)

**autoanswer** Corresponds to the ability of this modem to receive incoming calls from other sites. If you wish to be able to receive calls from other sites, then this indicator should have a value of **yes**. If it has a value other than **yes**, then incoming calls will not be answered and communication with other sites can only be initiated by the local site.

In general, if two sites wish to communicate over the dial network, at least one of the sites needs to enable **autoanswer**.

An example peripheral attribute might be:

```
peripheral modem unit 2 model va3450 phone-number 16175777348 autoanswer yes
```

# 8. Dialnet Registries

The shape of and possible connections on the dial network are represented in Dialnet *registries*. These files really contain namespace information in a form suitable for periodic distribution by separate administrative groups. The following registries exist:

*The public dialnet registry*

This registry is maintained and distributed by Symbolics. It contains information on publicly accessible Symbolics hosts and domains, on Telenet PADs (the GTE Telenet equivalent of the Arpanet TIP), and on dialing conventions for the international dial network. For example, information in this registry might be the addresses of Symbolics software support groups, all the GTE Telenet PAD access numbers, and enough information about the international phone system so as to most cheaply place calls to the Symbolics hosts described in the registry. This registry might also contain the domain of and address for the users' group, so that new customers could easily contact the group.

This file is sys: site; public-dialnet-registry.lisp.

*The users' group dialnet registry*

This registry is maintained and distributed by the Symbolics users' group, and contains whatever host and domain information the group's members see fit to enter and distribute.

For example, the users' group might distribute the addresses and domains of all members that wanted to share information via the dial network.

This file is sys: site; users-group-dialnet-registry.lisp.

*The private dialnet registry*

This registry is maintained at the local site. It contains local dialing conventions and any private host and domain information for the site.

A private dialnet registry might contain the names, addresses, and domains of the following:

- Other sites in the organization
- Other organizations that did not want to be published by the users' group
- Common carriers (for example, Tymnet)

- Subscriber data services (for example, Dow Jones Information Services)
- Gateways to other domains (for example, .ARPA)

This file is sys: site; private-dialnet-registry.lisp.

Information moves into the users' group registry when someone at the local site contacts the group, and the group distributes a registry. Information moves into the public registry when someone at the local site contacts someone at Symbolics, and Symbolics next distributes a registry.

Because of delays in the distribution of registries, the private registry may be useful as a repository for advance copies of public or users' group information. When two registries contain differing information about the same object, any conflict is resolved as follows:

1. The public registry is assumed to be least current.
2. The users' group registry is assumed to be more current than the public registry.
3. The private registry is assumed to be more current than either public or users' group.

## 8.1 Contents of a Dialnet Registry

A Dialnet registry actually contains Lisp forms. The forms are lists of alternating keywords and values. The first keyword in a form (the car of the list) indicates the type of information being conveyed and is one of the following:

**:subnet** Specifies the cost of and dialing information for a subnet of the dial network. The keywords and values for this form are described elsewhere. See the section "Dial Network Addressing", page 95.

```
(:subnet "1xxxyyyyyyy>1800zzzzzzz" :dial "1800zzzzzzz" :cost "1")
```

**:host** Specifies a host on the dial network and its address. The name specified by the **:host** keyword is the name of the host in the dial namespace. To avoid duplicate names in this namespace, we use the site name as a prefix. (Remember that site names are guaranteed to be unique because they are centrally administered by Symbolics Software Support). As an example, consider a site named Trilogy, one of whose hosts with a dial network address is named Cerebrus. The unique dial namespace name of this host is then constructed to be trilogy-cerebrus.

```
(:host "trilogy-cerebrus" :address "14151515151")
```

The address for the host is a string representing the address of the host on the international dial network. In our examples, drawn typically from hosts in the United States, the country code is 1, the area code a three-digit number whose second digit is always 0 or 1, and the rest of the address is the familiar seven-digit number one most often dials. The address is the concatenation of these three fields.

If a host is present in a local namespace other than the dial namespace, its local name should be noted with the :local-name keyword. For example, the following specifies a host whose name (in the dial namespace) is USMC-Gomer, whose address on the dial network (that is, the network named "dial" in the dial namespace) is 14155551212, and whose name in the local namespace (here, the namespace named USMC) is Gomer.

```
(:host "USMC-Gomer" :address "14155551212"
 :local-name "USMC|Gomer")
```

Each host in the dialnet registry is assumed to provide the following services:

- STORE-AND-FORWARD-MAIL (over the DIAL medium, using the SMTP protocol)
- MAIL-TO-USER (over the DIAL medium, using the SMTP protocol)
- MAIL-PROBE (over the DIAL medium, using the MAIL-PROBE protocol)

The format of **:host** entries will probably change in the future to allow more flexible representation of dialnet host services.

**:domain** Specifies an Internet mail domain and the name of the associated gateway host. Internet mail domains are used by ZMail and the store-and-forward mailer to direct mail around the dial network, in the absence of other namespace information. For example, if the following forms were present in the users' group registry

```
(:host "CSNY-Young" :address "12121234567")
(:domain "CSNY.DialNet.Symbolics.COM"
 :host "CSNY-Young")
```

then mail to the following addresses would be routed via DialNet to CSNY-Young for further distribution:

```
Neil@CSNY-Young.CSNY
Neil@CSNY-Young
Neil@anyhost.CSNY
```

**:telenet-pad** Specifies the name and address of a GTE Telenet PAD. These gateways to the Telenet network can be an economical way of routing traffic across the dial network and are also useful in their own right as access to higher level GTE Telenet services such as TeleMail. The Symbolics Dialnet implementation will use Telenet automatically when it determines that by doing so it can make a cheaper connection.

```
(:telenet-pad "boston-telenet-pad"
 :address "16172920662")
```

## 8.2 Loading a Dialnet Registry

All the information in the three Dialnet registries (public, users' group, and private, if present) is loaded into the local host with the **dial:load-dialnet-registry** function.

```
(dial:load-dialnet-registry)
```

This function can be called by the user (to load the names and addresses of all Telenet PADs, for instance); it is also called by some system programs (for example, the store-and-forward mailer).

# 9. Dialnet and Internet Domain Names

The dialnet registries allow the user to associate Internet Domain Names with individual hosts; in particular, they allow the user to specify that a given host will serve as a mail gateway for a given Internet Domain.

Internet Domain Names are part of a tree-structured naming scheme used by the Internet community for distributed administration of a very large namespace. That namespace is the set of all hosts connected with the Internet. This set of hosts is too large for any particular organization to handle, so a naming hierarchy is set up, with naming authority for each node parceled out to local adminstrators.

At the top of the naming tree is an implicit root node. Below that node are several very general classifications:

| | |
|---|---|
| **GOV** | United States government |
| **EDU** | educational institutions |
| **COM** | commercial institutions |
| **MIL** | military |
| **ARPA** | Arpanet community (temporary, pending reclassification of member hosts) |

and other domains representing countries outside of the United States; this last set would be two-letter codes drawn from the ISO standard for codes for the representation of names of countries.

Symbolics networks are represented under the second-level domain name Symbolics.COM, and the Symbolics dial namespace is a third-level domain named DialNet.Symbolics.COM. The namespaces of individual Symbolics customer sites are usually represented as fourth-level domains, subdomains of the Symbolics Dial Network. The dial namespace host named CSNY-Young (to borrow from a previous example in this section) would be in the CSNY.DialNet.Symbolics.COM domain.

The customer specifies the Internet Domain Name to be associated with the namespace in which local hosts are registered by editing the **Internet Domain Name** attribute of the namespace object that represents the local namespace itself. This Internet Domain Name will be associated with all hosts that are named within that namespace.

Not all sites will belong in subdomains of the DialNet.Symbolics.COM domain (although most will). Sites that are already resident in another Internet domain (such as .ARPA) will want to maintain their residency in that domain. This is why the user must explicitly edit the namespace object to place the local namespace into a subdomain of the DialNet.Symbolics.COM domain, should that be desired.

Internet Domain Names are used by ZMail and by the store-and-forward mailer to help route mail through the network, in the absence of other namespace information. The key point in this routing is that a mail server at a given domain in the hierarchy is assumed to know the addresses of mail servers for all the subdomains within that domain. Thus, if a mailer in the source domain doesn't know how to directly reach the destination domain, it can pass the mail up to a mail server at a higher-level domain, which will then apply the same routing algorithm. The "best" route between domains is considered to be the route with the least number of levels traversed.

For further information on Internet Domain Names, see the following document:

```
RFC 882: Domain Names - Concepts and Facilities
Paul Mockapetris
Network Working Group, USC ISI
```

which is available from:

```
Network Information Center
SRI International
Menlo Park, CA  94025
```

# 10. Using the Terminal Program with the Dial Network

Once you have set up the hardware and namespace information that describes how your host is connected to the dial network, you can use that host to dial up other hosts. This is an excellent test of the hardware and software configuation even if you don't usually dial up other hosts. And, of course, it can be very useful in its own right, providing access to hosts accessible only via dial-up lines.

Type SELECT T to get to the Terminal program, then type a host name at the Connect to Host: prompt. Host names are of the form dial|dial|16175777348. To break that down a bit, that's the host at address 16175777348 on the network named dial|dial, which in turn is the network named dial in the dial namespace. If you need to make the same call frequently, you can add hosts to your own local namespace (not the dial namespace) with addresses on the dial|dial network. In addition to an address attribute, you will probably want to give such a host a service attribute of:

```
login dial-raw tty-login
```

Telenet PADs have names in the dial namespace, names like dial|boston-telenet-pad. Telenet PADs are listed in the public dialnet registry, so to dial up a PAD you will have to load the dialnet registry first. See the section "Loading a Dialnet Registry", page 84.

# 11. An Example Dialnet Installation

This section details an example installation at site NYC. The goal of this installation is to bring up this site on the dial network, both to exchange electronic mail with other sites and to use the Terminal program to access other hosts over dialup lines. There are five machines at site NYC, named Bronx, Queens, Manhattan, Brooklyn, and Staten-Island. Bronx (a 3600) was the first machine installed at the site, so it bears the title of "site support system" and has an inboard Vadic modem. In this example it also has an attached Kanji tablet for Japanese language work, and supports an LGP printer. When Manhattan (a 3670 with a large disk) later arrived, it was made the SYS host and the namespace server, and Bronx was given over to being a file server for user files, in addition to its continued use as a Japanese workstation and spooler for the LGP printer.

- **Find the modem.** In this example, Bronx has an internally-mounted Vadic VA3450 modem by virtue of being the first system shipped to the site (the so-called "site support system").

- **Connect the modem to a serial port.** The serial port, of course, should be otherwise free. In this example, serial ports 1 and 2 are already in use (for example, they could be supporting a tablet and a printer), so you will be attaching the modem to serial unit 3. A short cable should be used to connect EIA 3 and EIA 4, where EIA 3 is the connector for serial port 3 and EIA 4 is the connector for the inboard modem. For details on how to make this cable (if such a cable wasn't shipped to you by Symbolics) or on how cabling should be done for a 3640 or 3670: See the section "Physical Connection to the Dial Network", page 77.

- **Connect the modem to the phone company's data circuit.** This requires a telephone line with modular connectors on both ends, of sufficient length to reach from the telephone jack to the **MODEM TELCO** jack on the 3600 I/O bulkhead. (3640 and 3670 installations use the same type of cable but the connection is from telephone jack to the **LINE** jack in the CDS modem). In this example, the data circuit is on a private branch exchange (PBX) that supports direct inward dialing, allows other extensions to be called by dialing their 4-digit trunk numbers, but requires a 9 be dialed first when making calls outside the PBX extensions. The example phone number for Bronx will be area code 212, phone number 765-4321 (**SOho5-4321**, if you prefer).

- **Register the modem in the namespace database.** Use the namespace editor to add a **peripheral** attribute to the local view of the host to which the modem is attached. In this example, you are attaching the modem to serial unit 3, the modem is a Vadic VA3450, the phone number is 1-212-765-4321 (1 (US) country code, 212 area code, 765 exchange, 4321 trunk). Furthermore,

this example supposes you want other sites to dial your site and assume some of the communications costs themselves, so you want to enable the autoanswer feature. Thus the peripheral attribute you add and save looks like:

```
peripheral modem unit 3 model va3450 phone-number 12127654321 autoanswer yes
```

Past the **modem** indicator, the order of indicator/value pairs is irrelevant; you could just as well have said:

```
peripheral modem phone-number 12127654321 autoanswer yes unit 3 model va3450
```

For further information on registering the modem in the namespace database: See the section "Dialnet Representation in the Namespace Database", page 79.

- **Create a private dialnet registry.** You need to enter into the private dialnet registry information concerning
  - local dialing conventions,
  - hosts connected to the dial network and their addresses, and
  - locally supported Internet mail domains.

This is done by creating a private dialnet registry. You should edit the file sys:site;private-dialnet-registry.lisp and enter the following:

```
;;; -*- Mode: LISP; Package: DIAL; Base: 10 -*-

;;; one PBX number to another: just dial the extension.
;;; local external phone number: dial 9, then the number.
;;; different area code: dial 9, then the number.
;;; WATS number: dial 9, then the number.
;;;      Note that WATS is cheaper than long-distance.
(:subnet "1212765ssss>1212765dddd" :dial "dddd" :cost "0")
(:subnet "1212765ssss>1212xxxdddd" :dial "9xxxdddd" :cost "1")
(:subnet "1212765ssss>1aaaxxxdddd" :dial "9aaaxxxdddd" :cost "2")
(:subnet "1212765ssss>1800xxxdddd" :dial "91800xxxdddd" :cost "1")

;;; modem is on DIAL|NYC-Bronx, phone (212) 765-4321,
;;;  and the local name of the host is Bronx.
(:host "NYC-Bronx" :address "12127654321" :local-name "NYC|Bronx")

;;; DIAL|NYC-Bronx will run a store-and-forward mailer,
;;; servicing the domain named NYC.DialNet.Symbolics.COM.
(:domain "NYC.DialNet.Symbolics.COM" :host "NYC-Bronx")
```

For more information on dialnet registries: See the section "Dialnet Registries", page 81. For more information on Internet domains: See the section "Dialnet and Internet Domain Names", page 85. For more information on the store-

and-forward mailer: See the section "Overview of the Mailer" in *Communicating with Other Users*.

- **Give the local namespace an Internet Domain Name.** Both ZMail and the store-and-forward mailer use the Internet Domain Name to route mail in some circumstances. Unless the namespace that names the site in which your hosts are registered already has an associated Internet Domain Name, you should use the namespace editor to add this attribute. In this example it is assumed that your site has no previous connection with the Internet, so you would edit the namespace object for the NYC namespace itself, and add an **Internet Domain Name** property of NYC.DialNet.Symbolics.COM.

- **Load the dialnet registry.** Type the following form:

```
(dial:load-dialnet-registry)
```

- **Test the dial network connection with the Terminal program.** This step is optional, of course, since you may not have a dial-accessible host handy. This example supposes there exists a timesharing host named NYC-Tammany, with a single dialup line at (212) 666-1040. You select the terminal program with SELECT T and enter the address of NYC-Tammany, in this case dial|dial|12126661040. The number is dialed, the connection is made, the screen clears and any subsequent communication is taking place with NYC-Tammany. After the session with the foreign host is complete, the connection can be broken by typing NETWORK L. For more information on using the Terminal program to access hosts via dialup lines: See the section "Using the Terminal Program with the Dial Network", page 87.

# 12. dial Network Medium

The dial network transport mechanism is interfaced to the Symbolics generic network system and can be used via the **dial** medium. This medium is a reliable byte stream built on the bare serial line connection between two modems. It provides the error detection and retransmission functions associated with most other networks, to protect the communication against line noise and against the loss of chatacters due to slow system response.

Any sufficiently generic network protocol can operate using the **dial** medium. Of course, the low transfer rates provided by modems make most interactive uses impractical. The supplied Symbolics software uses the **dial** medium only for transmitting electronic mail and for limited (that is, text-only) remote login.

# 13. Dial Network Addressing

The international dial network is modeled by a single namespace object: the **dial** network in the **dial** namespace. Addresses on this network are telephone numbers. Of course, area codes and other dialing prefixes make things more complicated.

Addresses for the dial network are complete telephone numbers, including country and area codes. For North American customers, the country code is 1, so a fully specified number looks like a common long distance sequence. Trunk 7348 in the 577 exchange of the 617 area code would be fully specified as 16175777348.

It is not generally appropriate to just dial a fully specified address; numbers within the same area code do not require the area code, and often require a 1 prefix if it is a toll call. The subnet attributes of the dial network encode the neccessary dialing prefixes. Each subnet represents a telephone company connection between two exchanges.

Since there are some rules stating which dialing prefix to use, it is not necessary to specify every possible binary combination of world-wide phone exchanges and their associated prefixes. Instead, Dialnet provides a simple pattern matcher that can be used to express both specific and general dialing rules. The name of each subnet on the dial network gives the input pattern to the pattern-matching system; these patterns are matched against the combined source and destination addresses for the connection, that is, against the local and foreign telephone numbers.

The pattern consists of two sequences of digits and letters. The digits represent the fixed parts of the pattern and the letters represent the variable parts. The two sequences are separated by a > character, indicating that the left-hand part of the pattern is the calling party and the right-hand part of the pattern is the called party. Contiguous occurrences of the same letter make up the same variable. Variable assignment takes place from left to right. If a letter is seen that has no assignment, the variable sub-sequence is tentatively assigned a value of the corresponding sub-sequence of the pattern to be matched. If the variable has an assignment (binding), or if there is a constant digit, it must match the corresponding part of the pattern to be matched.

A specific example will make this more clear. Suppose we are calling from 16175777348 to 14155200142. Given the subnet pattern 1xxxyyyyyyy>1zzzwwwwwww, we want to match it against 16175771212>14155200142. 1 is a fixed constant and matches. x has no binding so it is tentatively assigned 617. Likewise y is assigned 5777348, z 415, and w 5200142. The match is successful and the result is these four bindings.

Now suppose instead the subnet pattern was 1xxxyyyyyyy>1xxxzzzzzzz. The x assignment is the same, 617. Similarly the y assignment. On the second occurrence of x, however, it already has a binding, so this must be matched against the input. 617 does not match 415, so the whole subnet match fails.

The subnet that best represents a particular phone call is simply the one with the fewest variable bindings. So, if we were making the call 16175777348>16175777344, the pattern 1xxxyyyyyyy>1xxxzzzzzzz would have only three bindings, and so would be better than 1xxxyyyyyyy>1zzzwwwwwww, which has four.

The map between abstract subnet patterns and actual dialing sequences is maintained by the **subnet** attributes of the namespace object representing the international dial network. (This network is named **dial|dial**.) Each subnet pattern has associated pairs of indicators and values that encode the actual dialing sequence and the relative expense of the phone call.

The **dial** indicator is a string of numbers and letters that represents the actual dialing pattern. All of the variables in this attribute must have been assigned values as a result of the subnet matching process. The **cost** attribute is a small number (typically between 1 and 10) indicating the relative expense of the call. **cost** attributes are used for hosts with more than one address on the dial network (that is, hosts with more than one autoanswer modem) to determine the number to call, and to weigh use of a direct call against routing through a public carrier network.

Here is an example of typical subnet attributes for the **dial|dial** network:

```
subnet 1xxxyyyyyyy>1xxxzzzzzzz  dial zzzzzzz      cost 0
subnet 1xxxyyyyyyy>1zzzwwwwwww  dial 1zzzwwwwwww  cost 5
subnet 1212xxxxxxx>1yyyzzzzzzz  dial yyyzzzzzzz   cost 5
subnet 1617864xxxx>1617774yyyy  dial 1774yyyy     cost 3
subnet 1xxxyyyyyyy>1800zzzzzzz  dial 1800zzzzzzz  cost 1
```

These mean, respectively:

1. When dialing a call within the same area code, just dial the number.

2. When dialing a number outside the local area code, dial a 1, then the area code and number.

3. When dialing from the 212 area code, long-distance calls do not require a 1 prefix.

4. Within the 617 area code (Massachusetts), you need to dial a 1 to get from Cambridge (864) to East Boston (774).

5. The cost of a wide-area telephone service (WATS) call is less than a normal long distance call. Note that the cost of WATS is still declared higher than a local call; this is to avoid making a WATS call when a local call would do, leaving the WATS trunks available for those who need them.

# 14. Reducing Call Cost with Public Carrier Networks

Dialnet can make use of public carrier networks that provide terminal multiplexers. The cost of using this service is often considerably cheaper than a direct long-distance phone call. GTE Telenet, for example, provides local dial-ups attached to terminal concentrators, called PADs. These PADs connect through their X.25 network to other PADs and to multiplexors at other sites. Connection of a Symbolics computer to such a multiplexor is straightforward.

The public Dialnet registry shipped by Symbolics contains the dial network addresses of many Telenet PADs, as well as the Telenet addresses of some hosts.

So far as most of Dialnet is concerned, there is just a serial line connecting the two hosts. The intervening X.25 network is invisible. The part of Dialnet that knows how to make phone calls also knows how to make a phone call to the Telenet PAD and to negotiate with the PAD for a connection to another host on Telenet. Routing through Telenet occurs automatically if such a route would be cheaper than a direct dial network call to the same host.

# PART IV.

# Programmer's Reference on Networks

# 15. Chaosnet

## 15.1 Introduction to Chaosnet

Chaosnet is a *local network*, that is, a system for communication among a group of computers located within one or two kilometers of each other. The name *Chaosnet* refers to the lack of any centralized control element in the network.

Chaosnet was originally developed in 1975 by the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology as the internal communications medium of early Lisp Machine systems. Chaosnets also exist at several other universities and research laboratories.

In a network of Symbolics computers, each user is assigned a "personal" computer consisting of a processor, a suitable amount of memory, and a swapping disk. Files are stored in a central file system accessed through Chaosnet. This shared file system retains the traditional advantages of a timesharing system, namely, interuser communication, shared programs, and centralized backup and maintenance. Chaosnet is also used to access other shared resources; these include printers, tape drives, and one-of-a-kind specialized processors and I/O devices.

The design of Chaosnet was greatly simplified by ignoring problems irrelevant to local networks. Chaosnet contains no special provisions for things such as low-speed links, noisy (very high error-rate) links, multiple paths, and long-distance links with significant transit time. This means that Chaosnet is not particularly suitable for use across the continent or in satellite applications. Chaosnet also makes no attempt to provide unnecessary features (for local-area networks) such as multiple levels of service or secure communication (other than by end-to-end encryption).

Chaosnet consists of two parts—the hardware and the software—which, while logically separable, were designed for each other. The hardware provides a carrier-sense multiple-access structure such as usually found in Ethernet interfacing hardware. Network nodes contend for access to a cable, or *ether*, over which they may transmit packets addressed to other network nodes. The software defines higher-level protocols in terms of packets. These protocols can be (and are) used with media other than the Chaosnet cable, and with multiple interconnected cables.

## 15.2 Chaosnet Hardware Protocol

### 15.2.1 Chaosnet Ether

The transmission medium of Chaosnet is called the *ether*. Physically, it is a coaxial cable, of the semi-rigid 1/2-inch low-loss type used for cable TV, with 75-ohm termination at both ends. At each network node a *cable transceiver* is attached to the cable. A 10-meter flat cable connects the transceiver to an *interface* which is attached to a computer's I/O bus.

A network node consists of this transceiver and interface and a computer that executes the Network Control Program (NCP), which manages and controls Chaosnet, in addition to application software.

One network node at a time can seize the ether and transmit a packet, which arrives at all other nodes; each node decides in hardware whether to ignore the packet or to receive it.

A single ether must be a linear cable; it cannot contain branches nor stubs, and the ends cannot be joined in a circle. The maximum length of an ether cable is about 1 kilometer. The maximum number of nodes on a single ether is a few dozen.

The protocol provides for multiple ethers, joined together by nodes called *bridges*, which relay packets from one ether to another. A bridge is a computer with two or more network interfaces attached to it. A bridge node usually performs other tasks as well, such as interfacing terminals. Bridges attach other network media as well as ethers; some computers connect to the network through their manufacturer's high speed computer-to-computer interface to a nearby bridge, rather than being interfaced directly to an ether.

### 15.2.2 Chaosnet Packets

The basic unit of transmission is called a *packet*. This is a sequence of up to 4032 data bits, plus 48 bits of *header* information used by the hardware. Packets' bits are normally grouped into 16-bit words. The division of a transmitted bit stream into packets provides a conveniently sized unit for resource allocation and error control. The job of the hardware is to deliver a packet from one node to another.

The packet's hardware header consists of three 16-bit words, called *destination*, *source*, and *check*:

| | |
|---|---|
| Source | Identifies the node that transmitted this packet onto this ether. This is not necessarily the original source of the message, since it may have originated on a different ether. |
| Destination | Identifies the node intended to receive this packet from this ether. This is not necessarily the final destination of the message. It may be a bridge that should relay the packet to another ether; from there, it will eventually reach its final destination. |
| Check | A cyclic-redundancy checksum, generated and checked by |

hardware, which detects errors in transmission through the ether, entirely spurious packets created by noise on the cable, and memory errors in the transmitting and receiving packet buffers.

Software protocols define the meanings of the data bits in a packet, manage the hardware, compensate for imperfections of the hardware, and provide more useful services than simple transmission of packets from one computer to another. See the section "Packet Contents: Chaosnet Software Protocol", page 112.

### 15.2.3 Chaosnet Transceiver

All nodes are connected to the ether through a transceiver, which is a small box mounted directly on the cable via a UHF connector and a T-joint. All nodes use identical transceivers (the interface varies depending on what computer it is designed to interface to). The transceiver contains the analog portion of the interface logic, provides ground isolation between the ether cable and the computer, and contains some protective circuitry designed to prevent a malfunctioning program or interface from continuously jamming the ether.

The transceiver receives a differential digital signal from the computer interface and impresses it onto the cable as a level of about 8 volts for a 1, or 0 volts (open circuit) for a 0, through a very fast VMOS power FET. When the cable is idle it is held at 0 volts by the terminations. This simple unipolar scheme is adequate for medium cable lengths and transmission speeds. The transceiver monitors the cable by comparing it against a reference voltage, and returns a differential signal to the interface. In addition, it detects interference (another transceiver transmitting at the same time as this one) and informs the interface.

The transceiver includes indicators (light-emitting diodes) for power OK, transmitted data, received data, and interface attempting to jam the ether. A test button simulates an input of continuous 1s from the interface, which should light all the lights (dimly) if the transceiver is working. These indicators and the test button are useful for rapidly tracking down network problems. The transceiver requires its own power supply mounted nearby; one power supply can service three transceivers if they are all adjacent. High-voltage isolation between the cable and the computer is provided by optical isolators within the transceiver.

### 15.2.4 Chaosnet Interface

The interface is typically a wire-wrap board containing about 120 TTL logic chips, which plugs into the I/O bus of a computer and connects it to the ether (through a transceiver.) The interface implements the hardware protocols, buffers incoming and outgoing packets, generates and checks checksums, and interrupts the host computer when a packet is to be read out of the receive packet buffer or stored into the transmit packet buffer. These packet buffers shield the host computer from the high speed of data transmission on the cable. Instead of having to produce bits at a

high rate, the host can produce them at a lower rate, collect them into a packet, and then tell the interface to transmit the packet in a single burst of high-speed transmission. Programming information for the interface is supplied elsewhere. See the section "Chaosnet Hardware Programming Information", page 137.

### 15.2.5 Details of Chaosnet Hardware Protocols

The purpose of these protocols is to deliver packets intact from one node to another node on the same ether, with fairly high probability of success, and to guarantee to give an error indication or lose the packet entirely if it is not delivered intact. An additional purpose is to provide high performance and not to bog down when subjected to a heavy load.

Bits are represented on the ether using a technique which is called Upright Biphase NRZI. Each bit cell, which is approximately 250 nanoseconds long, begins with a transition in state, from high to low or from low to high. This transition marks the beginning of a bit cell and provides self-clocking. 3/4 of the way through the bit cell, the state of the cable is sampled; high represents a 1 and low represents a 0. If the bit being represented is the same as the previous bit, there will be one transition at the beginning of the bit cell and a second in the middle of the bit cell. If the bit being represented is the opposite of the previous bit, there will be no transition in the middle of the bit cell since the clock transition will have set the cable to the desired state. The AC frequency of the signal on the cable varies between 1/2 the bit rate and the full bit rate. The information bit-rate is 4 million bits per second.

The self-clocking feature allows for slight variations in transmission and cable propagation speed. However, since the 3/4 of a bit cell delay is a fixed delay, only modest variations in speed can be tolerated. A crystal clock is used as the source of the transmit timing in the interface.

Since there is always at least one state-transition per bit cell, the states where the ether remains high or low for an appreciable time are available for nondata uses. If the ether remains low for more than about two bit cells, it is considered to be not-busy. This condition marks the end of a packet and allows someone else to transmit. Note that if no transceivers are active, the terminations will hold the ether low.

If the ether remains high for about two bit cells, this is an "abort signal". Abort signals are used for two purposes. If the transceiver detects a collision (two nodes trying to transmit at the same time), each transmitting interface ceases to transmit and sends an abort signal (four bit cells long), which tells all receivers to ignore the aborted packet and ensures that the other transmitter also aborts. Thus when a collision occurs, the ether is cleared as soon as possible to help prevent long tie-ups under conditions of heavy load. The other use for the abort signal is in hardware flow-control. When a receiving interface determines that an incoming packet is addressed to it, but its receive buffer already contains a packet, it sends an abort signal which causes the transmitter to stop. This serves the dual purpose of

immediately informing the transmitter that its message did not get through, and preventing the ether from being tied up while a long packet is transmitted which the receiver cannot receive.

Packets are transmitted over the ether in reverse bit-order, for hardware convenience. The three header words, which to the software appear to be at the end of the packet, are transmitted first, in the order check, source, destination. The data words, in reverse order, follow. Words are transmitted least-significant bit first. Of course, the software need not be aware of this reversal; packets arrive at the destination in the same form as they were created by the source. At the end of the packet, an extra zero bit is appended to bring the ether to the low state so that an extra spurious clock-transition will not be generated when it goes idle. This bit is stripped off by the interface and is never seen by software.

The check word is used for error detection, as described above. The source word is made available to the software, which ignores it in most cases, and also serves to synchronize the clocks in the collision-avoidance mechanism. The destination word is compared by each receiver against its own address. If they match, or if the destination is zero, or if the software selects the "match any destination" mode, the packet is placed in the receive packet buffer and the host computer is interrupted. The zero destination feature is used for "broadcast" messages. Note that a receiver whose packet buffer is full will only generate an abort signal if the packet was specifically addressed to it.

### 15.2.6 Ether Contention

Chaosnet has no centralized control element; when a network node has a message to transmit, its interface seizes the ether and transmits a packet. The time when it seizes the ether is determined only by state inside that particular interface and by the local state of the cable at the point where that interface's transceiver is attached.

If two interfaces should decide to seize the ether and transmit at the same time, their transmissions will interfere and no useful information will be transmitted. This is called a *collision*. Collisions are the principal limitation on the bandwidth of a heavily loaded ether-type network, and should be avoided.

Chaosnet uses a novel collision-avoidance technique. First, an interface will never initiate transmission unless the ether is seen to be not busy, that is, it has been in the low state for some time. This ensures that collisions can occur only near the beginning of a packet. Once transmission of a packet has gotten well started, the ether is effectively "seized" (all interfaces realize that it is busy) and transmission will continue successfully through to the end of the packet. The amount of ether transmission time wasted by a collided packet is therefore limited to the round-trip cable propagation delay. This technique is called *carrier sense*.

Secondly, the hardware uses a time-division technique to attempt to prevent two

interfaces from initiating transmission at the same time. This technique should prevent essentially all collisions while imposing only a modest delay in the initiation of transmission. It is designed so that it works better as the load on the ether increases; the wasted time between packets and the relative rate of collisions both decrease.

The basic idea is that each interface is assigned a time-slot, or *turn*, according to its address. It may only initiate transmission during its turn. The turns are spaced far enough apart that if one interface initiates transmission, every other interface will perceive that the ether is busy by the time its own turn arrives, and will not initiate an interfering transmission. Each interface contains a time-slot counter which counts while the ether is not busy, keeping track of whose turn it is. Each packet synchronizes the counters in all of the interfaces by setting them from the source address of that packet; at the time the packet was transmitted, it must have been the turn of the interface that transmitted it.

Another way to think of this is to make an analogy with ring networks. One can imagine a *virtual token* which passes down the cable until it gets to the end, then jumps to the beginning of the cable and repeats. An interface may only initiate transmission at the instant the token passes by it. When an interface transmits, the token stops moving and remains at that interface until the end of the packet, whereupon it continues down the cable, passing every other interface, giving them each a chance to transmit before letting the first interface transmit a second packet.

The token is not represented by any physical transmission on the cable. That would constitute a form of centralized control, and would lead to reliability problems if the token was lost or duplicated. Instead, every interface contains a time-slot counter which keeps track of where the token is thought to be. Every time a packet is transmitted these counters are brought up-to-date. The token cannot be lost because a counter by its nature eventually returns to all previous states. It does not matter if the token is duplicated (that is, the counters lose synchronization) occasionally, since this will only cause collisions, which we know how to detect and deal with, and since the first successful transmission will resynchronize all counters. The basic mechanism of the ether network with contention and collisions is known to work, and the collision-avoidance scheme is an added-on optimization which improves performance without changing the basic mechanism.

There is a finite propagation delay time between interfaces, and this time is not small compared with the bit-rate of Chaosnet, nor when compared with the desirable length of a time slot. This time consists of the delay in the cable, about 5 nanoseconds per meter, and the delay through the two transceivers, about 220 nanoseconds. This propagation delay means that the time-slot counters in two different interfaces cannot be exactly synchronized, and that when interface A initiates transmission interface B will not instantaneously see that the ether is busy. Special relativity tells us that in fact the concept "exactly synchronized" is meaningless. Since the two time-slot counters are not in the same place, the only way we can compare them is to send a message from one to the other, through the

ether, containing the reading of the counter. But this message takes nonzero time to get there, so the counter-reading it contains is wrong by the time it is compared against the other counter! We in fact do send messages containing counter readings; the source address in a packet equals the reading of the time-slot counter in the interface that sent it--at the time it was sent. Since the network nodes are not in relative motion, we can measure the distance between them and use that information to improve their synchronization.

What we are trying to do is to prevent collisions. This means that if interface A starts transmitting a packet in its turn, then by the time interface B thinks that its own turn has arrived, it must perceive the ether as busy. We will assign addresses (and hence time slots) and set the length of a time slot in such a way that this will happen. Suppose the maximum delay through the ether between A and B is *t*. This would be the delay for one of them sending a packet to the other; the delay between A's receipt of a third party's packet and B's receipt of that packet is less, especially if the third party is between A and B on the cable. Then the maximum perceived difference between a clock at A and a clock at B is *2t*; if a message is sent from B to A synchronizing the clocks, and then a message is sent from A to B containing A's clock reading, at B this clock reading will be slow by *2t*.

When a packet transmitted by A arrives at B, B's clock may read as much as *2t* earlier or later than A's turn, depending on the transmission direction of the last synchronizing message. In order to guarantee that B's turn has not yet happened, the time between any of A's turns and any of B's turns must always be at least *2t*, twice the maximum propagation delay through the ether between A and B. This is the important idea! We cause this to be true by assigning addresses starting at one end of the cable; each node's address is the previous node's address plus twice the propagation delay between them, divided by the length of a turn. It is easy to see that if this is done for all adjacent pairs, the condition will automatically be true for nonadjacent pairs as well. When we get to the end of the cable, we must assign a number of empty slots equal to twice the propagation delay of the full length of cable, to provide the necessary separation between the turns of the two nodes at the ends of the cable.

The virtual token travels through the network at a substantially slower speed than a real signal such as a packet; in the fastest case, when nodes are very far apart, it travels at just half the speed of a real signal. Since a Chaosnet ether has the geometry of a line, as compared to the ring net's circle, the virtual token is also slowed down by the need to return from the end of the cable to the beginning. This slower speed of the token is the price one pays for the increased robustness of Chaosnet as compared with a ring network. In a real system, we slow the token down even more to provide a margin of safety. The speed of the network is not significantly affected by the slow token, since the interval between packet transmissions by a single node is much longer than the round-trip time of the token. Indeed, if the network is being used primarily for file transfer, and hence the packets are large, the transmission time alone for a typical packet is several times

the round-trip time of the token. A typical value for the token's round-trip time is 64 microseconds.

In spite of all this, sometimes collisions will occur anyway. If the cable has been idle for a long time, the various clocks will have lost synchronization. If a source address is corrupted by a transmission error, any interface that sees that source address will set its clock to an incorrect value. Sometimes a packet will collide with random noise rather than another legitimate packet. In addition, the transmitter does not distinguish receiver-busy aborts from real collisions.

When a collision does occur, we recover from it (in software) by retransmitting the packet again a couple of times, hoping that we will be lucky enough not to have another collision, or that the receiver will soon clear its packet buffer. This retransmission is done by the software, not the hardware, since the hardware destroys the packet in its packet buffer in the process of transmitting it. But if collisions continue to occur, we give up and let somebody else have the ether. The packet is lost. A higher level of protocol will soon realize that it has been lost and retransmit it. We assume that there is enough randomness in the higher-level software that the two nodes which originally collided will not collide again on the retransmission by deciding to retransmit at precisely the same instant.

## 15.3 Chaosnet Software Protocol -- Overview

The purpose of the basic software protocol of Chaosnet is to allow high-speed communication among processes on different machines, with no undetected transmission errors. The speed for file transfers in real-life circumstances was to be comparable with an inexpensive magnetic tape drive (30000 characters per second, or about 10 times the speed of the Arpanet). We actually get about double this in some favorable cases. To achieve this speed it was important to design out bottlenecks such as are found in the Arpanet, for instance the control-link which is shared between multiple connections and the need to acknowledge each message before the next message can be sent. The protocol must be simple, for the sake of reliability and to allow its use by modest computer systems. A full Chaosnet Network Control Program is just about half the size of an Arpanet NCP on the same machine, and the protocol allows low-performance implementations to omit some features. A minimal implementation exists for a single-chip microcomputer.

### 15.3.1 Connections: Chaosnet Software Protocol

The principal service provided by Chaosnet is a *connection* between two user processes. This is a full-duplex reliable packet-transmission channel. The network undertakes never to garble, lose, duplicate, or resequence the packets; in the event of a serious error it may break the connection off entirely, informing both user processes. User programs may either deal in terms of packets, or ignore packet

boundaries and treat the connection as two uni-directional streams of 8-bit or 16-bit bytes.

On top of the connection facility "user" programs build other facilities, such as file access, interactive terminal connections, and data in other byte sizes, such as 36 bits. The meaning of the packets or bytes transmitted through a connection is defined by the particular higher-level protocol in use.

In addition to reliable communication, the protocol provides flow control, includes a way by which prospective communicants may get in touch with each other (called *contacting* or *rendezvous*), and provides various network maintenance and housekeeping facilities. These are discussed later.

### 15.3.2 Contact Names: Chaosnet Software Protocol

When first establishing a connection, it is necessary for the two communicating processes to contact each other. In addition, in the usual user/server situation, the server process does not exist beforehand and needs to be created and made to execute the appropriate program.

We chose to implement contacting in an asymmetric way. (Once the connection has been established everything is completely symmetric.) One process is designated the *user*, and the other is designated the *server*. The server has some *contact name* to which it *listens*. The user process requests its local operating system to connect it to the server, specifying the network node and contact name of the server. The local operating system sends a message (a *Request for Connection*) to the remote operating system, which examines the contact name and creates a connection to a listening process, creates a new server process and connects to it, or rejects the request.

Automatically discovering to which host to connect in order to obtain a particular service is a subject for higher-level protocols and for further research. It is not dealt with by Chaosnet.

Once a connection has been established, there is no more need for the contact name and it is discarded. Indeed, often the contact name is simply the name of a service (such as "TELNET") and several users should be able to have simultaneous connections to separate instances of that service, so contact names must be reusable.

In the case where two existing processes that already know about each other want to establish a connection, we arbitrarily designate one as the listener (server) and the other as the requester (user). The listener somehow generates a "unique" contact name, somehow communicates it to the requester, and listens for it. The requester requests to connect to that contact name and the connection is established. In the most common case of establishing a second connection between two processes which are already connected, the index number of the first connection can serve as a unique contact name.

Contact names are restricted to strings of uppercase letters, numbers, and ASCII punctuation. The maximum length of a contact name is limited only by the packet size, although on ITS hosts the names of automatically started servers are limited by the file system to six characters.

The contact names for Chaosnet connections are retained in the connection data structures. The accessor function is **chaos:contact-name**.

The complete details about establishing a connection are given elsewhere. See the section "Connection Establishment: Chaosnet Software Protocol", page 120.

### 15.3.3 Addresses and Indices: Chaosnet Software Protocol

Each node (or host) on the network is identified by an *address*, which is a 16-bit number. These addresses are used in the routing of packets. There is a table (the system hosts table, `SYSBIN;HOSTS2`, in the case of ITS) which relates symbolic host names to numeric host addresses.

An address consists of two fields. The most-significant 8 bits identify a *subnet*, and the least-significant 8 bits identify a host within that subnet. Both fields must be nonzero. A subnet corresponds to a single transmission path. Some subnets are physical Chaosnet cables (*ethers*), while others are other media, for instance an interface between a PDP-10 and a PDP-11. The significance of subnets will become clear when routing is discussed. See the section "Routing: Chaosnet Software Protocol", page 114.

When a host is connected to an ether, the host's hardware address on that ether is the same as its software address, including the subnet field.

A connection is specified by the names of its two ends. Such a name consists of a 16-bit host address and a 16-bit connection index, which is assigned by that host, as the name of the entity inside the host which owns the connection. The only requirements placed by the protocol on indices are that they be nonzero and that they be unique within a particular host; that is, a host may not assign the same index number to two different connections unless enough time has elapsed between the closing of the first connection and the opening of the second connection that confusion between the two is unlikely.

Typically the least-significant $n$ bits of an index are used as a subscript into the operating system's tables, and the most-significant 16-$n$ bits are incremented each time a table slot is reused, to provide uniqueness. The number of uniquizing bits must be sufficiently large, compared to the rate at which connection-table slots are reused, that if two connections have the same index, a packet from the old connection cannot sit around in the network (for example, in buffers inside hosts or bridges) long enough to be seen as belonging to the new connection.

It is important to note that packets are *not* sent between hosts (physical computers). They are sent between user processes; more exactly, between channels attached to

user processes. Each channel has a 32-bit identification, which is divided into subnet, host, index, and uniquization fields. From the point of a view of a user process using the network, the Network Control Program section of the host's operating system is part of the network, and the multiplexing and demultiplexing it performs is no different from the routing performed by other parts of the network. It makes no difference whether two communicating processes run in the same host or in different hosts.

Certain control packets, however, are sent between hosts rather than users. This is visible to users when opening a connection; a contact name is only valid with respect to a particular host. This is a compromise in the design of Chaosnet, which was made so that an operational system could be built without first solving the research and engineering problems associated with making a diverse set of hosts into a uniform, one-level name space.

### 15.3.4 Packet Numbers: Chaosnet Software Protocol

There are two kinds of packets, controlled and uncontrolled. Controlled packets are subject to error-control and flow-control protocols, which guarantee that each controlled packet is delivered to its destination exactly once, that the controlled packets belonging to a single connection are delivered in the same order they were sent, and that a slow receiver is not overwhelmed with packets from a fast sender. (See the section "Flow and Error Control: Chaosnet Software Protocol", page 117.) Uncontrolled packets are simply transmitted; they will usually but not always arrive at their destination exactly once. The protocol for using them must take this into account.

Each controlled packet is identified by an unsigned 16-bit *packet number*. Successive packets are identified by sequential numbers, with wrap-around from all 1s to all 0s. When a connection is first opened, each end numbers its first controlled packet (RFC or OPN) however it likes, and that sets the numbering for all following packets.

Packet numbers should be compared modulo 65536 (2 to the 16th), to ensure correct handling of wrap-around cases. On a PDP-11, use the instructions

```
CMP A,B
BMI A_is_less
```

Do not use the BLT or BLO instruction. On a PDP-10, use the instructions

```
SUB A,B
TRNE A,100000
 JRST A_is_less
```

Do not use the CAMGE instruction. On a Symbolics computer, use the code

```
(IF (BIT-TEST #o100000 (- A B))
    <A is less>)
```

Do not use the LESSP (or <) function.

### 15.3.5 Packet Contents: Chaosnet Software Protocol

A packet consists of a header, which is 8 16-bit words, and zero or more 8-bit or 16-bit bytes of accompanying data. In addition there are three words put on by the hardware, described earlier in this paper.

The following are the 8 header words:

*Operation*
The most-significant 8 bits of this word are the *Opcode* of the packet, a number which tells what the packet means. The 128 opcodes with high-order bit 0 are for the use of the network itself. The 128 opcodes with high-order bit 1 are for use by users. The various opcodes are described elsewhere. See the section "Chaosnet Software Protocol – Details", page 120.

The least-significant 8 bits of this word are reserved for future use, and must be zero.

*Count* The most-significant 4 bits of this word are the forwarding count, which tells how many times this packet has been forwarded by bridges. Its use is explained elsewhere; See the section "Routing: Chaosnet Software Protocol", page 114.

The least-significant 12 bits of this word are the data byte count, which tells the number of 8-bit bytes of data in the packet. The minimum value is 0 and the maximum value is 488. Note that the count is in 8-bit bytes even if the data are regarded as 16-bit bytes.

The byte count must be consistent with the actual length of the hardware packet. Since the hardware cyclic redundancy check algorithm is not sensitive to extra zero bits, packets whose hardware length disagrees with their software length are discarded as hardware errors.

*Destination Address*
This word contains the network address of the destination host to which this packet should be sent.

*Destination Index*
This word contains the connection index at the destination host of the connection to which this packet belongs, or 0 if this packet does not belong to any connection.

*Source Address*
This word contains the network address of the source host which originated this packet.

*Source Index*
This word contains the connection index at the source host of the connection to which this packet belongs, or 0 if this packet does not belong to any connection.

*Packet Number*
If this is a controlled packet, this word contains its identifying number.

*Acknowledgement*
The use of this word is described elsewhere. See the section "Flow and Error Control: Chaosnet Software Protocol", page 117.

### 15.3.6 Data Formats: Chaosnet Software Protocol

Data transmitted through Chaosnet generally follow Symbolics standards. Bits and bytes are numbered from right to left, or least-significant to most-significant. The first 8-bit byte in a 16-bit word is the one in the arithmetically least-significant position. The first 16-bit word in a 32-bit double-word is the one in the arithmetically least-significant position.

The character set used is dictated by the higher-level protocol in use. Telnet and Supdup, for example, each specifies its own ASCII-based character set. The "default" character set, used for new protocols and for text that appears in the basic Chaosnet protocol (such as contact names) is the Symbolics character set. See the section "The Character Set" in *Reference Guide to Streams, Files, and I/O*. This is basically ASCII augmented with additional printing characters and a different set of format-effector (or "control") characters.

Because the rules for bit numbering conflict with the native byte-ordering in PDP-10s, and because it is quite expensive to rearrange the bytes using the PDP-10 instruction set, PDP-11s which act as front-ends for PDP-10s must reformat packets passing through them, and PDP-10s interfaced directly to the network must have interfaces capable of rearranging the bytes. This requires that the network protocols explicitly specify which portions of each type of packet are 8-bit bytes and which are 16-bit bytes. In general the header is 16-bit bytes and the data field is 8-bit bytes, but certain packet types (OPN, STS, RUT, and opcodes 300 through 377) have 16-bit bytes in the data field. Use of 32-bit data is rare, so no provision is made for putting 32-bit data into the standard format for PDP-10s. On our current network PDP-10s are the only hosts which require this packet reformatting assistance, because most modern computers number their bits and bytes from least-significant to most-significant.

The effect of this is that user programs that use the Chaosnet always see the data in a packet and its header in the native form of the machine they are running on, and the necessary conversions are automatically applied by the network. This statement applies to the order of bits and bytes within a word, but not to the character set (when packets contain textual data) which is dictated by protocols.

Unlike some other network protocols, Chaosnet does not use any software checksumming. Because of the diversity of hosts with different architectures attached to the Chaosnet, it is impossible to devise a checksumming algorithm which can be executed compatibly and efficiently on all hosts. Instead, Chaosnet relies on error-checking hardware in the network interfaces, and assumes that other sources of packet damage that checksums could detect, such as software bugs in a Network Control Program, either do not occur or will produce symptoms so obvious that they will be detected and fixed immediately.

### 15.3.7 Routing: Chaosnet Software Protocol

*Routing* consists of deciding how to deliver a packet to the network node specified by the destination address field of the packet. Having reached that node, the packet can trivially be delivered to the destination user process via the destination index. In general routing may be a multistep process involving transmission through several subnets, since there may not be a direct hardware connection between the source and the destination. Note that the routing decision is made separately for each packet, with no reference to the concept of connections.

Any host that is connected to more than one subnet acts as a *bridge* and *forwards* packets from one subnet to another when necessary. There could also be hardware bridges which are not hosts, although we have not yet designed any such device. Since routing does not depend on connections, a bridge is a very simple device (or program) which does not need much state. This makes the bridge function inexpensive to piggyback onto a computer which is also performing other functions, and makes reliable bridge software easy to implement.

The difference between a *bridge* and a *gateway*, in our terminology, is that a bridge forwards packets from one sub-Chaosnet to another, without modifying the packets or understanding them other than to look at the destination address and increment the forwarding count, and does not deal with connections nor with flow control, while a gateway interconnects two networks with differing protocols and must understand and translate the information passing through it. Gateways may also have to deal with flow and error control because they connect networks with slow or differing speeds. Bridges are suitable for local networks while gateways are suitable for long-distance networks and for connecting networks not produced by the same organization.

To prevent routing loops, each packet contains a forwarding-count field. Each bridge that forwards the packet increments this count; if the count reaches its maximum value the packet is discarded. The error-control protocol will recover discarded packets, or decide that no viable connection can be established between the two hosts.

The implementation of routing in an operating system is as follows, given a packet to be routed, which may have come in from the network or may have been originated by the local host. First, check the packet's destination address. If it is this host, receive the packet. Otherwise, increment the forwarding count and discard the packet if it has been forwarded too many times. If the destination is some other host on a subnet to which this host is directly connected, transmit the packet on that subnet; the destination host should receive it. If the destination is a host on a subnet of which this host has no knowledge, look up the subnet in the host's *routing table* to find the best bridge to that subnet, and transmit the packet to that bridge.

Each host has a routing table, indexed by subnet number, which tells how to get packets to hosts on that subnet. Each entry contains: (exact details may vary depending on implementation)

*Type* The type of connection between the host and this subnet. This can be one of *Direct*, *Bridge*, or *Fixed Bridge*. *Direct* means a physical connection such as a Chaosnet interface. *Bridge* means an indirect connection, via a packet-forwarding bridge. Which bridge is best to use is to be discovered by this routing mechanism. *Fixed Bridge* is the same except that the automatic mechanism is not to change which bridge is used. This is useful to set up explicit routing for purposes such as network debugging.

*Address*
Identifies the connection to this subnet in a way which depends on the type. For a direct connection, this identifies the piece of hardware which implements the connection. (It might be a UNIBUS address.) For a bridge or a fixed bridge, this is the network address of the bridge.

*Cost* A measure of the cost of sending a packet through this route. Costs are used to select the best route from among alternatives in a way described below. For a direct connection, the cost is 10 for a direct interface between two computers (for example, between a PDP-10 and its front-end PDP-11), 11 for a Chaosnet ether cable, 20 for a slow medium such as an asynchronous line, and so on. For a bridge or a fixed bridge, the cost is specified by the bridge in a RUT packet.

The routing table is initialized with the number of a more or less arbitrary existent host and a high cost, for each subnet to which the host is not directly connected. Until the correct bridge is discovered (which normally happens within a minute of coming up), packets for that subnet will be bounced off of that arbitrary host, which probably knows the right bridge to forward them to.

The cost for subnets accessed via bridges is increased by 1 every 4 seconds, thus typically doubling after a minute. When the cost reaches a "high" value, it sticks there, preventing problems with arithmetic overflow. The purpose of the increasing cost is to discount the value of old information. The cost for subnets accessed via direct connections and fixed bridges does not increase.

Every 15 seconds, a bridge advertises its presence by broadcasting a routing (RUT) packet on each subnet to which it is directly connected. Each host on that subnet receives the RUT packet and uses it to update its routing table. If the host's routing table says to access a certain subnet via bridges, and the RUT packet says that this is the best bridge to that subnet, the routing table is updated to say that this bridge should be used.

Note that it is important that the rate at which the costs increase with time be slow enough that it takes more than twice the broadcast interval to increase the cost of one hop to be more than the cost of two hops. Otherwise the routing algorithm is not well-behaved. Suppose subnet A has two bridges ($\alpha$ and $\beta$) on it, and bridge $\alpha$ is connected to subnet B but bridge $\beta$ is not (it goes to some other irrelevant subnet). Then if the costs increase too fast and bridges $\alpha$ and $\beta$ do not broadcast their RUT packets exactly simultaneously, sometimes packets for subnet B

may be sent to bridge $\beta$ because its cost appears lower. Bridge $\beta$ will then send them to bridge $\alpha$, where they should have gone directly. In more complicated situations packets can go around in a circle some of the time.

The source address of a RUT packet must be the hardware address of the bridge on the particular subnet on which the packet is broadcast. The destination address of a RUT packet must be zero; RUT packets are not forwarded onto other subnets. The byte count of a RUT packet is a multiple of 4 and the packet contains up to 122 pairs of 16-bit words:

word 1 The subnet number of a subnet which this bridge can get to, directly or indirectly, right-adjusted.

word 2 The cost of sending to that subnet via this bridge. This is the current cost from the bridge's routing table, plus the cost for the subnet on which the routing packet is being broadcast. Adding the subnet cost eliminates loops, and prefers one-hop paths over two-hop paths.

When a host receives a RUT packet, it processes each 2-word entry by comparing the cost for that subnet against its current cost; if it is less or equal the cost and the address of the bridge are entered into the routing table, provided that that subnet's routing table entry is not of the Direct or Fixed Bridge type.

When there are multiple equivalent bridges, the traffic is spread among them only by virtue of their RUT packets being sent at different times, so that sometimes one bridge has the lower cost, and sometimes the other. If this isn't adequate, hosts could have hairier routing tables which remember more than one possible route and use them according to their relative costs, but so far this has not been necessary since the network traffic is not so high as to saturate any one bridge.

The design of this routing scheme is predicated on the assumption that the network geometry is simple, there are few multiple paths, and the length of any path is quite short. This makes more sophisticated schemes unnecessary.

An important feature of this routing scheme is that the size of the table is proportional to the number of subnets, not to the number of hosts. Thus it does not take up an inordinate amount of memory in a small computer, and no complicated dynamic allocation schemes are required.

In the case of a PDP-10 which accesses the Chaosnet through a front-end PDP-11, we define the interface between the two computers to be a subnet, and regard the PDP-11 as a bridge which forwards packets between the network and the PDP-10. This gives the PDP-10 and the PDP-11 separate addresses so that we can choose to talk to either one, even though they are part of the same computer system. This is occasionally useful for maintenance purposes. It becomes more useful when the front-end PDP-11 has peripherals which are to be accessed through the Chaosnet, since they can simply look like hosts on that PDP-11's private subnet.

In the case of a host which is attached to more than one subnet, it is undesirable

for the host to have more than one address, since this would complicate user programs which use addresses. Instead, one of the host's network attachments is designated as primary, and that address is used as the host's single address. The other attachments are regarded as bridges which can forward to that host. Sometimes, we simplify the routing by inventing a new subnet which contains only that host and has no physical realization. The host's address is an address on that fake subnet. All of the host's network attachments are regarded as bridges which know how to forward packets to that subnet.

The ITS host table allows a host to have multiple addresses on multiple networks, but when you ask for *the* address of a certain host on a certain network you only get back the primary address. All packets coming from that host have that as their source address.

### 15.3.8 Flow and Error Control: Chaosnet Software Protocol

The Network Control Programs (NCPs) conspire to ensure that data packets are sent from user to user with no garbling, duplications, omissions, or changes of order. Secondarily, the NCPs attempt to achieve a maximum rate of flow of data, and a minimum of overhead and retransmission.

The fundamental basis of flow-control and error-control in Chaosnet is *retransmission*. Packets which are damaged in transmission, which won't fit in buffers, which are duplicated or out-of-sequence, or which otherwise are embarrassing are simply discarded. Packets are periodically retransmitted until an indication that they have been successfully received is returned. This retransmission is end-to-end; any intermediate bridges do not participate in flow-control and error-control, and hence are free to discard any packets they wish.

There are actually two kinds of packets, *controlled* and *uncontrolled*. Controlled packets are retransmitted and delivered reliably; most packets, including all packets used by the user (except for UNC packets), are of this type. Uncontrolled packets are not retransmitted; these are used for certain lower-level functions of the protocol such as the implementation of flow and error control. The usage of these packets is designed so that they need not be delivered reliably.

Retransmission of a packet continues until stopped by a signal from the receiver to the sender called a *receipt*. A receipt contains a *packet number*, and indicates that all controlled packets with a packet number less than or equal (modulo 65536) to that number have been successfully received, and therefore need not be retransmitted any more. A receipt does not indicate that these packets have been processed by the destination user process; it simply indicates that they have successfully arrived in the destination host, and are guaranteed to be there when the user process asks for them.

There is another signal from the receiver to the sender, called an *acknowledgement*. An acknowledgement also contains a packet number, and indicates that all controlled

packets with a packet number less than or equal (modulo 65536) to that number have been read by the destination user process. This is used to implement flow-control. Note that acknowledgement of a packet implies receipt of that packet. In fact, if the receiving process does not fall behind, explicit receipts need not be sent, because the receiving host will not have to buffer any packets, but will acknowledge them as soon as they arrive.

The purpose of flow-control is to match the speeds of the sending and receiving processes. The extremes to be avoided are, on the one hand, too small a "buffer size" causing the data transmission rate to be slower than it could be, and on the other hand, large numbers of packets piling up in the network because the sender is sending faster than the receiver is receiving. It is also necessary to be aware that receipts and acknowledgements must be transmitted through the network, and hence have an associated cost.

Chaosnet flow-control operates by controlling the number of packets "in the network". These are packets which have been emitted by the sending user process, but have not been acknowledged. We define a *window* into the set of packet numbers. The beginning of this window is the first packet number which has not been acknowledged, and the width of the window is a fixed number established when the connection is opened. The sending process is only allowed to emit packets whose packet numbers lie within the window. Once it has emitted all of the packets in the window, the window is said to be full. Thus, the size of the window is the "buffer size" for the connection, and is the maximum number of packets that may need to be buffered inside an NCP (sending or receiving). Acknowledgements move the window, making it not full, and allowing the sending process to emit additional packets.

We do not receipt and acknowledge every single controlled packet that is transmitted through a connection, since that would double or triple the number of packets sent through the network to move a given amount of data. Instead we batch the receipts and acknowledgements. But if acknowledgements are not sent sufficiently often, the data will not flow smoothly, because the window will often appear full to the sender when it is not. If receipts are not sent sufficiently often, there will be unnecessary retransmissions.

Whenever a packet is sent through a connection, an acknowledgement for the reverse direction of that connection is "piggy-backed" onto it, using the Acknowledgement field in the packet header. For interactive applications, where there is much traffic in both directions, this provides all the necessary acknowledgement and receipting with no need to send any extra packets through the network.

When this does not suffice, STS (status) packets are generated to carry receipts and acknowledgements. STS packets are uncontrolled, since they are part of the mechanism that implements controlled packets. If an STS packet is duplicated, it does no harm. If an STS packet is lost, mechanisms exist which will cause a

replacement to be generated later. An STS packet carries separate receipt and acknowledgement packet numbers.

When a user process reads a packet from the network, if the number of packets which should have been acknowledged but have not been is more than 1/3 the window size, an STS is generated to acknowledge them. Thus the preferred batch size for acknowledgement is 1/3 the window size. The advantage of this size is that if one STS is lost, another will be generated before the window fills up (at the 2/3 point).

When a packet is received with the same packet number as one which has already been successfully received, this is evidence of unnecessary retransmission, and an STS is generated to carry a receipt back to the sender. If this STS is lost, the next retransmission will stimulate another one. Thus receipts are normally implied by acknowledgements, and only sent separately when there is evidence of unnecessary retransmission.

Retransmission consists of sending all unreceipted controlled packets, except those that were last sent very recently (within 1/30'th of a second in ITS.) Retransmission occurs every 1/2 second. This interval is somewhat arbitrary, but should be close to the response time of the systems involved. Retransmission also occurs in response to an STS packet, so that a receiver may cause a faster retranmission rate than twice a second if it so desires. This should never cause useless retransmission, since STS carries a receipt, and very-recently-transmitted packets, which might still be in transit through the network, are not retransmitted.

Another operation is *probing*, which consists of sending a SNS packet, in the hope of eliciting either an STS or a LOS, depending on whether the other side believes the connection exists. Probing is used periodically as a way of testing that the connection is still open, and also serves as a way to get STS packets retransmitted as a hedge against the loss of an acknowledgement, which could otherwise stymie the connection. SNS packets are uncontrolled.

We probe every five seconds on connections which have unacknowledged packets outstanding (a nonempty window) and on connections which have not received any packets (neither data nor control) for one minute. If a connection receives no packets for 1 1/2 minutes, this means that at least 5 probes have been ignored, and the connection is declared to be broken; either the remote host is down or there is no viable path through the network between the two hosts.

The receiver can generate "spontaneous" STSs, to stimulate retransmission and keep things moving on fast devices with insufficient buffering for 1/2 second worth of packets. This provides a way for the receiver to speed up the retransmission timeout in the sender, and to make sure that acknowledges are happening often enough.

Note that the network still functions if either or both parties to a connection ignore the window. The window is simply an improver of efficiency. Receipts have the

same property. This allows very small implementations to be compatible with the same protocol, which is useful for applications such as bootstrapping through the network.

It would be possible to have dynamic adjustment of the window size in response to observed behavior. The STS packet includes the window size so that changes to it can be communicated. However, this has not been found necessary in practice. Each higher-level protocol has a standard pre-determined window size, which it establishes when it first opens a connection, and this seems to be close enough to optimum that careful dynamic adjustment of it wouldn't make a big difference.

This scheme for flow-control and error-control is based on several assumptions. It is assumed that the underlying transmission media have their own checking, so that they discard all damaged packets, making packet checksums unnecessary at the protocol level. The transit time through the network is assumed to be fast, so that a fairly small retransmission interval is practical, and negative acknowledgements are not necessary. The error rate is assumed to be low so that overall efficiency is not affected by the simple error recovery scheme of simply retransmitting all outstanding packets. It is assumed that no reformatting of packets occurs inside the network, so that flow-control and error-control can operate on a packet basis rather than a byte basis.

## 15.4 Chaosnet Software Protocol -- Details

In the following sections, each of the packet *Opcodes* and the use of that packet type in the protocol is described. Opcodes are given as an octal number, a three-letter code, and a name.

Unless otherwise specified, the use of the fields in the packet header is as follows. The source and destination address and index denote the two ends of the connection; when an end does not exist, as during initial connection establishment, that index is zero. The opcode, byte count, and forwarding count fields have no variations. The packet number field contains sequential numbers in controlled packets; in uncontrolled packets it contains the same number as the next controlled packet will contain. The acknowledgement field contains the packet number of the last packet seen by the user.

### 15.4.1 Connection Establishment: Chaosnet Software Protocol

The following packet types are associated with creating and destroying connections. First the packets are described and then the details of the various connection-establishment protocols are given.

All connections are initiated by the transmission of an RFC from the user to the server. The data field of the packet contains the contact name. The contact name

can be followed by arbitrary arguments to the server, delimited by a space character. The destination index field of an RFC contains 0 since the destination index is not known yet.

RFC is a controlled packet; it is retransmitted until some sort of response is received. Because RFCs are not sent over normal, error-controlled connections, a special way of detecting and discarding duplicates is required. When an NCP receives an RFC packet, it checks all pending RFCs and all connections which are in the Open or RFC-received state, to see if the source address and index match; if so, the RFC is a duplicate and is discarded. (See the section "Chaosnet Connection States", page 127.)

A server process informs the local NCP of the contact name to which it is listening by sending a LSN packet, with the contact name in the data field. This packet is never transmitted anywhere through the network. It simply serves as a convenient buffer to hold the server's contact name. When an RFC and a LSN containing the same contact name meet, the LSN is discarded and the RFC is given to the server, putting its connection into the RFC-received state. (See the section "Chaosnet Connection States", page 127. ) The server reads the RFC and decides whether or not to open the connection.

OPN is the usual positive response to RFC. The source index field conveys the server's index number to the user; the user's index number was conveyed in the RFC. The data field of OPN is the same as that of STS; it serves mainly to convey the server's window-size to the user. The Acknowledgement field of the OPN acknowledges the RFC so that it will no longer be retransmitted.

OPN is a controlled packet; it is retransmitted until it is acknowledged. Duplicate OPN packets are detected in a special way; if an OPN is received for a connection which is not in the RFC-sent state, it is simply discarded and an STS is sent. (See the section "Chaosnet Connection States", page 127. ) This happens if the connection is opened while a retransmitted OPN packet is in transit through the network, or if the STS which acknowledges an OPN is lost in the network.

CLS is the negative response to RFC. It indicates that no server was listening to the contact name, and one couldn't be created, or for some reason the server didn't feel like accepting this request for a connection, or the destination NCP was unable to complete the connection (for example, connection table full.)

CLS is also used to close a connection after it has been open for a while. Any data packets in transit may be lost. Protocols which require a reliable end-of-data indication should use the mechanism for that before sending CLS. (See the section "End-of-data: Chaosnet Software Protocol", page 124. )

The data field of a CLS contains a character-string explanation of the reason for closing, intended to be returned to a user as an error message.

CLS is an uncontrolled packet, so that the program which sends it may go away immediately afterwards, leaving nothing to retransmit the CLS. Since there is no error recovery or retransmission mechanism for CLS, the use of CLS is necessarily

optional; a process could simply stop responding to its connection. However, it is desirable to send a CLS when possible to provide an error message for the user.

This is a response to RFC which indicates that the desired service is not available from the process contacted, but may be available at a possibly different contact name at a possibly different host. The data field contains the new contact name and the Acknowledgement field--exceptionally--contains the new host number. The issuer of the RFC should issue another RFC to that address. FWD is an uncontrolled packet; if it is lost in the network, the retransmission of the RFC will presumably stimulate an identical FWD.

This is another kind of response to RFC. The data field contains the entirety of the response, and no connection is established. ANS is an uncontrolled packet; if it is lost in the network, the retransmission of the RFC will presumably stimulate an identical ANS.

There are several connection-initiation protocols implemented with these packet types. In addition to those described here, there is also a broadcast mechanism. See the section "Broadcast: Chaosnet Software Protocol", page 125.

When an RFC arrives at a host, the NCP finds a user process that is listening for this RFC's contact name, or creates a server process to provide the desired service, or responds to the RFC itself if it knows how to provide the requested service, or refuses the request for connection. The process that serves the RFC chooses which connection-initiation protocol to follow. This process is given the RFC as data, so that it can look at the contact name and any arguments that may be present.

A *stream connection* is initiated by an RFC, transmitted from user to server. The server returns an OPN to the user, which responds with an STS. These three packets convey the source and destination addresses, indices, initial packet numbers, and window sizes between the two NCPs. In addition a character-string argument can be conveyed from the user to the server in the RFC.

The OPN serves to acknowledge the RFC and extinguish its retransmission. It also carries the server's index, initial packet number, and window size. The STS serves to acknowledge the OPN and extinguish its retransmission. It also carries the user's window size; the user's index and initial packet number were carried by the RFC. Retransmission of the RFC and the OPN provides reliability in the face of lost packets. If the RFC is lost, it will be retransmitted. If the STS is lost, the OPN will be retransmitted. If the OPN is lost, the RFC will be retransmitted superfluously and the OPN will be retransmitted since no STS will be sent.

The exchange of an OPN and an STS tells each side of the connection that the other side believes the connection is open; once this has happened data may begin to flow through the connection. The user process may begin transmitting data when it sees the OPN. The server process may begin transmitting data when it sees the STS. These rules ensure that data packets cannot arrive at a receiver before it knows and agrees that the connection is open. If data packets did arrive before

then, the receiver would reject them with a LOS, believing them to be a violation of protocol, and this would destroy the connection before it was ever fully established.

Once data packets begin to flow, they are subject to the flow and error control protocol. (See the section "Flow and Error Control: Chaosnet Software Protocol", page 117. ) Thus a stream connection provides the desired reliable, bidirectional data stream.

A *refusal* is initiated by an RFC in the same way, but the server returns CLS rather than OPN. The data field of the CLS contains the reason for refusal to connect.

A *forwarded connection* is initiated by an RFC in the same way, but the server returns a FWD, telling the user another place to look for the desired service.

A *simple transaction* is initiated by an RFC from user to server, and completed by an ANS from server to user. Since a full connection is not established and the reliable-transmission mechanism of connections is not used, the user process cannot be sure how many copies of the RFC the server saw, and the server process cannot be sure that its answer got back to the user. This means that simple transactions should not be used for applications where it is important to know whether the transaction was really completed, nor for applications in which repeating the same query might produce a different answer. Simple transactions are a simple efficient mechanism for applications such as extracting a small piece of information (for example, the time of day) from a central data-base.

### 15.4.2 Status Packets: Chaosnet Software Protocol

STS is an uncontrolled packet which is used to convey status information between NCPs. The Acknowledgement field in the packet header contains an acknowledgement, that is, the packet number of the last packet given to the receiving user process. The first 16-bit byte in the data field contains a receipt, that is, a packet number such that all controlled packets up to and including that one have been successfully received by the NCP. The second 16-bit byte in the data field contains the window size for packets sent in the opposite direction (to the end of the connection which sent the STS). The byte count is presently always 4. This will change if the protocol is revised to add additional items to the STS packet.

SNS is an uncontrolled packet whose sole purpose is to cause the other end of the connection to send back an STS. This is used by the *probing* mechanism. See the section "Flow and Error Control: Chaosnet Software Protocol", page 117.

LOS is an uncontrolled packet which is used by one NCP to inform another of an error. The data field contains a character-string explanation of the problem. The source and destination addresses and indices are simply the destination and source addresses and indices, respectively, of the erroneous packet, and do not necessarily correspond to a connection. When an NCP receives a LOS whose destination corresponds to an existent connection and whose source corresponds to the supposed

other end of that connection, it *breaks* the connection and makes the data field of the LOS available to the user as an error message. Other LOSs that don't correspond to connections are simply ignored.

LOS is sent in response to situations such as: arrival of a data packet or an STS for a connection that does not exist or is not open, arrival of a packet from the wrong source for its destination, arrival of a packet containing an undefined opcode or too large a byte count, and so on.

LOSs are given to the user process so that it may read the error message.

No LOS is given in response to an OPN to a connection not in the RFC-Sent state, nor in response to a SNS to a connection not in the Open state, nor in response to a LOS to a nonexistent or broken connection. These rules are important to make the protocols work without timing errors. An OPN or a SNS to a nonexistent connection elicits a LOS.

### 15.4.3 Data: Chaosnet Software Protocol

Opcodes 200 through 277 (octal) are controlled packets with user data in 8-bit bytes in the data field. The NCP treats all 64 of these opcodes identically; some higher-level protocols use the opcodes for their own purposes. The standard default opcode is 200.

Opcodes 300 through 377 (octal) are controlled packets with user data in 16-bit bytes in the data field. The NCP treats all 64 of these opcodes identically; some higher-level protocols use the opcodes for their own purposes. The standard default opcode for 16-bit data is 300.

This is an uncontrolled packet with user data in 8-bit bytes in the data field. It exists so that user-level programs may bypass the flow-control mechanism of Chaosnet protocol. Note that the NCP is free to discard these packets at any time, since they are uncontrolled. Since UNCs are not subject to flow control, discarding may be necessary to avoid running out of buffers. A connection may not have more input packets queued awaiting the attention of the user program than the window size of the connection, except that you are always allowed to have one UNC packet queued. If no normal data packets are in use, up to one more UNC packet than the window size may be queued.

UNC packets are also used by the standard protocol for encapsulating packets of foreign protocols for transmission through Chaosnet . See the section "Using Foreign Protocols in Chaosnet", page 135.

### 15.4.4 End-of-data: Chaosnet Software Protocol

EOF is a controlled packet which serves as a "logical end of data" mark in the packet stream. When the user program is ignoring packets and treating a Chaosnet connection as a conventional byte-stream I/O device, the NCP uses the EOF packet

to convey the notion of conventional end-of-file from one end of the connection to the other. When the user program is working at the packet level, it may transmit and receive EOFs.

It is illegal to put data in an EOF packet; in other words, the byte count should always be zero. Most Chaosnet implementations will simply ignore any data that is present in an EOF.

EOF packets are used in the following protocol which is the recommended way to reliably determine that all data have been transferred before closing a connection (in applications where that is an important consideration).

The important issue is that neither side may send a CLS until both sides are sure that all the data have been transmitted. After sending all the data it is going to send, including an EOF packet to mark the end, the sending process waits for all packets to be acknowledged. This ensures that the receiver has seen all the data and knows that no more data are to come. The sending process then closes the connection. When the receiving process sees an EOF, it knows that there are no more data. It does *not* close the connection until it sees the sender close it, or until a brief timeout elapses. The timeout is to provide for the case where the sender's CLS gets lost in the network (CLS cannot be retransmitted). The timeout is long enough (a few seconds) to make it unlikely that the sender will not have seen the acknowledgement of the EOF by the time the timeout is over.

To use this protocol in a bidirectional fashion, where both parties to the connection are sending data simultaneously, it is necessary to use an asymmetrical protocol. Arbitrarily call one party the user and the other the server. The protocol is that after sending all its data, each party sends an EOF and waits for it to be acknowledged. The server, having seen its EOF acknowledged, sends a second EOF. The user, having seen its EOF acknowledged, looks for a second EOF and *then* sends a CLS and goes away. The server goes away when it sees the user's CLS, or after a brief timeout has elapsed. This asymmetrical protocol guarantees that each side gets a chance to know that both sides agree that all the data have been transferred. The first CLS will only be sent after both sides have waited for their (first) EOF to be acknowledged.

### 15.4.5 Broadcast: Chaosnet Software Protocol

Chaosnet includes a generalized broadcast facility, intended to satisfy needs such as:

- Locating services when it is not known what host they are on.
- Internal communications of other protocols using Chaosnet as a transmission medium, such as routing in their own address spaces.
- Reloading and remote debugging of Chaosnet bridge computers.
- Experiments with radically different protocols.

A BRD packet works much like an RFC packet; it contains the name of a server to be communicated with, and possibly some arguments. Unlike an RFC, which is delivered to a particular host, a BRD is broadcast to all hosts. Only hosts which understand the service it is looking for will respond. The response can be anything which is valid as a response to RFC. Typically BRD will be used in a simple-transaction mode, and the response will be an ANS packet. Actually it can be any number of ANS packets since multiple hosts may respond. BRD can also be used to open a full byte-stream connection to a server whose host is not known. In this case the response will be an OPN packet; only the first OPN succeeds in opening a connection. CLS is also a valid response, but only as a true negative response; BRDs for unrecognized or unavailable services should be ignored and no CLS should be sent, since some other host might be able to provide the service.

The TIME and STATUS protocols will work through BRD packets as well as RFC packets. (See the section "Higher-level Chaosnet Protocols", page 129.) No other standard protocols need to be able to work with BRD packets.

The data field of a BRD contains a subnet bit map followed by a contact name and possible arguments. The subnet bit map has a "1" for each subnet on which this packet is to be broadcast to all hosts; these bits are turned off as the packets flow through the network, to avoid loops. The sender initializes the bit map with 1s for whichever subnets he desires (often all of them).

In the packet header, the destination host and index are 0. The source host and index are who to send the reply (ANS or OPN) to. The acknowledgement field contains the number of bytes in the bit map (this would normally be 32, but may be changed in the future). The number of bytes in the bit map is required to be a multiple of 4. Bits in the bitmap are numbered from right to left within a byte and from earlier to later bytes; thus the bit for subnet 1 is the bit with weight 2 in the first byte of the data field. Bits that lie outside of the declared length of the bit map are considered to be zero; thus the BRD is not transmitted to those subnets.

After the subnet bit map there is a contact name and arguments, exactly as in an RFC. Operating systems should treat incoming BRD packets exactly like RFC, even to the extent that a contact name of STATUS must retrieve the host's network throughput and error statistics. BRD packets will never be refused with a "CLS", however; broadcast requests to nonexistent servers should simply be ignored, and no CLS reply should be sent. Most operating systems will simplify incoming BRD handling for themselves and their users by reformatting incoming BRD packets to look like RFCs; deleting the subnet bit map from the data field and decreasing the byte count. For consistency when this is done the bit map length (in the acknowledgement field) should be set to zero. The packet opcode will remain BRD (rather than RFC).

Operating systems should handle outgoing BRD packets as follows. When a user process transmits a BRD packet over a closed connection, the connection enters a special "Broadcast Sent" state. In this state, the user process is allowed to transmit

additional BRD packets. All incoming packets other than OPNs should be made available for the user process to read, until the allowed buffering capacity is exceeded; further incoming packets are then simply discarded. These incoming packets would normally be expected to consist of ANS, FWD, and CLS packets only. If an OPN is received, and there are no queued input packets, a regular byte-stream connection is opened. Any OPNs from other hosts elicit a LOS reply as usual, as do any ANSs, CLSs, and so on received at this point.

Operating systems should not retransmit BRD packets, but should leave this up to the user program, since only it knows when it has received enough answers (or a satisfactory answer).

BRD packets can be delivered to a host in multiple copies when there are multiple paths through the network between the sender and that host. The bit map only serves to cut down looping more than the forwarding-count would, and to allow the sender to broadcast selectively to portions of the net, but cannot eliminate multiple copies. The usual mechanisms for discarding duplicated RFCs will also cause most duplicated BRDs to be discarded.

BRD packets put a noticeable load on every host on the network, so they should be used judiciously. "Beacons" that send a BRD every 30 seconds all day long should not be used.

### 15.4.6 Low-level: Chaosnet Software Protocol

MNT is a special packet type reserved for the use of network maintenance programs. Normal NCPs should simply discard any MNT packets they receive. MNT packets are an escape mechanism to allow special programs to send packets that are guaranteed not to get confused with normal packets. MNT packets are forwarded by bridges although usually one would not be depending on this.

RUT is a special packet type broadcast by bridges to inform other nodes of the bridge's ability to forward packets between subnets. The source address is the network address of the bridge on the subnet on which the RUT was broadcast. The destination address is zero. The byte count is a multiple of 4, and the data field contains a series of pairs of 16-bit bytes: a subnet number and the "cost" of getting to that subnet via this bridge. The packet number and acknowledgement fields are not used and should contain zero. See the section "Routing: Chaosnet Software Protocol", page 114.

### 15.4.7 Chaosnet Connection States

A user process gets to Chaosnet by means of a capability or channel (dependent on the host operating system) which corresponds to one end of a connection. Associated with this channel are a number of buffers containing controlled packets output by the user and not yet receipted, and data packets received from the network but not yet read by the user; some of these incoming packets are in-order by packet number

and hence may be read by the user, while others are out of order and cannot be read until packets earlier in the stream have been received. Certain control packets are also given to the user as if they were data packets. These are RFC, ANS, CLS, LOS, EOF, and UNC. EOF is the only type that can ever be out-of-order.

Also associated with the channel is a state, usually called the *connection state*. Full understanding of these states depends on the descriptions of packet-types. The state can be one of:

*Open* The connection exists and data may be transferred.

*Closed* The channel does not have an associated connection. Either it never had one or it has received or transmitted a CLS packet, which destroyed the connection.

*Listening*
The channel does not have an associated connection, but it has a contact name (usually contained in a LSN packet) for which it is listening.

*RFC Received*
A *Listening* channel enters this state when an RFC arrives. It can become *Open* if the user process *accepts* the request.

*RFC Sent*
The user has transmitted an RFC. The state will change to *Open* or *Closed* when the reply to the RFC comes back.

*Broadcast Sent*
The user has transmitted a BRD. In this state, the user process is allowed to transmit additional BRD packets. All incoming packets other than OPNs are made available for the user process to read, until the allowed buffering capacity is exceeded; further incoming packets are then simply discarded. These incoming packets would normally be expected to consist of ANS, FWD, and CLS packets only. If an OPN is received, and there are no queued input packets, a regular byte-stream connection is opened (the connection enters the *Open* state). Any OPNs from other hosts elicit a LOS reply as usual, as do any ANSs, CLSs, and so on received at this point.

*Lost* The connection has been broken by receiving a LOS packet.

*Incomplete Transmission*
The connection has been broken because the other end has ceased to transmit and to respond to SNS. Either the network or the foreign host is down. (This can also happen if the local host goes down for a while and then is revived, if its clock runs in the meantime.)

*Foreign*
The channel is talking some foreign protocol, whose packets are encapsulated in UNC packets. As far as Chaosnet is concerned there is no connection. See the section "Using Foreign Protocols in Chaosnet", page 135.

## 15.5 Higher-level Chaosnet Protocols

This section briefly documents some of the higher-level protocols of the most general interest. There are quite a few other protocols which are too specialized to mention here. All protocols other than the STATUS protocol are optional and are only implemented by those hosts that need them. All hosts are required to implement the STATUS protocol since it is used for network maintenance.

### 15.5.1 Chaosnet Status Protocols

All network nodes, even bridges, are required to answer RFCs with contact name STATUS, returning an ANS packet in a simple transaction. This protocol is primarily used for network maintenance. The answer to a STATUS request should be generated by the Network Control Program, rather than by starting up a server process, in order to provide rapid response.

The STATUS protocol is used to determine whether a host is up, to determine whether an operable path through the network exists between two hosts, to monitor network error statistics, and to debug new Network Control Programs and new Chaosnet hardware. The **hostat** function on the Symbolics computer, and the `Hostat` command of the CHATST program on ITS are user ends for this protocol.

The first 32 bytes of the ANS contain the name of the node, padded on the right with zero bytes. The rest of the packet contains blocks of information expressed in 16-bit and 32-bit words, low byte first (PDP-11/Symbolics style). The low-order half of a 32-bit word comes first. Since ANS packets contain 8-bit data (not 16-bit), machines such as PDP-10s which store numbers high byte first will have to shuffle the bytes when using this protocol. The first 16-bit word in a block is its identification. The second 16-bit word is the number of 16-bit words to follow. The remaining words in the block depend on the identification.

This is the only block type currently defined. All items are optional, according to the count field, and extra items not defined here may be present and should be ignored. Note that items after the first two are 32-bit words.

word 0 A number between 400 and 777 octal. This is 400 plus a subnet number. This block contains information on this host's direct connection to that subnet.

word 1 The number of 16-bit words to follow, usually 16.

words 2-3
The number of packets received from this subnet.

words 4-5
The number of packets transmitted to this subnet.

words 6-7
: The number of transmissions to this subnet aborted by collisions or because the receiver was busy.

words 8-9
: The number of incoming packets from this subnet lost because the host had not yet read a previous packet out of the interface and consequently the interface could not capture the packet.

words 10-11
: The number of incoming packets from this subnet with CRC errors. These were either transmitted wrong or damaged in transmission.

words 12-13
: The number of incoming packets from this subnet which had no CRC error when received, but did have an error after being read out of the packet buffer. This error indicates either a hardware problem with the packet buffer or an incorrect packet length.

words 14-15
: The number of incoming packets from this subnet which were rejected due to incorrect length (typically not a multiple of 16 bits).

words 16-17
: The number of incoming packets from this subnet rejected for other reasons (for example, too short to contain a header, garbage byte-count, forwarded too many times.)

If the identification is a number between 0 and 377 octal, this is an obsolete format of block. The identification is a subnet number and the counts are as above except that they are only 16 bits instead of 32, and consequently may overflow. This format should no longer be sent by any hosts.

Identification numbers of 1000 octal and up are reserved for future use.

### 15.5.2 Chaosnet Pulsar Protocol

For network maintenance purposes, certain network nodes support a simple transaction with contact name PULSAR, which controls a "pulsar" feature. This feature periodically transmits a short packet which can be used to test and adjust cable transceivers. The packet consists of the three header words, a zero word, and a word of alternating ones and zeros. It is addressed to host 177777 which is guaranteed not to exist.

The returned ANS contains a single character, which is a digit. A 0 means that the pulsar is turned off. Any other digit indicates the number of sixtieths of a second between pulses. Sending an RFC with a digit as an argument sets the state of the pulsar to that digit, and returns an ANS containing the new state. Pulsars should be off by default, and should only be turned on when debugging the network. The waste of cable bandwidth and machine resources is negligible except in extremely

large networks, since pulsar packets are so short, but when debugging or making measurements on cables using pulsar packets it is important to know where the packets are coming from.

Bridge nodes which implement the PULSAR protocol and possess more than one network interface should should have a single pulsar which transmits on all network interfaces, rather than bothering to provide a more complex protocol by which pulsars on the individual interfaces could be turned on and off.

### 15.5.3 Chaosnet Telnet and Supdup Protocols

The Telnet and Supdup protocols of the Arpanet [TELNET] [SUPDUP] exist in identical form in Chaosnet. These protocols allow access to a computer system as an interactive terminal from another network node.

The contact names are TELNET and SUPDUP. The direct borrowing of the Telnet and Supdup protocols was eased by their use of 8-bit byte streams and only a single connection. Note that these protocols define their own character sets, which differ from each other and from the Chaosnet standard character set.

Chaosnet contains no counterpart of the INR/INS attention-getting feature of the Arpanet. The Telnet protocol sends a packet with opcode 201 in place of the INS signal. This is a controlled packet and hence does not provide the "out of band" feature of the Arpanet INS, however it is satisfactory for the Telnet "interrupt process" and "discard output" operations on the kinds of hosts attached to Chaosnet.

### 15.5.4 Chaosnet File Access Protocol

The FILE protocol is primarily used by Symbolics computers to access files on network file servers. ITS and TOPS-20 are equipped to act as file servers. A user end for the file protocol also exists for TOPS-20 and is used for general-purpose file transfer. For complete documentation on the file protocol, see [FILE].

### 15.5.5 Chaosnet Mail Protocol

The MAIL protocol is used to transmit interuser messages through the Chaosnet. This simple protocol is by no means the last word in mail protocols; however, it is adequate for the mail systems we presently possess.

The sender of mail connects to contact name MAIL and establishes a stream connection. It then sends the names of all the recipients to which the mail is to be sent at (or via) the server host. The names are sent one to a line and terminated by a blank line (two carriage returns in a row). The Symbolics character set is used. A reply is immediately returned for each recipient. A recipient is typically just the name of a user, but it can be a user-atsign-host sequence or anything else acceptable to the mail system on the server machine. After sending the recipients, the sender sends the text of the message, terminated by an EOF. After the mail has been

successfully swallowed, a reply is sent. After the sender of mail has read the reply, both sides close the connection.

In the MAIL protocol, a reply is a signal from the server to the user (or sender) indicating success or failure. The first character of a reply is a plus sign for success, a minus sign for permanent failure (for example, no such user exists), or a percent sign for temporary failure (for example, unable to receive message because disk is full). The rest of a reply is a human-readable character string explaining the situation, followed by a carriage return.

The message text transmitted through the mail protocol normally contains a header formatted in the Arpanet standard fashion.

### 15.5.6 Chaosnet Send Protocol

The SEND protocol is used to transmit an interactive message (requiring immediate attention) between users. The sender connects to contact name SEND at the machine to which the recipient is logged in. The remainder of the RFC packet contains the name of the person being sent to. A stream connection is opened and the message is transmitted, followed by an EOF. Both sides close after following the end-of-data protocol. (See the section "End-of-data: Chaosnet Software Protocol", page 124. ) The fact that the RFC was responded to affirmatively indicates that the recipient is in fact present and accepting messages. The message text should begin with a suitable header, naming the user that sent the message. The standard for such headers, not currently adhered to by all hosts, is one line formatted as in the following example:

```
Moon@MIT-MC 6/15/81 02:20:17
```

Automatic reply to the sender can be implemented by searching for the first "@" and using the SEND protocol to the host following the "@" with the argument preceding it.

### 15.5.7 Chaosnet Name Protocol

The Name/Finger protocol of the Arpanet [FINGER] exists in identical form on the Chaosnet. Both Symbolics computers and timesharing machines support this protocol and provide a display of the user(s) currently logged in to them.

The contact name is NAME which can be followed by a space and a string of arguments like the "command line" of the Arpanet protocol. A stream connection is established and the "finger" display is output in Lisp Machine character set, followed by an EOF.

Symbolics computers also support the FINGER protocol, a simple-transaction version of the NAME protocol. An RFC with contact name FINGER is transmitted and the response is an ANS containing the following items of information separated by carriage returns: the logged-in user ID, the location of the terminal, the idle time in

minutes or hours-colon-minutes, the user's full name, and the user's group affiliation.

### 15.5.8 Chaosnet Time Protocol

The Time protocol of the Arpanet [TIME] exists on Chaosnet as a simple transaction. An RFC to contact name TIME evokes an ANS containing the number of seconds since midnight Greenwich Mean Time, Jan 1, 1900 as a 32-bit number in four 8-bit bytes, least-significant byte first. Some computers—Symbolics computers, for example—which don't have hardware calendar-clocks use this protocol to find out the date and time when they first come up.

### 15.5.9 Chaosnet Arpanet Gateway Protocol

This protocol allows a Chaosnet host to access almost any service on the Arpanet. The gateway server runs on each ITS host that is connected to both networks. It creates an Arpanet connection and a Chaosnet connection and forwards data bytes from one to the other. It also provides for a one-way auxiliary connection, used for the data connection of the Arpanet File Transfer Protocol.

The RFC packet contains a contact name of ARPA, a space, the name of the Arpanet host to be connected to, optionally followed by a space and the contact-socket number in octal, which defaults to 1 if omitted. The Arpanet Initial Connection Protocol is used to establish a bi-directional 8-bit connection.

If a data packet with opcode 201 (octal) is received, an Arpanet INS signal will be transmitted. Any data bytes in this packet are transmitted normally.

If a data packet with opcode 210 (octal) is received, an auxiliary connection on each network is opened. The first eight data bytes are the Chaosnet contact name for the auxiliary connection; the user should send an RFC with this name to the server. The next four data bytes are the Arpanet socket number to be connected to, in the wrong order, most-significant byte first. The byte-size of the auxiliary connection is 8 bits.

The normal closing of an Arpanet connection corresponds to an EOF packet. Closing due to an error, such as Host Dead, corresponds to a CLS packet.

### 15.5.10 Chaosnet Host Table Protocol

The HOSTAB protocol may be used to access tables of host addresses on other networks, such as the Arpanet. Servers for this protocol currently exist for Tenex and TOPS-20.

The user connects to contact name HOSTAB, undertakes a number of transactions, then closes the connection. Each transaction is initiated by the user transmitting a host name followed by a carriage return. The server responds with information about that host, terminated with an EOF, and is then ready for another

transaction. The server's response consists of a number of attributes of the host. Each attribute consists of an identifying name, a space character, the value of the attribute, and a carriage return. Values may be strings (free of carriage returns and *not* surrounded by double-quotes) or octal numbers. Attribute names and most values are in uppercase. There can be more than one attribute with the same name; for example, a host may have more than one name or more than one network address.

The standard attribute names defined now are as follows.

ERROR The value is an error message. The only error one might expect to get is "no such host".

NAME The value is a name of the host. There may be more than one NAME attribute; the first one is always the official name, and any additional names are nicknames.

MACHINE-TYPE
The value is the type of machine, such as LISPM, PDP-10, and so on.

SYSTEM-TYPE
The value is the type of software running on the machine, such as LISPM, ITS, and so on.

ARPA The value is an address of the host on the Arpanet, in the form *host/imp*. The two numbers are decimal.

CHAOS The value is an address of the host on Chaosnet, as an octal number.

DIAL The value is an address of the host on Dialnet, as a telephone number.

LCS The value is an address of the host on the LCSnet, as two octal numbers separated by a slash.

SU The value is an address of the host on the SUnet, in the form *net#host*. The two numbers are octal.

### 15.5.11 Chaosnet Dover Printer Protocol

A press file may be sent to the Dover printer by connecting to contact name **DOVER** at host **AI-CHAOS-11**. This host provides a protocol translation service which translates from Chaosnet stream protocol to the **EFTP** protocol spoken by the Dover printer. Only one file at a time can be sent to the Dover, so an attempt to use this service may be refused by a CLS packet containing the string **"BUSY"**. Once the connection has been established, the press file is transmitted as a sequence of 8-bit bytes in data packets (opcode 200). It is necessary to provide packets rapidly enough to keep the Dover's program (Spruce) from timing out; a packet every five seconds suffices. Of course, packets are normally transmitted much more rapidly.

Once the file has been transmitted, an EOF packet must be sent. The transmitter must wait for that EOF to be acknowledged, send a second one, then close the

connection. The two EOFs are necessary to provide the proper connection-closing sequence for the **EFTP** protocol. Once the press file has been transmitted to the Dover in this way and stored on the Dover's local disk, it will be processed and prepared for printing, and then printed.

If an error message is returned by the Dover while the press file is being transmitted, it will be reported back through the Chaosnet as a LOS containing the text of the error message. Such errors are fairly common; the sender of the press file should be prepared to retry the operation a few times.

Most programs that send press files to the Dover first wait for the Dover to be idle, using the Foreign Protocol mechanism of Chaosnet to check the status of the Dover. This is optional, but is courteous to other users since it prevents printing from being held up while additional files are sent to the Dover and queued on its local disk.

It would be possible to send to a press file to the Dover using its **EFTP** protocol through the Foreign Protocol mechanism, rather than using the **AI-CHAOS-11** gateway service. This is not usually done because **EFTP**, which requires a handshake for every packet, tends to be very slow on a timesharing system.

## 15.6 Using Foreign Protocols in Chaosnet

Foreign protocols which are based on the idea of a bidirectional (or unidirectional) stream of 8-bit bytes can simply be adopted wholesale into Chaosnet, using a Chaosnet stream connection instead of whatever stream protocol the protocol was originally designed for. This was done with the Arpanet Telnet protocol, for example.

When using such protocols between a Chaosnet process and a process on a foreign network, a protocol-translating gateway stands at the boundary between the two networks and has a connection on both networks. Bytes received from one connection are transmitted out the other. If the protocol uses any features besides a simple stream of bytes, for instance special out-of-band signals, these are translated appropriately by the gateway. The connection is initially set up by the user end connecting explicitly to the protocol-translating gateway and demanding of it a certain service from a certain host on the other network; the gateway then opens the appropriate pair of connections. For an example: See the section "Chaosnet Arpanet Gateway Protocol", page 133.

However, there are many packet-oriented protocols in the world and sometimes it is desirable to access these protocols at the packet level rather than the connection level, and to transport the packets of these protocols through Chaosnet links without using a Chaosnet connection. For example, there are gateways attached to Chaosnet which provide connections to other networks that use Internet as their packet protocol. User processes in Chaosnet hosts may talk to these other networks in those networks' own protocols by using the foreign-protocol protocol of Chaosnet.

A foreign packet is transmitted through Chaosnet by storing it in the data field of an UNC packet. The foreign packet is regarded as being composed of 8-bit bytes. The source and destination addresses of the UNC packet are used in the usual fashion to control the delivery of the packet within Chaosnet. The packet number and acknowledgement fields of the packet header are not used for their normal purposes, since this packet is not associated with a Chaosnet stream connection. By convention, the acknowledgement field of the packet contains a protocol number. The number 100000 octal means Internet. Other numbers will be assigned as needed. The packet number field of the packet can be used for any purpose.

If a user process transmits an UNC packet through a Chaosnet channel which is in the *Closed* state, the channel goes into the *Foreign* state and the NCP assumes that the user is not talking normal Chaosnet protocol, but is using Chaosnet to transport packets of some other protocol. See the section "Chaosnet Connection States", page 127. The NCP fills in the source address and index in these packets, but believes whatever destination address and index are placed in the packet by the user. The packet number and acknowledgement fields of the UNC packets are not touched by the NCP. Any incoming UNC packets addressed to the user's index on this host will be given to the user, regardless of their source address/index; it is up to the user program to filter out any unwanted packets. The NCP should also provide a way for one user to receive any unclaimed incoming UNC packets, so that rendezvous subprotocols of foreign protocols may be simulated.

When a packet-translating gateway to a foreign network receives an UNC packet with the appropriate protocol number, it extracts the foreign packet from the data field and fires it into the foreign network. When it receives packets from the foreign network, it maps the destination address of the packet into a Chaosnet address and index in some suitable fashion, encapsulates the packet in an UNC, and launches it into Chaosnet.

In the case of Internet, only protocols built on the idea of ports can be straightforwardly supported without a table of connections in the gateway. The Internet address space includes the Chaosnet host address space as a subset but does not provide any address breakdown within a host unless ports are used. However, it appears that most protocols are built on a protocol that uses ports, such as the User Datagram Protocol [UDP] or the Transmission Control Protocol [TCP].

In the case of foreign protocols where the addressing structure is not identical to Chaosnet, a program must somehow know the Chaosnet address of a packet-translating gateway to the foreign network. By sending UNC packets to this gateway, a user program can initiate connections to processes on that other network without requiring the local NCP (nor any bridges involved in routing the packets) to know anything about the protocol the program is using. If the inter-network gateway translates rendezvous protocols appropriately, connections may be initiated in the reverse direction also--from a user process on the foreign network to a server for the foreign protocol that resides on a Chaosnet host.

The foreign-protocol protocol may also be used between two user processes on Chaosnet, with no foreign network involved, if they simply wish to speak a different protocol from Chaosnet. They are on their own for a rendezvous mechanism, however, unless they use a Chaosnet simple transaction for rendezvous or otherwise have some way of conveying their addresses and index numbers to each other.

When foreign packets are too large to fit in the data field of a Chaosnet packet (more than 488 bytes), the user program and the packet-translating gateway must agree on a technique for dividing packets into fragments and reassembling them, unless the foreign protocol itself provides for this, as Internet does. The packet-number field in an UNC packet is available for use by such a technique, since UNC packets are not normally numbered.

UNC packets not associated with a connection are useful for other things besides encapsulating foreign protocols. Any application which wants to use Chaosnet as simply a packet transmission medium, essentially the raw hardware, should use UNC packets so that its packets do not interfere with standard packets and so that the standard routing mechanisms may be used. For example, the M.I.T. Architecture Machine uses UNC packets to communicate with non-stream-oriented I/O devices such as graphic tablets. Here Chaosnet is being used as an I/O bus which may be attached to more than one computer. Numbers between 140000 and 177777 octal in the acknowledgement field of an UNC packet are reserved for such applications. Note that this number is not part of the protocol; it is simply a hint about what a packet is being used for. Normally no program that is not specifically supposed to deal with such packets would ever receive one.

## 15.7 Chaosnet Hardware Programming Information

This section describes the UNIBUS version of the Chaosnet interface, which attaches to PDP-11s and Symbolics computers. The interface contains one buffer which holds a received packet and a second buffer which holds a packet to be transmitted. Packets are moved between these buffers and the computer under program control. Direct memory access (DMA) is not used; the small gain in performance was not thought to be worth the extra hardware complexity. The usual performance penalty of programmed I/O is not incurred since the packet buffers can transfer data at the full speed of the computer and neither busy waiting nor multiple interrupts are required.

To transmit a packet, successive 16-bit words of the packet are written into the outgoing packet buffer. First the eight 16-bit words of the header should be written, then *exactly* the number of 16-bit data words implied by the byte count in the header. If the byte count is odd, the last 16-bit word will contain the last byte in its low half and a garbage padding byte in its high half. After writing the data words, the last 16-bit word to be written is the cable address of the destination of the packet, or 0 to broadcast it. The hardware is then told to initiate transmission. It

waits until the cable is not busy and this node's turn to transmit arrives, then shifts the packet out onto the cable. At the completion of transmission transmit-done is set and the computer is interrupted. If transmission is aborted by a collision, transmit-done and transmit-abort are set and the computer is interrupted. As the packet is written into the outgoing packet buffer, a 16-bit cyclic redundancy checksum is computed by the hardware. This checksum is transmitted with the packet and checked by the receiver.

To receive a packet, the clear-receiver bit is asserted by the program. The next packet on the cable which is addressed to this node, or is broadcast, will be stored into the incoming packet buffer. After the packet has been stored, the computer is interrupted. The packet buffer will then not be changed until the next clear-receiver operation is performed, giving the computer a chance to read out the packet. If a packet appears on the cable addressed to this node while the incoming packet buffer is busy, a collision is simulated so as to abort the transmission. As a packet is stored into the incoming packet buffer, the 16-bit cyclic redundancy checksum is checked, and it is checked again as the packet is read out of the packet buffer. This provides full checking for errors in the network and in the packet buffers.

The standard interrupt-vector address for the Chaosnet interface is 270. The standard interrupt priority level is 5. The standard UNIBUS address is 764140. These are the device registers:

764140 *Command/Status Register*

This register contains a number of bits, in the usual PDP-11 style. All read/write bits are initialized to zero on power-up. Identified by their masks, these are:

" 1" Timer Interrupt Enable (read/write). Enables interrupts from the interval timer present in some versions of the interface (not described here).

" 2" Loop Back (read/write). If this bit is 1, the cable and transceiver are not used and the interface is looped back to itself. This is for maintenance.

" 4" Spy (read/write). If this bit is 1, the interface will receive all packets regardless of their destination. This is for maintenance and network monitoring.

" 10" Clear Receiver (write only). Writing a 1 into this bit clears Receive Done and enables the receiver to receive another packet.

" 20" Receive Interrupt Enable (read/write). If Receive Done and Receive Interrupt Enable are both 1, the computer is interrupted.

" 40" Transmit Interrupt Enable (read/write). If Transmit Done and Transmit Interrupt Enable are both 1, the computer is interrupted.

" 100" Transmit Abort (read only). This bit is 1 if the last transmission was aborted, by a collision or because the receiver was busy.

" 200" Transmit Done (read only). This bit is set to 1 when a transmission is completed or aborted, and cleared to 0 when a word is written into the outgoing packet buffer.

" 400" Clear Transmitter (write only). Writing a 1 into this bit stops the transmitter and sets Transmit Done. This is for maintenance.

" 17000"
Lost Count (read only). These 4 bits contain a count of the number of packets which would have been received if the incoming packet buffer had not been busy. Setting Clear Receiver resets the lost count to 0.

" 20000"
Reset (write only). Writing a 1 into this bit completely resets the interface, just as at power up and UNIBUS Initialize.

" 40000"
CRC Error (read only). If this bit is 1 the receiver's cyclic redundancy checksum indicates an error. This bit is only valid at two times: when the incoming packet buffer contains a fresh packet, and when the packet has been completely read out of the packet buffer.

"100000"
Receive Done (read only). A 1 in this bit indicates that the incoming packet buffer contains a packet.

764142 *My Address* (read)
Reading this location returns the network address of this interface (which is contained in a set of DIP switches on the board).

764142 *Write Buffer* (write)
Writing this location writes a word into the outgoing packet buffer. The last word written is the destination address.

764144 *Read Buffer* (read only)
Reading this location reads a word from the incoming packet buffer. The last three words read are the destination address, the source address, and the checksum.

764146 *Bit Count* (read only)
This location contains the number of bits in the incoming packet buffer, minus one. After the whole packet has been read out, it will contain 7777 (a 12-bit minus-one).

764152 *Start Transmission* (read only)
Reading this location initiates transmission of the packet in the outgoing packet buffer. The value read is the network address of this interface. This method for starting transmission may seem strange, but it makes it easier for the hardware to get the source address into the packet.

## 15.8 Chaosnet Lisp Machine Implementation

Lisp Machine Chaosnet support consists of a set of Lisp functions and data-structure definitions in the **chaos:** package. There are three important data structures. A **conn** represents a connection. A **pkt** represents a packet. A **stream** is a standard I/O stream which transmits to and receives from a connection. The details of these data structures are described later.

There are two processes which belong to the Chaosnet NCP. The receiver process looks at packets as they arrive from the network. Control packets are processed immediately. Data packets are put on the input packet queue of the connection to which they are directed. The background process wakes up periodically to do retransmission, probing, and certain "background tasks" such as starting up a server when an RFC arrives and processing "connection interrupts."

### 15.8.1 Opening and Closing Connections: Chaosnet Lisp Machine Implementation

#### 15.8.1.1 User-side: Opening and Closing Connections: Chaosnet Lisp Machine Implementation

**chaos:connect** *host contact-name* &optional *window-size timeout* *Function*
Opens a stream connection, and returns a **conn** if it succeeds or a string giving the reason for failure. *host* may be a number or the name of a known host. *contact-name* is a string containing the contact name and any additional arguments to go in the RFC packet. If *window-size* is not specified it defaults to 13. If *timeout* is not specified it defaults to 600 (ten seconds).

**chaos:simple** *host contact-name* &optional *timeout* *Function*
Taking arguments similar to those of **chaos:connect**, this performs the user side of a simple-transaction. The returned value is either an ANS packet or a string containing a failure message. The ANS packet should be disposed of (using **chaos:return-pkt**) when you are done with it.

**chaos:remove-conn** *conn* *Function*
Makes *conn* null and void. It becomes inactive, all its buffered packets are freed, and the corresponding Chaosnet connection (if any) goes away.

**chaos:close-conn** *conn* &optional *reason* *Function*
Closes and removes the connection. If it is open, a CLS packet is sent containing the string *reason*. Don't use this to reject RFCs; use **chaos:reject** for that.

**chaos:open-foreign-connection** *host index* &optional *pkt-allocation distinguished-port* *Function*
Creates a **conn** which may be used to transmit and receive foreign protocols encapsulated in UNC packets. *host* and *index* are the destination address for

packets sent with **chaos:send-unc-pkt**. *pkt-allocation* is the "window size", that is, the maximum number of input packets which may be buffered. It defaults to 10. If *distinguished-port* is supplied, the local index is set to it. This is necessary for protocols which define the meanings of particular index numbers.

**chaos:host-up** *host* &optional *timeout* *Function*

Asks a host whether or not it is up (alive, functional, responding). If it is up, this function returns **t**; if not, it returns two values: **nil**, and the error that occurred (usually "Host not responding."). *host* can be a host object or the name of a host; *timeout* is in 60ths of a second and defaults to three seconds. If the host does not respond after this much time, it is assumed to be down.

Note that if this function returns **nil**, it is possible that the host is up but is not connected to the Chaosnet. This function tests whether the Symbolics computer is capable of communicating with the host over the Chaosnet.

**chaos:notify-local-lispms** &optional (*message* *Function*
**(zwei:qsend-get-message "all lisp machines")**)

Sends *message* to all Lisp Machines at your site based upon information it gets from the namespace database about the Lisp Machines at the local site. *message* should be a string; if it is not provided, the function prompts for a message. Each recipient receives the message as a notification, rather than as an interactive message.

**chaos:notify** *host* &optional *message* *Function*

Sends a message to the specified host. *host* should be a host (the host name, as a string, or a host object). *message* is a string; if it is not provided, the function prompts for a message. The recipient receives the message as a notification, rather than as an interactive message.

**net:finger-location** *Variable*

This variable sets the location reported by the finger functions. Its value should be a string to print as the location part of a finger display. When this variable is **nil**, (the default), the system uses the value **si:local-finger-location**, which is set automatically by remote file servers. When the variable has a string value, it overrides the value in **si:local-finger-location**.

**net:finger-local-lispms** *Function*

Displays a list of who is using each of the Symbolics computers at the current site. (It uses **si:machine-location-alist**.) This function replaces the function **chaos:finger-all-lms** from previous releases.

**net:finger-all-lispms** *Function*

Displays a list of who is using each of the Symbolics computers in the host table. (It uses **si:host-alist**.)

### 15.8.1.2 Server-side: Opening and Closing Connections: Chaosnet Lisp Machine Implementation

**chaos:listen** *contact-name* &optional *window-size wait-for-rfc* *Function*

Waits for an RFC for the specified contact name to arrive, then returns a **conn** which will be in the *RFC Received* state. If *window-size* is not specified it defaults to 13. If *wait-for-rfc* is specified as **nil** (it defaults to **t**) then the **conn** will be returned immediately without waiting for an RFC to arrive.

**chaos:server-alist** *Variable*

Contains an entry for each server which always exists. When an RFC arrives for one of these servers, the specified form is evaluated in the background process; typically it creates a process which will then do a **chaos:listen**. Use the **add-initialization** function to add entries to this list.

**chaos:accept** *conn* *Function*

*conn* must be in the *RFC Received* state. An OPN packet will be transmitted and *conn* will enter the *Open* state. If the RFC packet has not already been read with **chaos:get-next-pkt**, it is discarded. You should read it before accepting if it contains arguments in addition to the contact name.

**chaos:reject** *conn reason* *Function*

*conn* must be in the *RFC Received* state. A CLS packet containing the string *reason* will be sent and *conn* will be removed.

**chaos:answer-string** *conn string* *Function*

*conn* must be in the *RFC Received* state. An ANS packet containing the string *string* will be sent and *conn* will be removed.

**chaos:answer** *conn pkt* *Function*

*conn* must be in the *RFC Received* state. *pkt* is transmitted as an ANS packet and *conn* is removed. Use this function when the answer is some binary data rather than a text string.

**chaos:fast-answer-string** *contact-name string* *Function*

If a pending RFC exists to *contact-name*, an ANS containing *string* is sent in response to it and **t** is returned. Otherwise **nil** is returned. This function involves the minimum possible overhead. No **conn** is created.

### 15.8.2 Connection States: Chaosnet Lisp Machine Implementation

**chaos:state** *conn* *Function*

Returns the current state of the connection, as one of the following symbols:

**chaos:inactive-state**
A **conn** which does not correspond to any Chaosnet connection.

**chaos:open-state**
An open connection.

**chaos:rfc-sent-state**
An RFC has been transmitted and no response has yet been received.

**chaos:answered-state**
An ANS has been received.

**chaos:cls-received-state**
A CLS has been received.

**chaos:los-received-state**
A LOS has been received.

**chaos:host-down-state**
The connection is in the *Incomplete Transmission* state; communications with the foreign host have broken down.

**chaos:listening-state**
A LSN has been "transmitted" and the connection is awaiting an RFC.

**chaos:rfc-received-state**
An RFC has been received while listening and has not yet been responded to.

**chaos:foreign-state**
The connection is being used with a foreign protocol encapsulated in UNC packets.

**chaos:wait** *conn state timeout* &optional *whostate* *Function*

Waits until the state of *conn* is not the symbol *state*, or until *timeout* 60ths of a second have elapsed. If the timeout occurs, **nil** is returned; otherwise **t** is returned. *whostate* is the process state to put in the status line; it defaults to **"net wait"**.

### 15.8.3 Stream I/O: Chaosnet Lisp Machine Implementation

**chaos:make-stream** *connection* &key *Function*
(*direction* **':bidirectional**) (*characters* **t**) (*ascii-translation* **nil**) (*accept-p* **t**)

Creates a bidirectional stream that accesses *connection*, which should be open as a stream connection, as 8-bit bytes. In addition to the usual I/O operations, the following special operations are supported:

**:force-output** Any buffered output is transmitted. Normally output is accumulated until a full packet's worth of bytes are available, so that maximum-size packets are transmitted.

**:finish** Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns **t**; if the connection goes into a bad state, returns **nil**.

**:eof** Forces out any buffered output, sends an EOF packet, and does a **:finish**.

**:clear-eof** Allows you to read past an EOF packet on input. Each **:tyi** will return **nil** or signal the specified eof error until a **:clear-eof** is done.

**:close** Behaves like the **:eof** message if not given an **abort-p** argument. The connection is also freed, so this need not be done manually.

Keyword arguments are:

**:direction** **:input**, **:output**, or **:bidirectional**. The default is **:bidirectional**.

**:characters** Boolean. The default is **t**. If not **nil**, character rather than binary data are to be sent.

**:ascii-translation** If not **nil**, characters are translated from ASCII to the Symbolics internal character set on input, and to ASCII on output. The default is **nil**.

**:accept-p** When not **nil** and the connection is in *RFC Received* state, accepts the connection. The default is **t**.

### 15.8.4 Packet I/O: Chaosnet Lisp Machine Implementation

Input and output on a Chaosnet connection can be done at the whole-packet level, using the functions in this section. A packet is represented by a **pkt** data structure. Allocation of **pkts** is controlled by the system; each **pkt** that it gives you must be given back. There are functions to convert between **pkts** and strings. A **pkt** is an **art-16b** array containing the packet header and data; the **chaos:first-data-word-in-pkt**'th element of the array is the first 16-bit data word. The leader of a **pkt** contains a number of fields used by the system.

**chaos:pkt-opcode** *pkt* *Function*

Accessor for the opcode field of *pkt*'s header. For each standard opcode a symbol exists in the **chaos:** package, consisting of the standard 3-letter code and a suffix of "**-op**", **chaos:rfc-op** for example. The value of the symbol is the numeric opcode.

**chaos:pkt-nbytes** *pkt* *Function*
Accessor for the number-of-data-bytes field of *pkt*'s header.

**chaos:pkt-string** *pkt* *Function*
An indirect array which is the data field of *pkt* as a string of 8-bit bytes. The length of this string is equal to **(chaos:pkt-nbytes** *pkt***)**.

**chaos:set-pkt-string** *pkt* &rest *strings* *Function*
Copies the *strings* into the data field of *pkt*, concatenating them, and sets **(chaos:pkt-nbytes** *pkt***)** accordingly.

**chaos:get-pkt** *Function*
Allocates a **pkt** for use by the user.

**chaos:return-pkt** *pkt* *Function*
Deallocates a **pkt**.

**chaos:send-pkt** *conn pkt* &optional *(opcode* **chaos:dat-op***)* *Function*
Transmits *pkt* on *conn*. *pkt* should have been allocated with **chaos:get-pkt** and then had its data field and n-bytes filled in. *opcode* must be a data opcode (200 or more) or EOF. An error is signalled, with condition **chaos:not-open-state**, if *conn* is not open. **chaos:send-pkt** automatically returns the packet via **chaos:return-pkt**.

**chaos:send-string** *conn* &rest *strings* *Function*
Sends a data packet containing the concatenation of *strings* as its data.

**chaos:send-unc-pkt** *conn pkt* &optional *pkt-number ack-number* *Function*
Transmits *pkt*, an UNC packet, on *conn*. The opcode, packet number, and acknowledge number fields in the packet header are filled in (the latter two only if the optional arguments are supplied). **chaos:send-unc-pkt** does an implicit **chaos:return-pkt**, which returns the packet to the free pool at the appropriate time.

**chaos:may-transmit** *conn* *Function*
A predicate which returns **t** if there is any space in the window.

**chaos:finish-conn** *conn* &optional *(whostate* **"Net Finish"***)* *Function*
Waits until either all packets have been sent and acknowledged, or the connection ceases to be open. If successful, returns **t**; if the connection goes into a bad state, returns **nil**. *whostate* is the process state to display in the status line while waiting.

**chaos:conn-finished-p** *conn* *Function*
A predicate that returns something other than **nil** if all data that have been output have been received *and* acknowledged by the foreign side of the connection.

**chaos:get-next-pkt** *conn* &optional *(no-hang-p* **nil)** *Function*

Returns the next input packet from *conn*. When you are done with the packet you must give it back to the system with **chaos:return-pkt**. This can return an RFC, CLS, or ANS packet, in addition to data, UNC, or EOF. If *no-hang-p* is **t**, **nil** will be returned if there are no packets available or the connection is in a bad state. Otherwise an error will be signalled if the connection is in a bad state, with condition name **chaos:host-down**, **chaos:los-received-state**, or **chaos:read-on-closed-connection**. If no packets are available and *no-hang-p* is **nil**, **chaos:get-next-pkt** will wait for packets to come in or the state to change. The process state in the status line is "**NETI**".

**chaos:data-available** *conn* *Function*

A predicate which returns **t** if there any input packets available from *conn*.

### 15.8.5 Connection Interrupts: Chaosnet Lisp Machine Implementation

**chaos:interrupt-function** *conn* *Function*

This attribute of a **conn** is a function to be called in the background process when certain events occur on this connection. Normally this is **nil**, which means not to call any function, but you can use **setf** to store a function here. Since the function is called in the Chaosnet background process, it should not do any operations that might have to wait for the network, since that could permanently hang the background process.

The function's first argument is one of the following symbols, giving the reason for the "interrupt". The function's second argument is *conn*. Additional arguments may be present depending on the reason. The possible reasons are:

**:input** A packet has arrived for the connection when it had no input packets queued. It is now possible to do **chaos:get-next-pkt** without having to wait. There are no additional arguments.

**:output**
An acknowledgement has arrived for the connection and made space in the window when formerly it was full. Additional output packets may now be transmitted with **chaos:send-pkt** without having to wait. There are no additional arguments.

**:change-of-state**
The state of the connection has changed. The third argument to the function is the symbol for the new state.

**chaos:read-pkts** *conn* *Function*

Some interrupt functions will want to look at the queued input packets of a connection when they get a **:input** interrupt. **chaos:read-pkts** returns the

first packet available for reading. Successive packets can be found by following **chaos:pkt-link**.

**chaos:pkt-link** *pkt* *Function*

Lists of packets in the NCP are threaded together by storing each packet in the **chaos:pkt-link** of its predecessor. The list is terminated with **nil**.

### 15.8.6 Information and Control: Chaosnet Lisp Machine Implementation

**chaos:host-data** &optional *host* *Function*

*host* may be a number or a known host name, and defaults to the local host. Two values are returned. The first value is the host name and the second is the host number. If the host is a number not in the table, it is asked its name using the **STATUS** protocol; if no response is received the name "**Unknown**" is returned.

**hostat** &rest *hosts* *Function*

Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, all hosts on the Chaosnet are asked. Hosts can be specified by either name or octal number.

For each host, a line is displayed that either says that the host is not responding or gives metering information for the host's network attachments. If a host is not responding, probably it is down or there is no such host at that address. A Lisp Machine can fail to respond if it is looping inside **without-interrupts** or paging extremely heavily, such that it is simply unable to respond within a reasonable amount of time.

To abort the host status report produced by **hostat** or `FUNCTION H`, press `c-ABORT`.

**chaos:print-conn** *conn* &optional *(short* **t***)* *Function*

Prints everything the system knows about the connection. If *short* is **nil** it also prints everything the system knows about each queued input and output packet on the connection.

**chaos:print-pkt** *pkt* &optional *(short* **nil***)* *Function*

Prints everything the system knows about the packet, except its data field. If *short* is **t**, only the first line of the information is printed.

**chaos:print-all-pkts** *pkt* &optional *(short* **t***)* *Function*

Calls **chaos:print-pkt** on *pkt* and all packets on the threaded list emanating from it.

**chaos:status** *Function*

Prints the hardware status.

**neti:reset** *Function*

Resets the local networks. Disables and then resets the interfaces. After using **neti:reset** you must call **neti:enable** if you want to turn the network back on.

**chaos:assure-enabled** *Function*

Turns on the network if it is not already on. It is normally always on unless you call one of these functions.

**neti:enable** *Function*

Enables the local networks and interfaces.

**neti:disable** *Function*

Disables the local networks and interfaces. If you want to reset the local networks and interfaces and then turn them back on, you should call **neti:reset** and then **neti:enable**.

## 15.9 Chaosnet VAX/VMS Implementation

This describes the interface to Chaosnet through the routines in the "CHAOS.B32" BLISS-32 subroutine package. Definitions of standard values are in "NCPDEFS.R32". Though it is possible to interface to the NCP at the VAX/VMS I/O level, it is not recommended practice. All references to Chaosnet in this text are with respect to the subroutine package, and not VAX/VMS QIOs.

A Chaosnet connection is represented by a one longword "channel number", which has no direct relationship to a VAX/VMS channel number. However, for every Chaosnet channel currently allocated, there is an associated VAX/VMS channel maintained by the subroutine package.

All of the routines described below are declared "global".

### 15.9.1 Opening and Closing: Chaosnet VAX/VMS Implementation

**parse_host** *(host, ret-host-num)* *Function*

Parses the string pointed to by *host* (which points to a standard VAX/VMS string descriptor), and stores the resulting host number in the word pointed to by *ret-host-num*. Returns a status code.

**chaos_rfc** *(ret-chan, host, contact-name, wait-time)* *Function*

Opens a new Chaosnet channel and sends an RFC. *ret-chan* is a longword to receive the channel number. *host* is a string acceptable to **parse_host**. *contact-name* is a pointer to a string descriptor. *wait-time* is either zero, which means to wait indefinitely for a response to the RFC, or a pointer to a quadword block acceptable to the **$SETIMR** system service. A status code is returned, which will be **SS$_TIMEOUT** if the routine times out.

**chaos_lsn** *(ret-chan, contact-name, wait-time)* *Function*
Like **chaos_rfc**, but "sends" a LSN instead of an RFC. No host is specified.

**chaos_accept** *(chan, window, rfc-arg, ret-rfc-arg-size)* *Function*
Accepts an incoming RFC. The connection must be in RFC-received state. *window* is the window size. *rfc-arg* is an optional string descriptor which receives the argument to the RFC. *ret-rfc-arg-size* is also optional, and gets the argument's length.

**chaos_ans** *(chan, data, wait-time)* *Function*
Sends an ANS packet to the Chaosnet channel. *data* points to a string descriptor, *wait-time* is ignored. A status code is returned, and if an error occurs, the channel is deassigned.

**chaos_close** *(chan, reason)* *Function*
Closes the connection, and deassigns the channel. *reason* is a pointer to a string descriptor of a string to be included in the CLS packet.

**chaos_assign** *(ret-chan)* *Function*
Assigns a Chaosnet channel, and stores it in the longword pointed to by *ret-chan*. This routine allocates a VAX/VMS channel. A status code is returned.

**chaos_deassign** *(chan)* *Function*
Given a Chaosnet channel previously assigned by **chaos_assign**, deassigns it and the associated VAX/VMS channel.

### 15.9.2 Stream I/O: Chaosnet VAX/VMS Implementation

**chaos_in_char** *(chan, ret-char, timeout)* *Function*
Returns the next character from the channel in the longword pointed to by *ret-char*. Waits until a character is available or until *timeout*, whichever comes first. A status code is returned.

**chaos_out_char** *(chan, char)* *Function*
Outputs one character. Characters are buffered until a packet fills up or until the output is forced out by **chaos_force_out**. A status code is returned.

**chaos_sout** *(chan, string)* *Function*
Like repeated calls to **chaos_out_char**: sends string from string descriptor pointed to by *string*.

**chaos_force_out** *(chan)* *Function*
If doing serial output, and a partial packet is buffered, force it to be sent.

**chaos_finish** *(chan)* *Function*

Does a **chaos_force_out**, then waits for all packets to be acknowledged by the foreign end.

**chaos_eof** *(chan)* *Function*

Sends an EOF packet after forcing out any buffered output.

### 15.9.3 Packet I/O

**allocate_pkt** *(size, chan, ret-pkt)* *Function*

Allocates a packet suitable for **chaos_in_pkt** and **chaos_out_pkt**. The packet can hold up to *size* bytes of data; the number of bytes field in the packet's header is filled in from *size*. *ret-pkt* points to a longword to receive a pointer to the packet. A status code is returned.

**deallocate_pkt** *(pkt)* *Function*

Returns a previously allocated packet to the free pool. A packet may be reused, since the I/O routines do not deallocate them, as long as the I/O is being done synchronously. Returns a status code.

**chaos_out_pkt** *(chan, pkt, efn, astadr, astprm)* *Function*

Outputs *pkt* to *chan*, waiting if there is no window room available. *efn* is the event channel to use for waiting. *astadr* and *astprm* are as for VAX/VMS system services: an AST address and parameter, respectively, that get signalled when the packet is read by the NCP. **chaos_out_pkt** returns as soon as there is space in the window, without waiting for the NCP to finish transmitting the packet.

**chaos_in_pkt** *(chan, efn, pkt, astadr, astprm)* *Function*

Reads the next input packet, whatever opcode it may be, from the connection, waiting indefinitely if there are no input packets. *efn* is the event channel to wait on, and *astadr* and *astprm* are for an AST to be delivered when the read completes. **chaos_in_pkt** does not return until the read completes. A status code is returned.

### 15.9.4 Checking the State: Chaosnet VAX/VMS Implementation

**chaos_xmit_room** *(chan, wait)* *Function*

Returns **SS$_NORMAL** if there is room left in the transmit window. Returns an error if the connection went into a bad state. If *wait* is true, and there is no room left, then **chaos_xmit_room** waits until room is available. If there is no room left and *wait* is false, it returns **SS$_EXQUOTA**.

**chaos_state** *(chan)* *Function*

Updates the state of the Chaosnet channel via a request to the NCP. Returns a status code. To check the state of the connection, first call this routine then look at **chan_state** in the channel block.

**chaos_wait** *(chan, old-state, timeout)* *Function*

Waits until the channel goes out of the specified state or until timeout occurs. Timeout is either zero (no timeout) or a pointer to a quadword block acceptable to **$SETIMR**. A status code is returned.

**chaos_wait_til** *(chan, state, timeout)* *Function*

Waits until the channel goes into the specified state or until timeout occurs. Timeout is either zero (no timeout) or a pointer to a quadword block acceptable to **$SETIMR**. A status code is returned.

The channel number is used as an index into the global blockvector **channel**, defined in the "CHAOS.B32" file. Since BLISS-32 does not allow the field definitions to be global, they should be copied into any program that needs to look inside the channel blockvector. The most useful fields are

**chan_state**
One of the state codes defined below.

**chan_sta_txw**
The window size in the transmit direction.

**chan_sta_rxw**
The window size in the receive direction.

**chan_sta_txwa**
The number of packet slots available in the transmit window.

**chan_sta_rxav**
The number of input packets available.

The states are as follows:

**conn_st_closed (0)**
Connection closed by a CLS packet.

**conn_st_rfcrcv (1)**
RFC received by listening connection.

**conn_st_rfcsnt (2)**
RFC sent, no response yet.

**conn_st_open (3)**
Connection open.

**conn_st_los (4)**
Connection broken by a LOS packet.

**conn_st_incom (5)**
Incomplete transmission (no response from foreign host).

**conn_st_new (6)**
Connection newly allocated.

**conn_st_lsn (7)**
Listening for an incoming RFC.

**conn_st_full (%O'400')**
This bit is set when the transmit window is full. Usually, the remainder of the state will be **conn_st_open**.

## 15.10 Chaosnet UNIX Implementation

Chaosnet support on UNIX is implemented as a device driver in the operating system and a set of user and server programs. The code runs on VAX systems running the current Berkeley UNIX.

The Network Control Program is implemented entirely in the kernel as a device driver, and is thus accessed from user programs with the normal input/output system calls. Packets received from the network are processed at interrupt level. All other processing is done with system calls issued by user processes.

### 15.10.1 Header Files: Chaosnet UNIX Implementation

All header files relevant to the Chaosnet software are kept in the **chaos** subdirectory of the global header file directory. They are:

**<chaos/user.h>**
Normally the only useful header file for user programs on the network. This file contains **ioctl** command definitions, associated data structures and constants, and pathnames of special files needed to access the network.

**<chaos/contacts.h>**
The contact names to access network services (names to put in RFC packets).

**<chaos/dev.h>**
The bit fields, constants, and macros used to encode and decode the minor device numbers for the Chaosnet special files.

**<chaos/chaos.h>**
All definitions of data structures used in the kernel. Rarely used by any user program.

### 15.10.2 Special Files for Creating Connections: Chaosnet UNIX Implementation

There are several *special files* in the file system that provide ways of creating and accessing connections. Their names are defined in **<chaos/user.h>**.

CHRFCDEV
: Opening this file creates a connection to a remote host. Specify the host address as an additional pathname component following the file name. It should contain the ASCII digits that represent the Chaosnet address in decimal (soon to be octal). The rest of the pathname after the host address is taken as the contact name that will be sent in the RFC packet. For example, to open a Telnet connection to the host at address 234 use:

```
#include <chaos/user.h>
#include <chaos/contacts.h>
char pathbuf[100];
int fd;
sprintf(pathbuf,
        "%s/%d/%s", CHRFCDEV, 234, CHAOS_TELNET);
fd = open(pathbuf, 2);
```

To send a message to User at host address 567 use:

```
sprintf(pathbuf,
        "%s/%d/%s %s", CHRFCDEV, 567, CHAOS_SEND, User);
fd = open(pathbuf, 1);
```

Opening CHRFCDEV returns when the response to the RFC is received from the remote host or a fixed timeout, whichever happens first. Other timeouts may be implemented by the user program, with the alarm-system call. ANS packets are acceptable responses. The data in the ANS packet are readable, and are followed by end-of-file, as with a full connection or a normal file.

CHRFCADEV
: This device provides the same functions as CHRFCDEV except that it returns immediately after transmission of the RFC packet with the connection in the CSRFCSENT state. This allows the user program to have access to the contents of packets refusing the connection (CLS, LOS). See CHIOCSWAIT and CHIOCPREAD; See the section "**ioctl** System Call Commands: Chaosnet UNIX Implementation", page 155.

CHLISTDEV
: Opening this file creates a connection in the listening state. Express the contact name as the pathname component following the device name. For example, to listen for a Telnet connection use:

```
sprintf(pathbuf, "%s/%s", CHLISTDEV, CHAOS_TELNET);
fd = open(pathbuf, 2);
```

Use the CHIOCSWAIT **ioctl** to wait for a RFC to arrive, and CHIOCREJECT, CHIOCACCEPT, or CHIOCANSWER to respond. See the section "**ioctl** System Call Commands: Chaosnet UNIX Implementation", page 155.

CHURFCDEV

When this file (the *unmatched RFC server* device) is opened and read, it returns the contents of RFC packets that have no listener. Read calls on this connection just return RFC data. If another read on this file is done before the RFC is matched, it is discarded. This file may only be opened by one user at a time. Normally this file is opened by the system unmatched-RFC server process.

### 15.10.3 Stream-mode Connections: Chaosnet UNIX Implementation

Stream mode is the default when a Chaosnet device is opened. This mode makes the connection behave like a UNIX file, with the exception that **seek** system calls are disallowed and **read** calls will return any available data (rather than returning the full number of bytes requested). Thus, standard I/O library routines can easily be used to read and write on these connections. A normal UNIX end-of-file indication is returned when an EOF packet is received; it will continue to be returned until either the connection is closed or more data arrive on the connection. If the connection is closed before an EOF packet is received (because of the arrival of a CLS or LOS packet, or the occurrence of a connection timeout), an error is returned after all data and EOF packets are read.

If the file has been opened for writing (open mode 1 or 2), an EOF packet is sent and its acknowledgement awaited when the file is closed (unless the connection has already been closed).

In stream mode all nondata packets are discarded and data packet opcodes are all treated the same. **ioctl**s can be used to read nondata packets (RFC, CLS, LOS, and so on) and perform other network-specific functions.

The contents of ANS and UNC packets are read as data in the stream, just as if they had been data packets.

### 15.10.4 Record-mode Connections: Chaosnet UNIX Implementation

Record mode is set on a connection by issuing

```
ioctl(fd, CHIOCSMODE, CHRECORD);
```

It gives the user program access to packet opcodes and packet boundaries. No further awareness of network data structures is necessary. Read calls from the connection return all the data in a single packet; the first byte of the data is the opcode in the packet. The count of bytes transferred, including the opcode, is returned. Opcodes are defined in chaos/user.h. See the section "Header Files: Chaosnet UNIX Implementation", page 152.

RFC, ANS, CLS, LOS, EOF, UNC, FWD, and data packets are returned to the user. The specified buffer must be large enough to fit the entire packet, including the opcode byte, or an error is returned.

In write calls, the first byte of data must include the desired opcode; this byte must also be reflected in the byte count. The data to be written must not exceed the maximum packet size.

If a record-mode connection is closed in the OPEN state, a CLS packet is automatically sent. The CHIOCREJECT **ioctl** should be used to send a CLS packet containing a specific reason. See the section "**ioctl** System Call Commands: Chaosnet UNIX Implementation", page 155.

### 15.10.5 Tty-mode Connections: Chaosnet UNIX Implementation

TTY mode (via CHTTY) allows the connection to act exactly like a UNIX tty. This allows, for example, remote log in service with no extra process for the NVT. Unfortunately, none of the remote protocols (Telnet, Supdup) can work over a transparent connection that just acts like a terminal. This mode is currently unused.

### 15.10.6 Foreign-protocol-mode Connections: Chaosnet UNIX Implementation

This mode is used to transmit and receive foreign protocols encapsulated in UNC packets. It is currently unimplemented.

### 15.10.7 ioctl System Call Commands: Chaosnet UNIX Implementation

The following **ioctl** codes can be used on Chaosnet connections.

**CHIOCSMODE**
Sets the connection mode. Argument is CHSTREAM, CHRECORD, CHTTY, or CHFOREIGN.

**CHIOCSWAIT**
Waits until the connection state changes from the given state (in the third argument). Typically used for listeners waiting for an RFC:

```
ioctl(fd, CHIOCSWAIT, CSLISTEN);
```

or for an end user waiting for a response to an RFC:

```
ioctl(fd, CHIOCSWAIT, CSRFCSENT);
```

**CHIOCFLUSH**
In stream mode, sends out any data waiting for a full packet. This is done every half-second at clock level.

**CHIOCOWAIT**
Waits for all transmitted data (after doing a CHIOCFLUSH) to be acknowledged by the other end of the connection. If the argument is nonzero, an EOF packet is sent first; it must also be acknowledged.

**CHIOCGSTAT**
Gets the status of the connection. The argument is the address to which

the status structure (**struct chst** in chaos/user.h is returned. See the section "Header Files: Chaosnet UNIX Implementation", page 152. This is frequently used to ascertain the state of the connection after a CHIOCSWAIT call, or to find out the Chaosnet address of the other end.

**CHIOCANSWER**
When a connection is in the CSRFCRCVD state, this code causes data writes on the connection to be sent with an ANS packet. In stream mode, the packet is filled incrementally. In record mode, the first packet sent is made into an ANS. In every case, only one packet is sent and the connection is closed.

**CHIOCACCEPT**
When a connection is in the CSRFCRCVD state, this code causes an OPEN packet to be sent and the connection to be opened.

**CHIOCREJECT**
When the connection is in either the CSRFCRCVD or CSOPEN state, this code causes a CLS packet to be sent, thus closing the connection. The argument is the address of a null-terminated string, which is copied into the close packet.

**CHIOCPREAD**
Reads a packet from the received packet queue. Used in stream mode to read control packets that are otherwise ignored. Typically used to read RFC or CLS packets.

**CHIOCRSKIP**
Skips over the unmatched RFC at the head of the unmatched RFC queue and marks it to be only matched against a listen, not queued as an unmatched RFC. This is used by the unmatched RFC server to ignore RFCs it knows someone else might want.

**FIONREAD**
A normal UNIX **ioctl**, this code returns an integer at the address specified by the argument, which contains the number of bytes available to be read.

### 15.10.8 Signals: Chaosnet UNIX Implementation

All read, write, open, close, and **ioctl**s are interruptable, except when waiting for buffer allocation. On VAX UNIX, read and write calls are automatically restarted. All others currently return EINTR errors.

### 15.10.9 Software Installation: Chaosnet UNIX Implementation

Global header files are placed in a chaos subdirectory of the system header file directory (usually /usr/include) so that #include <chaos/foo.h> works.

The kernel code is found in two subdirectories of the kernel source directory, parallel to **sys**, **dev**, and **conf**:

**chncp** This directory contains the parts of the NCP which do not depend on the operating system. It also contains the actual Chaosnet interface drivers, which have some operating-system-dependent code. These drivers interface only to the Chaosnet code (except interrupt vectors) and thus are not usable as UNIX device drivers.

**chunix**
This directory contains the top-level device-driver interface from the system-call level (through **cdevsw**) to the NCP, and some system dependent utilities (for example, buffer allocation).

The NCP needs two entries in the character-device switch and one other small change in **conf.c**.

In the UNIX kernel proper, several small changes are required:

**nami.c**A four-line change is required to **nami** to allow Chaosnet special files to have additional pathname components after the one that matches the special file in the file system.

**fio.c pty.c mx2.c autoconf.c locore.s**
Several small bugs which never were encountered by other drivers need fixing.

In the normal (Berkeley) VAX configuration scheme, normal entries made in the configuration file are sufficient to cause all the right files to be included in the system, if the CHAOS option is included in the options line and the following line is specified:

```
pseudo-device chaos
```

The **files** file gets a few more lines.

For PDP-11 UNIX, the configuration system is much more primitive; therefore, some handwork is usually required to make the kernel correctly.

All user program sources can be put in **/usr/src/cmd/chaos**, **/usr/local/src/cmd/chaos**, and **/usr/src/local/cmd/chaos**. The **make** file contains variables for destinations of all programs. The default destination for user programs is **/usr/local**. Server programs are placed in **/usr/local/lib/chaos**. The unmatched RFC server (**chserver**) is placed in **/etc** and should be started in the **/etc/rc** file at boot time. It may be killed and restarted at any time.

## 15.11 Chaosnet References

The following documents are of some related interest. AIM is an AI Memo of the M.I.T. Artificial Intelligence Laboratory. RFC is a Request for Comments of the Arpanet Network Working Group. IEN is an Internet Experiment Note of the Arpanet Network Working Group.

[CPR] C. Ryland, TOPS-20 Chaosnet Manual, unpublished.

[FILE] Documented online on the file AI:LMDOC;CHFILE >.

[FINGER] K. Harrenstien, Name/Finger, RFC-742.

[RFC733] D. Crocker et al., Standard for the Format of Arpa Network Text Messages, RFC-733.

[SUPDUP] M. Crispin, Supdup Protocol, RFC-747, RFC-734.

[TCP] DOD Standard Transmission Control Protocol, IEN-129.

[TELNET] Telnet Protocol Specification, RFC-542.

[TIME] K. Harrenstien, Time Server, RFC-738.

[UDP] J. Postel, User Datagram Protocol, IEN-88.

[UNIBUS] PDP-11 Peripherals Handbook, Digital Equipment Corporation.

# 16. Chaosnet File Protocol

## 16.1 Introduction: Chaosnet File Protocol

The QFILE protocol is the Lisp Machine implementation of the Chaosnet File Protocol. It allows reading and writing of files, and other manipulations of host file systems, from remote hosts. It was originally designed for Chaosnet only, but is now being adapted for implementation over more general media. Although originally designed with Symbolics computers as the intended user ends, other user ends (CFTPs) can be written. Chaos File Transfer Protocols (CFTPs) have been written for all hosts for which servers have been written.

The QFILE protocol operates over several logical user-to-server data paths, or *connections*. Each connection is assumed to be capable of data transfer in both directions, although not necessarily full duplex (allowing truly simultaneous bidirectional transfer). Currently, the connections must meet the following requirements:

- They must be capable of transmitting discrete parcels of data, minimum or maximum length not specified, each such parcel integral (a host cannot transmit any of one without transmitting it all).

- Each parcel must be capable of carrying an *opcode*, an arbitrary 8-bit tag defined by the protocol.

The Chaosnet implementation of these parcels is as Chaosnet *packets*; for convenience, we use the term packet for these parcels.

The user seeking file service on a server host establishes and maintains contact with a *server process*, or *server*, on the server machine, over one such data path, the *control connection*. This contact, the *user-server dialogue*, is expected to last for a long time, perhaps hours.

Communication over the control connection is, with one exception, synchronous, and initiated by the user. By preparing and issuing *commands* to the server, the user instructs the server to perform file operations. Each command is sent as one packet over the control connection. The server responds, unconditionally, to each command, by sending a *response* packet. A response packet reports the success or failure of the requested operation, and might contain *return values*, information requested or required by the user as a result of this operation.

Part of the information in each command is a *transaction identifier*, or *TID*, generated by the user, uniquely identifying the command within the context of this user-server dialogue. A response packet contains, among other things, the TID for the command for which it is a response. The TID identifies the response as a

response to that command. The server is not required to issue responses in the same order as it received the corresponding commands.

When reading or writing files, actual data are sent over separate connections called *data connections*. Each data connection, being bidirectional, is divided into two *data connection halves*, or *data channels*. Each active data transfer requires a data channel. Transfers on each half of a data connection can be completely unrelated. At any time during the user-server dialogue, some (possibly empty) set of data connections between user and server exists. When data connections exist, some data channels might be allocated to active data transfers. Such channels are known as *active* channels. Data channels not active are called *free*. Both user and server must keep track of the active/free status of each data channel.

Data connections are established and disestablished at user request, by means of a control connection command. Hence, the control connection must be established before any data connections. Data connections are associated with a specific dialogue, and thus with a specific instance of a server, and thus, indirectly, with a specific control connection. Data channels are serially re-usable: they can be used for many data transfers, in sequence. The number of data connections per server instance is often limited by server operating system constraints, such as limits on the number of open network channels per job.

Each data channel has, on the server side, a logical process associated with it. A physical process may or may not be the implementation of choice. This process has several well-defined states. State transitions are provoked by receipt of special packets (*marks*), reaching ends of files, and reactions to commands received on the control connection. When a data transfer is active on a data channel, the process is in a loop; it is either reading bytes from the data channel and writing them to the file, or vice versa. Reactions to commands on the control connection must be able to interrupt this loop in a clean, well-defined, resumable way.

The protocol nomenclature is defined from the point of view of the user side. Hence, terms such as read, write, input, and output are from the user side's point of view, unless explicitly indicated otherwise.

## 16.2 Qfile File Transfer Philosophy

The user accesses file data by requesting the server to open a file, then transferring data, then requesting the server to close the file. Actually, there is no such thing as "closing a file". The result of opening a file, in any program on any system, is an *open stream*. It is this stream that one closes, not the file. One may open a file, but one can only close a stream.

### 16.2.1 Opening a File with Qfile

Opening an input file implicitly requests the server to start sending the entire content of the file over a data channel, as fast as the network medium and path allow. The user side expects an uninterrupted stream of bytes to appear immediately on its side of the data channel.

Similarly, opening an output file instructs the server to prepare to receive an uninterrupted stream of bytes over a data channel, and write them to a file until the user indicates that he has sent the entire content to be transmitted. The user side sends bytes over the data channel until it has sent what it considers to be the whole file, and then sends special indicators signalling the end. See the section "Ending the Qfile Transfer", page 161.

### 16.2.2 Transferring Data with Qfile

To effect a transfer, the user side selects a free data channel of the correct direction (input or output). If no free data channels are available, the user must negotiate with the server beforehand to establish a new data connection. The user then issues the OPEN command to the server. The OPEN command (like many other commands) contains a field, the *file handle*, by which the user identifies the selected data channel. The server, on receiving this command, opens the file, and if successful, *binds* the data channel to the resulting open stream. Binding the data channel to the open stream means that the server considers the data channel active, and is prepared to transfer data between the channel and the stream. The server replies to the user and considers the OPEN transaction complete. The user side, upon receipt of a successful response to the OPEN command, begins transferring data over the data channel (for an output opening); the server begins transferring input data (for an input opening). In either case, the logical process associated with the data channel at the server side begins transferring data between the open stream and the data channel until it reaches the end of data in the file.

It is an error, a protocol violation, if the server finds the channel designated by the user to be already bound (active) at OPEN time. Unfortunately, this can happen in all current user end implementations if the user interrupts the user-side file-protocol program at certain critical points.

### 16.2.3 Ending the Qfile Transfer

To end the transfer, the user side issues a CLOSE command. (In the input case, this is usually motivated by receipt of an *end-of-file* (EOF) over the data connection). On receiving this command, the server performs the following actions:

- Goes through a synchronization procedure to ensure that all data have been transferred. For details on the synchronization procedure: See the section "Close Qfile Command", page 181.

- Closes the open stream. On different operating systems, this may correspond to closing the file, setting the segment's bit count and terminating it, renaming the file from its temporary to its real name, and so forth.
- *Unbinds* the data channel, that is, declares it to not be in use.
- Replies to the user.

The user, on receipt of the successful response, marks the data channel as no longer active: it is now free for further use.

Opening a file always requires a preexisting data channel and uses that channel until the CLOSE.

The user can request the server to *close-abort* the open stream, instead of simply closing it. To close-abort a stream means to close it in such a way, if possible, that it is as if the file had never been opened. In the specific case of a file being created, it must appear as if the file had never been created. This may be more difficult to implement on certain operating systems than others, but tricks with temporary names and close-time renamings by the server can usually be used to implement close-abort in these cases. In the case of a file being appended to, close-abort means to forget the appended data.

The default and most common mode of Symbolics computer output is to create new files. Except for specialized applications, no software overwrites old files. "Output" openings are thus "create and write" openings. Opening a file for output is thus both the means and equivalent of creating a new file. There are OPEN options for overwriting old files: See the section "Qfile Open Options", page 172. However, most of the protocol has been designed with the assumption that writing and file creation are one and the same.

## 16.3 Qfile Character Set Translation

All numbers designating values of character codes are to be interpreted in octal.

The QFILE protocol was designed to provide access to ASCII-based file systems for Symbolics computers. Symbolics computers support 8-bit characters and have 256 characters in their character set. This results in difficulties when communicating with ASCII machines, which have 7-bit characters.

The server, on machines not using the Symbolics character set, is required to perform character translations for any CHARACTER (not BINARY) opening. Table 1 shows the translations between Lisp Machine characters and the standard ASCII representation, as used on the PDP-10 (where the sequence CRLF, 015 012 represents a new line). Some Symbolics characters expand to more than one ASCII character. Thus, for character files, when we speak of a given position in a file or

the length of a file, we must specify whether we are speaking in *Symbolics units* or *server units*, for the counting of characters is different.

This causes major problems in file position reckoning. Specifically, it is futile for the Symbolics computer (or other user side) to carefully monitor file position, counting characters, during output, when character translation is in effect. This is because the operating system interface for "position to point *x* in a file", which the server must use, operates in server units, but the Symbolics computer (or other user end) has counted in Symbolics units. The user end cannot try to second-guess the translation-counting process without losing host-independence. (Although the Symbolics mail reader, Zmail, does anyway, as certain types of PDP-10 mail files contain embedded encoded character counts that are measured in server units.) See the section "Filepos Qfile Command", page 185.

Table 1 contains the standard ASCII table (all values octal). The notation `x in <c1, c2>` means "for all character codes x such that c1 <= x <= c2." Hosts using other variations of ASCII, or other character sets, must translate accordingly.

Table 1. Translations Between Symbolics Characters and Standard ASCII

| *Symbolics character* | *ASCII character(s)* |
|---|---|
| x in <000, 007> | x |
| x in <010, 012> | 177 x |
| 013 | 013 |
| x in <014, 015> | 177 x |
| x in <016, 176> | x |
| 177 | 177 177 |
| x in <200, 207> | 177 <x - 200> |
| x in <210, 212> | <x - 200> |
| 213 | 177 013 |
| 214 | 014 |
| 215 | 015 012 |
| x in <216, 376> | 177 <x - 200> |
| 377 | no corresponding code |

Table 1 might seem confusing at first, but there are some general rules about it that should make it appear more sensible. First, Symbolics characters in the range <000, 177> are generally represented as themselves, and x in <200, 377> is generally represented as 177 followed by <x - 200>. That is, 177 is used to quote the second 200 Symbolics characters. It was deemed that 177 is more useful and common character than 377, so 177 177 means 177, and there is no way to describe 377 with ASCII characters. On the Symbolics computer, the formatting control characters appear offset up by 200. This explains why the preferred mode of expressing 210 (backspace) is 010, and 010 turns into 177 010. The same reasoning applies to 211 (Tab), 212 (Linefeed), 214 (Formfeed), and 215 (Newline).

More special care is needed for the Newline character, which is the mapping of the system-independent representation of "the start of a new line". Thus, for ASCII as used on many systems, Symbolics Newline (215) is equivalent to 015 012 (CRLF) in ASCII characters. When converting ASCII characters to Lisp machine characters, an 015 followed by an 012 therefore turns into a 215. A "stray CR", that is, an 015 *not* followed by an 012, therefore causes character-counting problems. To address this, a stray CR is arbitrarily translated into a single M (115).

Table 1 applies in the case of NORMAL translation, that is, the default character translation mode.

The other translation modes available are:

RAW — Performs no translation. ASCII characters are obtained by simply discarding the high order bit of Symbolics characters, and Symbolics characters supplied by an ASCII server are always in the range <000, 177>.

SUPER-IMAGE — Suppresses the use of Rubout for quoting. That is, each entry beginning with a 177 in the ASCII column of the translation table presented above has the 177 removed. The ASCII character 015 always maps to the Symbolics character 215, as in normal translation. Here is the SUPER-IMAGE mode table:

Table 2. Translations in SUPER-IMAGE Mode

| *Symbolics character* | *ASCII character(s)* |
|---|---|
| x in <000, 177> | x |
| x in <200, 214> | <x - 200> |
| 215 | 015 012 |
| x in <216, 376> | <x - 200> |
| 377 | no corresponding code |

In SUPER-IMAGE mode as well, stray CR is translated to Symbolics character M.

## 16.4 Qfile Command and Response Format

Commands and responses are indivisible, single parcels of data sent over the control connection. User end always sends commands to server and server sends responses to user.

## 16.5 Qfile Packet Opcodes

The parcel (packet) opcode associated with both commands and responses is **chaos:dat-op** (= 200 octal). The data content of both packet types are character strings in the Symbolics character set: See the section "Qfile Packet Data Contents", page 165.

Responses to commands, successful or not, come as data packets (that is, **chaos:dat-op**) on the control connection.

In normal usage, only packets with opcode **chaos:dat-op** are sent on the control connection. There are three other types of packet that can be sent over the control connection in unusual circumstances:

1. EOF
   From user to server. Opcode = **chaos:eof-op** (= 14 octal). EOF provides a clean way for the user to close out the dialogue: the server aborts any transfers in progress, disestablishes all data connections associated with this dialogue, and disestablishes the control connection. Since this is what the server should be doing if the control connection is broken (by the medium supporting it) in any case, sending the EOF packet is optional.

2. Notification
   From server to user. Opcode = **fs:%file-notification-opcode** (= 203 octal). The server can send notification to the user at any time. This is an unsolicited user-directed message by the server. The data content of the packet is a string to be printed at the user (where he is clearly identifiable) in real time. It generally warns of impending system shutdown or similar notifications.

3. Asynchronous mark
   From server to user. Opcode = **fs:%file-asynchronous-mark-opcode** (= 202 octal). Asynchronous mark is sent over data channels to report an error during active user-to-server data transfer. (There is no other obvious channel over which to communicate this error.) See the section "Qfile Errors and Asynchronous Marks", page 197.

## 16.6 Qfile Packet Data Contents

The contents of both command and response packets contain some number of *tokens* followed by some number of *string values*.

### 16.6.1 Qfile Tokens

There are always some tokens in each command or response. The tokens are fixed-format, protocol-specified strings, such as operation and option names, and protocol-specified identifiers, such as TIDs. Nonnegative decimal integers, with no trailing decimal point or other random punctuation, (*<num>*s), are valid tokens. All other tokens must be in uppercase. Tokens thus never contain spaces or funny characters that could cause parsing ambiguities. Tokens are separated by spaces; in most contexts, *exactly one space* separates tokens and multiple spaces imply the existence of *missing tokens.*

### 16.6.2 Qfile String Values

String values convey file pathnames, properties, error messages, and other string data whose content cannot be so syntactically restricted. Except in the case of error messages, string values are only constrained to not contain the Symbolics Newline (215 octal) character. (This does create certain problems and is a known deficiency). If there are string values, a Lisp Machine Newline (215 octal, henceforth <NL>) follows all the tokens, followed by the string values, themselves separated by <NL>. If there are no string values, there is no <NL>. When there are string arguments, the protocol requires a <NL> following the last argument. Servers are inconsistent about requiring the final <NL>, though.

### 16.6.3 Qfile Syntax

In the following syntactic descriptions:

- Brackets ([]) followed by an asterisk (*) represent optional groups repeated 0 to more times.
- Brackets ([]) not followed by an asterisk (*) represent an optional construct.
- Braces ({}) denote precedence grouping.
- <SP> is a Symbolics Space character (40 octal).
- <NL> is a Symbolics Newline character (215 octal).
- Actual spaces are for legibility only.

The general syntax of command and response packets:

```
token  [<SP> | {<SP> token}]*
[<NL> string-value [<NL> string-value]* <NL>]
```

The format of the common part of the token portions of command and response packets:

```
command:    TID <SP> [ FH ] <SP> CMD [ <SP> token-arg ]*
              [<NL> string-arg [<NL> string-arg]* <NL>]

response:   TID <SP> [ FH ] <SP> CMD [ <SP> token-result]*
              [<NL> string-result [<NL> string-result]* <NL>]
```

The tokens shown above specifying TID, FH, and CMD are all required in every command or response. Their interpretation and format:

TID — Transaction identifier — Generated by the user in the command packet and sent back by the server in the response packet. This field is required in every command or response packet. Via the TID, the user can tell which response is for which command. (Remember that the server is *not* constrained to reply in order.) The TID is limited to a maximum of 5 characters.

FH — File handle — Identifies a data channel or open file. Although not every command or response contains a file handle, this field is required, and its value (a null file handle) is an empty string to indicate omission. Thus, if the file handle is omitted, both the space that would precede it and the space that would follow it remain. Like TIDs, file handles are limited to a maximum of 5 characters.

Each data channel has a unique file handle associated with it. The user side defines the file handles for a given data connection at the time the data connection is established. The user and the server use these file handles to identify the data channels of that connection for the life of the dialogue. Thus, there are *input file handles* for data channels from user to server and *output file handles* for data channels from server to user. Data channels are thus said to have a *direction*, *polarity*, or *gender*.

There is a one-to-one mapping between data channels and files that the server has opened at the request of the user. Thus, one can think of input and output file handles as identifying both an open file stream and a data channel.

CMD — Command name — An arbitrary-length string identifying the required operation. These command names must be presented in uppercase, and are chosen from the fixed set described below. We reserve the right to extend this set arbitrarily. All characters are significant.

The args and results vary depending on the command and are included in the description of each command. The successful response to a particular command has:

- The same TID as the command.
- The same file handle (null or nonnull).
- The same command name.

An error response is identical, but has the command name ERROR in all cases. The values of success and error response are described in the appropriate command descriptions.

## 16.7 Qfile Marks and EOF Packets

The protocol designates three special types of packets:

| *Packet Type* | *Opcode* |
|---|---|
| EOF | **chaos:eof-op** (= 14 octal) |
| Synchronous mark | **fs:%file-synchronous-mark-opcode** (= 201 octal) |
| Asynchronous mark | **fs:%file-asynchronous-mark-opcode** (= 202 octal) |

EOFs on the control connection are rare (maximum one per): See the section "Qfile Packet Opcodes", page 165. EOFs are sent on data channels to mark the end of the data in the file when reading or writing. That is, when the user is writing a file and wishes to show that there is no more data, he sends an EOF packet on the relevant data channel to tell the server it has reached the end. This is necessary, over and above the CLOSE command, so that the server can know that all data have been read from the data channel before closing the file. Similarly, the server sends an EOF packet after the last byte of file data when reading. EOF packets are always empty (contain no data). The user side detects end of file on input by receipt of this packet.

Synchronous marks are sent on data channels to signal synchronization points, corresponding to control connection events, other than end of file. There is no data in the synchronous mark; its opcode conveys all its meaning. For example, when the server recieves a FILEPOS (set file position) command to an input file, it sends a synchronous mark over the data channel before actually setting the new file position and pumping the new data into the data channel. Since the user side has requested an unbounded amount of data, there is no way that the user side could tell where the old data (sent before the new file position was set) ended and the new data (sent after) began, save for the synchronous mark. The command descriptions describe the use of synchronous marks where appropriate: See the section "Qfile Command Descriptions", page 169.

Asynchronous marks are sent over the control connection to signal errors during active user-to-server data transfer. For details on asynchronous marks: See the section "Qfile Errors and Asynchronous Marks", page 197.

## 16.8 Qfile Command Descriptions

Conventions used in the descriptions:

- Byte sizes are expressed as decimal numbers. Any other number larger than one digit is explicitly stated to be octal or decimal.
- <none> means that there are no arguments of this type.
- <num> means a sequence of ASCII characters that are all digits (octal codes 060 through 071, that is, 0 through 9). Such numbers are always expressed in decimal. These numbers are unsigned but can have leading zeroes.

The descriptions of the commands include listings of each of the types of arguments and results. For a detailed description of the handling of <SP> and <NL> characters: See the section "Qfile String Values", page 166. We use lines and line breaks in the descriptions freely; a <NL> is *not* implied in the packet format when we continue a description on a second line.

Bear in mind that the end of a command or response is not marked by a special character. The medium supporting the packets is assumed to be able to convey the length of transmitted data.

### 16.8.1 Data-connection Qfile Command

```
DATA-CONNECTION
        token args = IFH OFH
        string args = <none>
        token results = <none>
        string results = <none>
```

This establishes a new data connection. The IFH and OFH fields are the input and output file handles to be associated with this data connection. The user side generates the file handles, which must be distinct from all other file handles of other data connections of this dialogue. The file handle (FH) field of the command must be omitted. The input file handle describes the server-to-user data channel of the connection to be established and the output file handle describes the user-to-server channel.

When the server receives a DATA-CONNECTION command, it attempts to contact the user host in some way derivable from the values of the file handles. The user side must be prepared to complete this new data connection. The server responds to

the DATA-CONNECTION command immediately, even though the new data connection might not yet be established. Of course, the user side cannot use the data connection until its establishment is complete.

The Chaosnet implementation of this transaction uses the OFH as a contact name. The user sends the DATA-CONNECTION command to the server, and then listens on the contact name equal to the OFH, while awaiting the response to the DATA-CONNECTION command. This requires a parallel wait, is difficult to get right, and is an acknowledged deficiency. The server attempts to contact to the name equal to the OFH received in the DATA-CONNECTION command and reports success via the response packet. This Chaosnet connection becomes the new data connection.

### 16.8.2 Undata-connection Qfile Command

```
UNDATA-CONNECTION
        token args = <none>
        string args = <none>
        token results = <none>
        string results = <none>
```

This command explicitly disestablishes a data connection from the user side. The user side has the option of disestablishing data connections at its discretion; the Symbolics computer user side disestablishes data connections that have not been used for long periods of time. There is no place in the protocol where disestablishment of data connections is required, other than at the end of the dialogue, where it is implicit.

The data connection to be disestablished is the one whose input or output file handle appears in the FH field of the UNDATA-CONNECTION command.

It is not permitted to explicitly disestablish a data connection either of whose channels is active. If the dialogue is terminated by breaking of the control connection, all file handles become meaningless, and the server must close all data connections known to it and close-abort all files opened on behalf of the user during the dialogue.

### 16.8.3 Open Qfile Command

```
OPEN
        token args = [option]
        string args = filename
        token results = creation-date length qfaslp characters
                        [author [byte-size [filepos]]]
        string results = truename
```

The author, byte-size, and filepos results are optional (they are optional tokens as well).

This command opens a file for reading, writing, or direct access at the server host.

That means, in general, asking the host file system to access the file and obtaining a file number, pointer, or other quantity for subsequent rapid access to the file.

The `filename` argument is the pathname of the file to be opened. It is in the full pathname syntax of the server host. It contains no host identifier of any kind. It is always fully defaulted, in the sense that it has a directory and file name (and file type if the operating system concerned supports file type). It might or might not reference a device, if that is appropriate. If the file system of the server supports version numbers, there is always an explicit version number, even if that number or other specification is "newest" or "oldest". For some purposes (for example, the OPEN option PROBE-DIRECTORY), only the directory specified by this pathname is utilized. See the section "Qfile Open Options", page 172.

The `FH` field of the OPEN command specifies a file handle that is used in all future commands (in this dialogue) to talk about the open stream that is created by opening the specified file. If an input file handle is given in the file handle field of the command, then the file is to be opened for reading. If an output file handle is given, then the file is to be opened for writing. In either case the file handle serves to indicate not only the direction of transfer, but also the data channel to be used for the transfer. A file handle must be supplied, unless a PROBE (a file status request) is being done. In the case of PROBE, a null (omitted) file handle is given in the `FH` field. For a description on the OPEN option PROBE: See the section "Qfile Open Options", page 172.

The `option` field of the OPEN command consists of a number of options, which are optional tokens. The options encode all the conceptual arguments to the file system for the OPEN operation except the pathname of the file. Some options also influence server treatment of the result of the opening. The options are a mixture of singleton keywords and keywords followed by single token values. Some options are always singleton keywords; the remainder of the options are always followed by values. It is impossible to parse the OPEN command unless the server knows which are which. Other than for preserving keyword/value pairs, the order of options is not significant. For a description of all the supported OPEN options: See the section "Qfile Open Options", page 172.

The token results reflect information about the file opened, when opening is successful. In the case of a PROBE opening, this information is returned, when the file exists and is accessible, even though the file is not opened. For detail on the format and semantics of these values: See the section "Qfile Open Response Result Values", page 178.

The single string results of a successful opening is a `truename` of the file actually opened. This value is discussed more fully: See the section "Qfile Open Response Result Values", page 178.

#### 16.8.3.1 Qfile Open Options

Here are the known open options. All are assumed to be singleton keywords unless otherwise noted.

**READ** Specifies that the file is to be opened for input (server-to-user transfer). The READ, WRITE, and PROBE options are mutually exclusive. One and only one must always be supplied. This option is redundant, as the direction and presence of the supplied file handle is sufficient to select READ, WRITE or PROBE. The server should check for consistency anyway.

**WRITE** Specifies that the file is to be opened for output (user-to-server transfer). The READ, WRITE, and PROBE options are mutually exclusive. One and only one must always be supplied. This option is redundant, as the direction and presence of the supplied file handle is sufficient to select READ, WRITE or PROBE. The server should check for consistency anyway.

**PROBE** Specifies that the file is not to be opened at all, but only checked for existence. If the file does not exist or is not accessible, the error indications and actions are identical to those that would be given by READ. If the file does exist, the successful response packet contains the same information as it would have if the file had been opened for READ. No (a null) file handle is to be supplied. The READ, WRITE, and PROBE options are mutually exclusive. One and only one must always be supplied. This option is redundant, as the direction and presence of the supplied file handle is sufficient to select READ, WRITE or PROBE. The server should check for consistency anyway.

**CHARACTER** Specifies that we will be transferring character data as opposed to binary. This affects the mode in which the server opens a file, as well as whether or not character set translations are performed. CHARACTER is the default, so it need never be specified. The server performs character set translation between its native character set and the Symbolics character set. The data are transferred over the data connection one character per eight-bit byte. The check (described in the DEFAULT OPEN option) for Symbolics object files is not performed. For the effect of CHARACTER on PROBE openings: See the section "Qfile Open Response Result Values", page 178.

**BINARY** Specifies that we will be transferring binary data. This affects the mode in which the server opens the file, as well as requesting it not to attempt character set translation. The user side supplies the byte size via the BYTE-SIZE option; if not supplied, the default byte size is 16 bits. No matter what the byte size, the server transfers each byte of the file as two eight-bit bytes, low-

order first. (The Chaosnet implementation also stipulates that no 16-bit data byte shall be split across two Chaos packets). The check for Symbolics object files is not performed. Specification of BINARY for PROBE openings is the nominal default: See the section "Qfile Open Response Result Values", page 178.

DEFAULT — Specifies that the server is to figure out whether to transfer binary or character data. (The BINARY or CHARACTER options explicitly specify this information.) This is only meaningful for input openings: it is an error for output or PROBE openings. For file systems that maintain the innate binary or character nature of a file, the server simply asks the file system which case is in force; the server tells the file system. (If the file was created by the file server, it establishes this information from the BINARY or DEFAULT options). For file systems that do not maintain this information, the server is required to perform a heuristic check for Symbolics object files upon the first two 16-bit bytes of the file. If the file is thus determined to be a Symbolics object file, the server performs a BINARY opening; otherwise, it performs a CHARACTER opening.

The details of the check are as follows: if the first two 16-bit bytes are the octal numbers 143150 and 071660 respectively, the file is recognized as a Symbolics object file. Alternatively, if the first 16-bit byte is the octal number 170023 and the second 16-bit byte is any number between 0 and 77 octal, inclusive, it is also recognized as a Symbolics object file. In any other case, it is not.

BYTE-SIZE — Must be followed by a decimal <num> between 1 and 16, inclusive. BYTE-SIZE can only be supplied for BINARY openings and can be ignored for PROBE openings.

If a BINARY opening is requested and BYTE-SIZE is not supplied, the assumed value is 16 for output openings. For input BINARY openings, the default is the host file system's stored conception of the file's byte size (for those hosts that natively support byte size). This information is of great value to the Symbolics computer file copier when it does not know about the particular file type involved. For file systems that do not natively support byte size, the default on BINARY input is 16.

For file systems that support byte size, the server should supply this number to the appropriate operating system interface that performs the semantics of opening the file. For other operating systems, a file written with a given byte size *must* produce the same bytes in the same order when read with that byte size. In this case, the server can choose any packing scheme that complies with this rule.

Operating systems that do not support byte size are only required that binary files written from user ends of the current protocol can be read back correctly. However, the server can increase the utility of the Symbolics computer at a customer site by choosing packing schemes that allow all bits of the server host's word to be accessed and concur with other packing schemes used by native host software.

For example, the Multics server packs:

| *Byte Size* | *Packing Scheme* |
|---|---|
| 7, 8, or 9 bits | four per 36-bit word |
| 10, 11, or 12 bits | three per 36-bit word |
| 13, 14, 15, or 16 bits | two per 36-bit word |

In the 9-bit packing mode, native Multics character-oriented software can access each logical byte sequentially. In 18-bit packing mode, each Symbolics byte is in a halfword, easily accessible and visible in an octal representation. To achieve maximum data transfer rate and access all bits of a Multics word, a byte size of 12 can be specified.

DELETED — Specifies that "deleted" files are to be treated as though they were not "deleted". This is only meaningful for operating systems that support "soft deletion", or undeletion of files. Other operating systems *must* ignore this option. Normally, deleted files are not visible to the OPEN operation; this option makes them visible. For output openings, DELETED is meaningless and an error if supplied; instead, use the IF-EXISTS option.

RAW — Specifies that character set translation is not to be performed, but characters are to be transferred intact, without inspection. This option is only meaningful for CHARACTER openings; it is an error otherwise, and an error for PROBE openings. Servers operating natively in the Symbolics character set (for example, Symbolics computers) can ignore this option.

SUPER — Specifies that Rubout quoting is not to be performed. This operation is only meaningful in CHARACTER mode; it is an error otherwise or in PROBE mode. This reads or writes character files where ASCII Rubout characters are a significant part of the file content (such as ITS XGP files), not where they are an escape for this protocol. Nevertheless, this is different than RAW, for other translations are still to be performed: See the section "Qfile Character Set Translation", page 162.

The PDP-10 (all operating systems) server considers SUPER-IMAGE to be the official name of this option, but two counteracting bugs, namely the PDP-10 server's comparing of only five characters and the Symbolics computer's belief that SUPER is the name of this option, mask this fact.

It is felt that the name of this option should be neither of the above, but NO-ESCAPE, NO-RUBOUT-ESCAPE, NO-ESC-177, or something similarly descriptive.

TEMPORARY Used by the TOPS-20 server only: says to use GJ%TMP in the GTJFN. This is useful mainly when writing files, and indicates that the foreign operating system is to treat the file as temporary. See TOPS-20 documentation for more about the implications of this option. Other servers can ignore it.

This option is meaningless and an error for input or PROBE openings.

PRESERVE-DATES
Specifies that the server is to make an attempt to prevent the operating system from updating the "reference date" or "date-time used" of the file. This is only meaningful and legal for input openings. The Symbolics computer operating system invokes this option for operations such as View File in the editor, where it wishes to assert that nobody "read" the file, but just "looked at it". Servers on operating systems that either do not support reference dates or do not support users fraudulating or suppressing update of the reference dates can ignore this option.

INHIBIT-LINKS Only meaningful and permitted for PROBE openings. This option controls what happens when an attempt is made to open, for PROBE, something that turns out to be a link. When an attempt is made to read or write a file that turns out to be a link, normally, the `target` file of that link is read or written instead: that is the contract of links. Normally, PROBE openings act the same way, describing the target file of the link. If, however, the INHIBIT-LINKS option is specified, information about the link itself is returned in the OPEN response, not information about the target. If the file being opened for PROBE turns out not to be a link, the INHIBIT-LINKS option can be ignored. It must be completely ignored by servers for operating systems that do not support links.

PROBE-DIRECTORY
Only meaningful and permitted for PROBE openings. When PROBE-DIRECTORY is specified, information is sought about the directory designated by the file pathname. Every file pathname designates some directory: normally, it is the directory containing the file to be opened. In this case, it is the directory about which

information is sought. The file name and type of the file pathname are ignored as long as they are syntactically valid. This option exists because on some systems it is syntactically impossible to explicitly specify a directory any way other than as the directory portion of a pathname. Such a pathname is called a *directory pathname.*

SUBMIT
VMS uses SUBMIT. SUBMIT is for output only. It submits the contents of the file being written to the operating system as a job, after the file is closed.

ESTIMATED-LENGTH
An optional parameter. When specified, it must be followed by a decimal integer value. This option is only meaningful and permitted for output openings. It allows the user end to suggest to the server's file system how long the file is going to be. This can be useful for file systems that must preallocate files or file maps or that accrue performance benefits from knowing this information at the time the file is first opened. This estimate, if supplied, is not required to be exact. It can be ignored by servers to whom it is not useful or interesting. The units of the estimate are characters for CHARACTER openings and bytes of the agreed-upon size for BINARY openings. The CHARACTER units should be server units, if possible, but since this is only an estimate, Symbolics units are acceptable. See the section "Qfile Character Set Translation", page 162.

IF-EXISTS
Meaningful only for output openings, ignored otherwised, but not diagnosed as an error. It specifies the action to be taken if a file of the specified name already exists. IF-EXISTS always takes a value that specifies the action taken. The semantics of the values are derived from the Common Lisp specification and repeated here for completeness. If the file does not already exist, the IF-EXIST option and its value are ignored. The default value of the IF-EXIST parameter (that is, the action taken if IF-EXISTS is not specified) depends on whether or not the file system supports file versions. If it does, the default is ERROR (if an explicit version is given in the file pathname) and NEW-VERSION (if the version therein is the newest version). For file systems not supporting versions, the default is SUPERSEDE.

IF-EXISTS provides the mechanism where files can be overwritten or appended to. With the default setting of IF-EXISTS, new files are created by every output opening.

Operating systems supporting soft deletion can have different native policies of what to do if there is already a "deleted" file with the same name (and type and version, where appropriate) as a file to be created. The Symbolics computer native file system

(LMFS) effectively opts for SUPERSEDE, silently, even if not asked to do so. Other servers and file systems are urged to do similarly. Recommended action is to not allow deleted files from preventing successful file creation (with specific version number) even if an IF-EXISTS option weaker than SUPERSEDE, RENAME, or RENAME-AND-DELETE is specified or implied.

Here are the possible values and their meanings:

ERROR — Reports an error.

NEW-VERSION — Creates a new file with the same file name but with a larger version number. This is the default when the version component of the filename is newest. File systems without version numbers can choose to implement this by effectively treating it as SUPERSEDE.

RENAME — Renames the existing file to some other name and then creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

RENAME-AND-DELETE — Renames the existing file to some other name and then deletes it (but does not expunge it, on those systems that distinguish deletion from expunging). Then it creates a new file with the specified name. On most file systems, this renaming happens at the time of a successful close.

OVERWRITE — Output operations on the stream destructively modify the existing file. New data starts replacing old data at the very beginning of the file; however, it does not truncate the file back to length zero upon opening.

TRUNCATE — Output operations on the stream destructively modify the existing file. The file pointer is initially positioned at the beginning of the file; at that time, TRUNCATE truncates the file to length zero and frees disk storage occupied by it.

APPEND — Output operations on the stream destructively modify the existing file. New data are placed at the current end of the file.

SUPERSEDE — Supersedes the existing file. This means that the old file is totally destroyed, that is, removed

or deleted/expunged. The new file takes its place. If possible, the file system does not destroy the old file until the new stream is closed, against the possibility that the stream will be close-aborted. This differs from NEW-VERSION in that SUPERSEDE creates a new file with the same name as the old one, rather than a file name with a higher version number.

There are currently no standards on what a server can do if it cannot implement some of these actions.

IF-DOES-NOT-EXIST

Meaningful for input openings, never meaningful for PROBE openings, and sometimes meaningful for output openings. IF-DOES-NOT-EXIST takes a value token, which specifies the action to be taken if the file does not already exist. Like IF-EXISTS, it is Common Lisp-derived. The default is as follows: If this is a PROBE or read opening, or if the IF-EXISTS option is specified as OVERWRITE, TRUNCATE, or APPEND, the default is ERROR. Otherwise, the default is CREATE.

These are the values for IF-DOES-NOT-EXIST:

| | |
|---|---|
| ERROR | Reports an error. |
| CREATE | Creates an empty file with the specified name and then proceeds as if it already existed. |

#### 16.8.3.2 Qfile Open Response Result Values

The results of a successful open operation are reported in the response packet. They describe the file that is opened. We repeat here the specification of the OPEN results:

```
token results = creation-date length qfaslp characters
                        [author [byte-size [filepos]]]
string results = truename
```

`truename`

The full, correct name of the file, all links and nonspecific version numbers having been resolved. This is a pathname string in the native syntax of the server host. It should be computed by the server once it has opened the file, via some request to its operating system. The request can be of the form: "What file corresponds to this JFN, file number, pointer, etc.?" If the operating system supports version numbers, there is always an explicit version number in this string. (There is always a directory name, a file name, and so on.)

Some operating systems might not know the truename of an

output file until it is closed. It is permissible not to supply an explicit version number in the pathname in the OPEN response in this specific case.

`creation-date` The creation date of the file, expressed in the form `mm/dd/yy <SP> hh:mm:ss`. Note the space between the date and the time; thus, the creation date is really *two* tokens, not one. Leading zeros in the all fields are mandatory. The Symbolics computer software expects the logical quantity "creation date" to be the "data creation date" of the information in the file. By "data creation date", we mean the time that the bits in the file were decided upon.

This is a fairly tricky and subtle thing. Creation date does *not* necessarily mean the time the file system created the directory entry or records of the file. For systems that support modification or appending to files, it is usually the modification date of the file. Creation date can mean the date that the bit count or byte count of the file was set by an application program.

In the best case, the host file system supports a user-settable quantity, which the user set to an arbitrary time, to indicate that the data in this file were created a long time ago by someone else on another computer. The default value of this quantity, if the user has not set it, is the time someone last modified the information in the file. This is the ideal creation date.

This quantity, in the OPEN response for an output file, is disregarded by the user side, but must be syntactically valid anyway. See the section "Close Qfile Command", page 181.

The Symbolics computer system software uses this quantity as a unique identifier of file contents, for a given file name, type, and version, to prove that a file has not changed since it last recorded this quantity for a file.

The value of this time is expressed in the local time zone. The identification of this time zone is not communicated or negotiated in the protocol; this is a known deficiency.

`length` A <num> reporting the length of the file, in characters for CHARACTER openings and bytes of the agreed-upon size for BINARY openings. It is to be returned as 0 for output openings, even if appending to an existing file. The server usually only knows the length for a CHARACTER opening in server units, thus, it reports length in server units.

The handling of PROBE is ambiguous here. Nominally, the server should return the value of the length that it would return if the file were opened in the requested (BINARY or

CHARACTER) mode. This is what, for example, the PDP-10 server does. The LMFS server ignores CHARACTER/BINARY in PROBE and always gives the LMFS idea of the file length. No one has ever complained, which indicates that nobody uses this capability (to specify CHARACTER). But then again, LMFS knows innately whether a file is binary or character and thus knows the correct file length. PROBEF gives you no way to specify CHARACTER/BINARY in PROBE, although anyone certainly can open a file for :PROBE via **open**. Thus, it is fair to say that the servers should implement this distinction in behavior.

`qfaslp` is an indicator of whether or not this is thought to be a Symbolics binary file. The acceptable values are the strings **T** and **NIL**. It is obsolete, and present for compatibility. The `characters` value is the current way to convey this information: See the value `characters`. The `qfaslp` value can be reported as the Lisp inverse of the `characters` value.

`characters` is an indicator of whether or not this is a CHARACTER opening, as opposed to a BINARY one. The permissible values are the strings **T** and **NIL**, for CHARACTERS and BINARY, respectively. In the case of DEFAULT openings, the user end does not know this information by itself, but the value must be returned in any case. See the OPEN option DEFAULT. In other cases, it can be reported as the (represented) value of the CHARACTERS/BINARY parameter of the OPEN command. PROBE openings are never specified with DEFAULT.

The next three OPEN response values are optional in a restricted way: Only *trailing* values can be omitted and their delimiting spaces are omitted with them. That is, `filepos` can be omitted, if `byte-size` is omitted then `filepos` must also be, and if `author` is omitted then all three values must be omitted.

`author` The "name" of the "author" of the file. This is some kind of user identifier, whose format is highly system-specific. If the name is all numbers (or contains any nonalphanumeric characters) or if the server wants, the server can supply this value in Lisp String printed-representation (enclosed in ASCII doublequotes (042 octal), with embedded ASCII quotes or slashes (057 octal) preceded by ASCII slashes).

As with the `creation date` value, this quantity is supposed to represent the logical determinor of the current data content of the file, not necessarily the agency that actually created the file. In the best possible case, this is a user-settable quantity that the Symbolics computer software can set to assert a time-and-space distant creation of the data in the file. The Symbolics computer software uses it, too, as part of a unique identifier of the data content of the file.

byte-size — The byte-size agreed upon via the rules described for the BYTE-SIZE option. See the OPEN option BYTE-SIZE. This parameter is only meaningful for BINARY openings, but must be returned anyway for character openings, if present, which it must be if the filepos value is returned. See the value filepos. A numeric value or the string NIL is acceptable to byte-size.

filepos — A <num> giving the position of the logical file pointer, in characters or bytes as appropriate for the opening. This is always zero for an input opening and for an output opening creating a new file. For an output opening appending to an existing file, filepos is the number of characters or bytes, as appropriate, currently in the file. This number, for CHARACTER openings, is measured in server units: See the section "Qfile Character Set Translation", page 162.

### 16.8.4 Close Qfile Command

```
CLOSE
                token args = <none>
                string args = <none>
                token results = creation-date length qfaslp characters
                                [author [byte-size [filepos]]]
                string-results = truename
```

Terminates a data transfer. See the section "Qfile File Transfer Philosophy", page 160. A direct file handle must be given and a transfer must be active on that handle. Steps include:

- A synchronous mark is sent or awaited accordingly.
- The data channels, if any, are unbound from the open stream.
- The file is closed.

"Closing the file" has highly operating-system specific implementations and implications. It generally implies invalidation of the pointer or logical identifier obtained from the operating system when the file was "opened", and freeing of operating system and/or job resources associated with active file access. For output files, it involves ensuring that every last bit sent by the user has been successfully written to disk. The server should not send a successful response until all these things have completed successfully.

If a DELETE command has been sent to the stream during the time it is open, the file is "abort-closed" instead of closed normally. See the section "Delete Qfile Command", page 186.

The results are syntactically identical to the results of the OPEN command, with some semantic differences. For output, the creation date is known and must be

reported in such a way that that value is reported by the OPEN command if this file is later opened for input. Similarly, the exact truename is known after all close-time renaming of output files is done and must be reported with its version number (for operation systems supporting version numbers) resolved. Similarly, the length of an output file in characters is known and is reported (for CHARACTER openings) in server units.

In an earlier implementation, a formula for determining file length on PDP-10s stated that on TOPS-20 file length is the number of bytes in the file, if it was written using 7-bit bytes, otherwise, length is five times the number of words. For binary input, it is twice the number of words in the file. For ASCII output, "len" is the exact number of characters written, and for binary output, it is the number of words times the number of bytes per word for the current byte size.

What this actually means is that the reported length is always the number of bytes or characters (in PDP-10 character set, not Symbolics character set) in the file, except in a variety of circumstances the length is rounded up to the next even multiple of the number of bytes/characters packed per word. The system does not fail to remember the byte length; the problem is that some programs write the same files with a different byte size, so the file server must compensate. This applies to all PDP-10 operating systems (but the circumstances alluded to above are different for each operating system. For example, ITS TECO and DUMP write character files in 36-bit bytes. On TOPS-20 the file server writes binary files in 36-bit bytes rather than the BYTE-SIZE option to OPEN. KST files are naturally 36-bit bytes, but accessed as 9-bit bytes through the FILE protocol because of the 16-bit maximum byte-size limitation.)

#### 16.8.4.1 Qfile Close and Synchronous Marks

The transmission and receipt of a synchronous mark must occur as part of closing an open stream. For an output opening, the user side sends a synchronous mark on the data channel at the same time as the CLOSE command on the control connection. The server, upon receipt of the CLOSE command, must await the synchronous mark on the active data channel before closing the file (indicating successful completion). There is no other way to make sure all the data to be transmitted have been transmitted. This has some functional overlap with the EOF packet in the case of a normal close, but in the case of a premature close, there is no EOF packet. Even if the data are to be thrown away (for an opening abort-closed), data must be read out of the data channel up to and including the synchronous mark. Unless this is done, garbage can remain in the data channel and inadvertently be read during its next use. Note that the process (if that is the implementation) handling the data channel might receive (and process) the synchronous mark before the CLOSE command is read: the server must be aware of this possibility.

There is no difference in synchronous mark processing and synchronization between the two cases of normal and aborting close of an output transfer. In the aborting

case, the data channel never sees an EOF, but sees the synchronous mark occur in the data stream. The EOF conveys no information in the output CLOSE case, and is currently thought to be gratuitous (in design). See the section "Waiting for Acknowledgement of the EOF Packet on Writing with Qfile", page 183.

For an input opening (including data streams produced by the DIRECTORY and PROPERTIES commands), the server sends a synchronous mark on the data channel as part of the processing of the CLOSE command. The user awaits the control connection response *before* reading the data channel for the synchronous mark. Thus, to ensure that the server and user do not deadlock in output blocks, the server should send the synchronous mark *after* sending the control connection response.

In the input case, there is a substantial difference between premature and normal close: the CLOSE command is normally initiated by the user after receipt of the EOF from the data channel. In this (normal case), the data channel has sent the EOF and already stopped transferring data. In the premature close case, the user decides to issue the CLOSE command before receipt of the EOF: he has no way of knowing (and does not care) whether or not the server has sent the EOF. In this case, the server must actively *stop* the data channel's logical process from transferring data, if it is still transferring data (not yet having reached EOF) before responding to the CLOSE command. It does not matter to the user side whether or not the server actually has to stop its data channel or it has arrived at a stop by itself. The user wades through data, even no data, until passing a synchronous mark, EOF or no EOF.

#### 16.8.4.2 Waiting for Acknowledgement of the EOF Packet on Writing with Qfile

Formerly, we warned user-side writers that when the EOF packet is sent for a write opening, it is *critical* that server acknowledgement (that is, of reading the packet, a Chaosnet possibility) be awaited before the CLOSE command is sent, lest the server truncate the file. Chaosnet provides a way for the sender of a packet to determine if it has been read by the foreign program.

Some research has been done to determine the need for this, or for that matter, the need for the EOF on output at all, since the synchronous mark seems to convey all necessary information. The server, upon receipt of a CLOSE command, should not touch the file (open stream) until *receipt* of the synchronous mark. In the absence of data transfer errors, it is now believed to be superfluous (designwise) and a simple bug that the PDP-10 server closes its file *before* receipt of the synchronous mark.

On the other hand, the detection of asynchronous marks reporting write errors near the end of the file (specifically, after the user side has sent the final synchronous mark) does not work at all in the current protocol. This is an acknowledged deficiency. When the user sends the synchronous mark, the server is usually still writing data from the data channel to the stream, and might get an error and send an *asynchronous* mark to the user. See the section "Qfile Errors and Asynchronous Marks", page 197. Once the user side sends the CLOSE command, it (probably)

stops looking for asynchronous marks. There is no clearly definable time after sending the synchronous mark when it is safe to send the CLOSE command. This is a bug. The EOF acknowledgment is thought to be an an attempt to solve this problem, but does not, for two reasons. First, the reading of the EOF does not (depending on implmentation) mean that the actual disk I/O has completed. Second, the asynchronous mark is received via a different channel than the EOF acknowledgement and can thus still be arbitrarily delayed.

### 16.8.5 Finish Qfile Command

```
FINISH
                token args = <none>
                string args = <none>
                token results = creation-date length qfaslp characters
                                [author [byte-size [filepos]]]
                string-results = truename
```

This command finishes a file, that is, closes it but leaves it open for further I/O. The arguments, results, and their meaning are identical to those of the CLOSE command. See the section "Close Qfile Command", page 181. FINISH requires a file handle, which has same meaning as the file handle with the CLOSE command. The mark processing and synchronization protocol are identical to that of the CLOSE command. No EOF is ever involved.

Instead of closing the file, the server, for output, writes out all buffers and sets the byte count of the file. It leaves the file in such a state that if the system or server crashes at any time between now and the (later) time that the file was supposed to be closed, it would later appear as though the file had been CLOSEd by this command. FINISH is a reliability feature.

FINISH can be thought of as closing the file and reopening it with the file position pointer saved. It is somewhat pointless in the input case, but legal. The native Symbolics file system (LMFS) implements FINISH on an output file by an internal operation that effectively goes through the work of closing but leaves the file open for appending.

After successful response and synchronous mark-sending in the input case, active data transfer is resumed. That is, for a read opening, the data channel is reactivated and resumes sending data from the file at the point where the control channel interrupted it. See the section "Close Qfile Command", page 181. In the case of an output opening, the output channel is set back into a state where it is prepared to receive data to transmit to the file at the point it left off (when it received the synchronous mark).

### 16.8.6 Filepos Qfile Command

```
FILEPOS
        token args = <num>
        string args = <none>
        token results = <none>
        string results = <none>
```

This is QFILE's "set pointer", or file-position setting command.

An input file handle must be supplied in the FH field.

The FILEPOS command requests the server to reset its file access pointer to the indicated <num> of the file currently open. <num> can indicate either a byte number according to the current byte size being used or characters for character openings. If this is a CHARACTER opening, the <num> is measured in server units, *not* Symbolics units. See the section "Qfile Character Set Translation", page 162.

The user side expects and awaits a synchronous mark from the server *after* it has received a *successful* response to the FILEPOS command. By means of this mark, the user can differentiate data sent before the new file position was set from data sent thereafter. Thus, the server's required sequence of actions is:

1. Stop the data channel from transferring data.
2. Set the new file position in the open stream.
3. Issue a response.
4. If the setting of file position was successful, send a synchronous mark.
5. Successful or not, resume input (server sending) data flow in the data channel.

#### 16.8.6.1 The Filepos Qfile Command Byte Size Problem

There is a major design problem with FILEPOS and character set translation, with regard to character files only. Since there is not a one-to-one correspondence between characters in the host's file and characters perceived at the user end, for hosts that translate, the user has no way to determine file positions via a user-end program (as opposed to a native program running on the host). The user cannot determine file positions by counting the number of characters he has output, for some characters output might correspond to more than one character on some hosts, and might not on others. Thus, file positions in character files obtained by user-side programs writing files are worthless, as is the FILEPOS operation on character files. If the user is willing to compromise generality (that is, assume a Symbolics computer, nontranslating server), he can evade this problem.

See the section "Qfile Character Set Translation", page 162.

### 16.8.7 Delete Qfile Command

```
DELETE
        token args = <none>
        string args = [filename]
        token results = <none>
        string results = <none>
```

This is the QFILE interface to the "delete" action of host operating systems.

The `filename` string argument is the full pathname of a file to be deleted. The same syntactic and other constraints that apply to file pathnames in the OPEN command apply here. If there is a string argument, the file handle must not be supplied in the `FH` field.

"Delete" signifies whatever the word delete means to users of the host operating system concerned. That is, delete means soft deletion on TOPS-20 and LMFS and hard deletion on UNIX and Multics. If you try to delete a delete-through link on a LMFS, you delete its target instead.

If a file handle is given, the string argument must not be supplied. For example, with a file handle and no string argument, DELETE does not delete anything now. Instead, it sets a flag in the server or the operating system implementation of the open stream (ITS DELEWO — "delete while open") that causes the server, at the time the file (open stream) is closed, to "close-abort" it instead. See the section "Qfile File Transfer Philosophy", page 160. In the output case, this usually results in deletion of the file. For a file being appended to, the appendage is reverted or ignored.

The previous implementation stated that in the file handle case, the file is to be deleted at close time, input or output, no matter what. The combination of the Symbolics computer server and native file system do not implement this abort close deletion for input files.

We currently feel that the better definition of close-abort makes it look as though you never opened it. See the section "Qfile File Transfer Philosophy", page 160. The historical behavior seems to have an origin in the fact that the ITS DELEWO system call implements both sides of the file handle case of DELETE and its behavior for an input file is to delete the file. However, DELEWO deletes it *immediately* in this case, not at close time.

It has also been pointed out that the current protocol feature of implementing close-aborting via the DELETE command is misplaced and instead should be a parameter to CLOSE, as it is for Symbolics computer programs.

### 16.8.8 Rename Qfile Command

```
RENAME
        token args = <none>
        string args = [filename1] filename2
        token results = <none>
        string results = [new-truename [old-truename]]
```

This is the QFILE command to rename files.

There are either one or two string args. If there is a file handle given in the FH field, only one string arg, `filename2` is present. Otherwise, two are present.

The parameters `filename1` (if present) and `filename2` are pathnames. For a discussion of file pathnames: See the section "Open Qfile Command", page 170. They both have directory, file name, and file type (as appropriate to certain systems). Some operating systems can only rename within a directory. Nevertheless, the target pathname of the rename is fully specified in any case and it is up to the server on these systems to check for and reject an attempted cross-directory rename.

If there are two string args, the file designated by `filename1` should immediately be renamed to `filename2`. This should be an interface to the system's native rename operation, with all of its system-specific semantics and constraints.

If a file handle is given, only the second filename should be supplied. A supplied FH also means that the current command is a request to rename the file that is open on the data channel designated by the handle to `filename2`. The file is renamed at "some time", perhaps now or perhaps at CLOSE time. For a file that is going to be renamed at CLOSE time anyway, care must be taken to rename it to the new name at CLOSE time.

On ITS, this is called a "rename while open" (RENMWO) and is done immediately. On TOPS-20, a GTJFN is done to the open stream. Then, when a CLOSE is later performed, the CLOSF is done with CO%NRJ set and a RNAMF is subsequently done to the JFN so obtained.

The string results are optional in a restricted way: Only *trailing* values can be omitted and their delimiting spaces are omitted with them. That is, `old-truename` can be omitted, if `new-truename` is omitted then `old-truename` must also be. They are full pathnames, that is, truenames as described in the OPEN command description. They specify, respectively, what the new name of the file is and what the old name was. These quantities are fully known by the user side for operating systems such as UNIX that support neither version numbers nor subtle interactions of links with renaming. However, for operating systems that support either version numbers or interactions of links with renaming, the user side cannot know what the new name of the file is unless told by the server. For example, if you: `rename foo.lisp.7 to bar.lisp.newest`, which version of `bar.lisp` results?

### 16.8.9 Continue Qfile Command

```
CONTINUE
        token args = <none>
        string args = <none>
        token results = <none>
        string results = <none>
```

Resumes a data transfer that was temporarily suspended due to an error. For a discussion of errors: See the section "Qfile Errors and Asynchronous Marks", page 197.

A file handle must be supplied. The data channel designated by that handle must be in the asynchronously marked state, which data channels enter when an error of some kind occurs during data transfer. If the error is potentially recoverable, CONTINUE tries to resume from it. If the error is nonrecoverable, CONTINUE gives an error response.

### 16.8.10 Create-link Qfile Command

```
CREATE-LINK
        token args = <none>
        string args = link-pathname target-pathname
        token results = <none>
        string results = <none>
```

This command, on file systems that support links, creates a link.

A file handle must not be supplied. The pathname strings are host pathnames, as described in the OPEN command. The pathname strings might or might not have specific version, but always specify the pathname in the full pathname syntax of the server host.

A link named by `link-pathname`, which is a link to `target-pathname`, is created. If links have esoteric attributes on this system, a reasonable default set is applied.

CREATE-LINK is currently deficient insofar as it should return a truename.

### 16.8.11 Create-directory Qfile Command

```
CREATE-DIRECTORY
        token args = <none>
        string args = directory-pathname
        token results = <none>
        string results = <none>
```

This command, on file systems that support such an activity, creates a directory. Default access and creation attributes apply and should be assured by the server.

A file handle must not be supplied. The directory-pathname string is a directory pathname for the directory to be created. The PROBE-DIRECTORY option to the OPEN command discusses this: See the section "Qfile Open Options", page 172. The directory portion of the valid pathname directory-pathname is the specification of the directory to be created.

CREATE-DIRECTORY is currently deficient insofar as it should return a truename in some form, although there is no parallel for directory truenames in the protocol.

### 16.8.12 Expunge Qfile Command

```
EXPUNGE
        token args = [temporary]
        string args = directory-pathname
        token results = number-of-records
        string results = <none>
```

This command effects expunging of directories (final, actual removal of soft deleted files). For file systems that do not support soft deletion, the command is to be *ignored*, that is, responded to positively with no action.

A file handle must not be supplied. The directory-pathname string is a directory pathname for the directory to be created. The PROBE-DIRECTORY option to the OPEN command discusses this: See the section "Qfile Open Options", page 172. The directory portion of the valid pathname directory-pathname is the specification of the directory to be expunged.

The single optional argument temporary is the token TEMPORARY, when supplied, and is recognized by the PDP-10 server to mean that it should expunge temporary files, as well as deleted ones. The Lisp Machine user side, however, cannot send this.

The single token result is a <num> specifying how many records, blocks, or whatever unit is used to measure file storage on the host system, were recovered. The server should return 0 if it does not know.

There is currently a definitional unclarity as to whether directory-pathname is really a directory pathname or a wildcard pathname of files to be expunged. Thus, using TOPS-20 as an example, should a pathname of:

```
<FOO>A.B.*
```

expunges all deleted files in <FOO> or just versions of A.B?

There is also unclarity about whether or not wildcards are permitted, or required to be supported, in the directory portion of the pathname (representing an implicit request to expunge many directories). The TOPS-20 server allows this and TOPS-20 uses it. TENEX has such a facility but the server does not utilize it.

### 16.8.13 Set-file-system Qfile Command

```
SET-FILE-SYSTEM
        token args = varies
        string args = <none>
        token results = <none>
        string results = <none>
```

On hosts having multiple file systems, this command selects the identity of the file system to which all following commands in this dialogue are to be directed. Use it when that file system is other than the default file system that one gets by initiating a dialogue with that host.

The `token args` specify the file system. Their number and format is completely dependent on host-dependent parameters.

The only current use of this is for Symbolics computer servers to select the FEP hosts. In this case, the first token arg is the string FEP and the second is a <num> specifying the disk unit associated with the FEP host.

### 16.8.14 Login Qfile Command

```
LOGIN
        token args = [userid [password]]
        string args = <none>
        token results = username home-directory
        string results = personname [usergroup]
```

This command is for logging in the file server at the foreign host. This might or might not be necessary (depending on operating system and server) to allow the sheer possibility of a running a program (in this case, the QFILE server program) on the host. It establishes a user identity that is used by the operating system, among other things, for establishing file authors and determining file access rights.

The server has the option to reject with error any command except LOGIN if a successful LOGIN command has not been performed. This is recommended. Many operating systems perform the login function in a different process and/or environment than user programs. This above allowance allows the portion of the QFILE server running in the special login environment to have only the capability of processing the LOGIN command.

The `userid` argument specifies the user identity, in the server host's terms, to be established; the `password` is the password for that user identity. Both are optional. The fact that these are token args, and not string args, is a known deficiency and causes problems with user names containing spaces. An omitted user name means to log out, not in, and has the same semantics as an EOF packet on the control connection. An omitted password is valid if the host allows password-less login for

the specified userid. If the first character of the password is an ASCII asterisk (052 octal), user capabilities should be enabled.

The return values are somewhat subtle. username is the operating system's idea of the user's user identifier. This may be different than the userid argument, if, for instance, the user logs in with a synonym or short name.

home-directory is a representation of the name of the home directory of the user identity established. This has system-specific meaning. There is special-case code in each operating system's user side in the Symbolics computer to interpret various formats of this representation. It is thus impossible to write a consistent CFTP without knowing the behavior of each existing server in this regard. The canonical, default expected behavior, however, is that the representation of the home directory is a directory pathname for the home directory. The PROBE-DIRECTORY option to the OPEN command discusses this: See the section "Qfile Open Options", page 172.

It is a known deficiency that home-directory is a token result instead of a string result.

personname is the user's personal name, last name first, for example, "McGillicuddy, Aloysius X.". The server can pass a null string if this information is not available to it.

usergroup is an optional string value, currently returned only by the PDP-10 family systems. It is a one-character code identifying user group affiliation (project, user class, and so on). It is not required for other servers to return this argument.

### 16.8.15 Directory Qfile Command

```
DIRECTORY
        token args = [option]*
        string args = pathname
        token results = <none>
        string result = <none>
```

This is the command for directory listing, that is, finding the identities and attributes for spatially-related groups of files, directories, and links.

Using the DIRECTORY command is much like opening an input file. It starts a complex multiphase operation that must be terminated with a CLOSE command. A free input data channel must be selected by the user side and specified by its file handle in the FH field of the DIRECTORY command. Upon successful response, the requested data start flowing on this data channel and are followed by an EOF and a synchronous mark. See the section "Qfile Directory Data Format", page 192.

A CLOSE command must be issued to terminate the DIRECTORY operation, either after all the data have been read, or before then, if it is decided to abort the operation. The CLOSE is *identical to that for an input file*: the server stops data channel flow if it has not ended by itself and sends a synchronous mark after the CLOSE response, and so on. See the section "Close Qfile Command", page 181.

The pathname, in host syntax, specifies the files that are to be described. For a description of the constraints on pathnames of this type: See the section "Open Qfile Command", page 170. The pathname generally contains wildcard characters, in operating-system-specific format, describing potential file name matches. Most operating systems provide a facility that accepts such a pathname and disgorges vast quanitities of data about all files matching this pathname. Some operating systems allow wildcard (potential multiple) matches in the directory or device portions of the pathname; other operating systems do not. There is no clear contract at this time about what is expected, in this regard, of servers on systems that do not.

#### 16.8.15.1 Qfile Directory Data Format

The data sent over the data channel are a <directory-list>, in the following language:

```
<directory-list>    = <global-info> [<file-info>]*
<global-info>       = <NL> [<prop-line> <NL>]* <NL>
<file-info>         = <truename> <NL> [<prop-line> <NL>]* <NL>
<prop-line>         = <property-name> [<SP> <property-value>]
```

Each `<file-info>` gives the `truename` of one file and lists its properties. For a description of `truename`: See the section "Open Qfile Command", page 170. Each `<prop-line>` describes one property and its value for that file. The `<global-info>` describes properties of the file system as a whole.

The following properties are required to be among those listed in the <global-info>:

FREE-SPACE-DESCRIPTION
: A string-type property containing a Symbolics Character Set description of the amount of free file space available on the system. For a description of string properties: See the section "Qfile Directory Options", page 194.

SETTABLE-PROPERTIES
: A keyword-list-type property enumerating all `<property-name>`s for properties of the files listed that the server and operating system:

  - Consider user-settable
  - Support the change thereof via the CHANGE-PROPERTIES command

  There is a definitional ambiguity around the issue of listing changeable properties. Suppose many files (and, potentially, directories and links) are listed. Some changeable properties are appropriate to some files listed and some to others. The question is: which changeable properties shall be listed? It does not matter anymore what the server does, since the PROPERTIES command solves this. No one any longer looks at the <changeable-info> result of a directory list, or should. See the section "Properties Qfile Command", page 195.

Each <property-name> is an alphanumeric (although hyphens are permitted), uppercase token. Each <property-value> is an encoded form of a property value. The only overall syntactic constraint on <property-value>s is that they cannot contain <NL>s. Exactly how each <property-value> is encoded depends on the type of property. There are defined types of conversion as follows:

| | |
|---|---|
| Numbers | Numeric properties (for example, generation retention count) are encoded as <num>s. |
| Dates | As with creation date in the OPEN and CLOSE responses, that is, mm/dd/yy <SP> hh:mm:ss, with mandatory leading zeros and time zone interpreted in the local time zone (acknowledged deficiency). |
| Date-or-never's | Like a date, but the string Never (case not significant) is permitted to mean "never". This is appropriate for values such as the time a directory was last expunged. |
| Time intervals | An interval of time, or the string Never (case not significant). The range of values here is fairly complex to describe: it is the set of valid inputs to the function **time:parse-interval-or-never**. Examples of such strings are: `6 hours 4 minutes 2 seconds`, `3 days 5 hours`, and `2 days 5 hours 6 seconds`. No resolution finer than seconds can be used. |
| Keyword lists | Some, possibly none, uppercase tokens separated by single <SP>s. The list of settable properties is an example. |
| Boolean values | The two strings **T** or **NIL**, for true and false respectively. An omitted <property-value> (this is the only case where <property-value> can be omitted) implies a boolean value of true. When the <property-value> is omitted, the <SP> preceding it is also omitted. |
| Strings | The default conversion when the type of a property is not known. The string value of the property, converted per site specification, is used. It cannot contain <NL>s — this is an acknowledged deficiency. |

Here is the list of known <property-names>, sorted by conversion, known to the system:

| | |
|---|---|
| Numbers | BLOCK-SIZE, BYTE-SIZE, GENERATION-RETENTION-COUNT, LENGTH-IN-BLOCKS, LENGTH-IN-BYTES, DEFAULT-GENERATION-RETENTION-COUNT |
| Dates | CREATION-DATE, MODIFICATION-DATE |
| Date-or-never's | REFERENCE-DATE, INCREMENTAL-DUMP-DATE, COMPLETE-DUMP-DATE, DATE-LAST-EXPUNGED, EXPIRATION-DATE |

| | |
|---|---|
| Time intervals | AUTO-EXPUNGE-INTERVAL |
| Keyword Lists | SETTABLE-PROPERTIES, LINK-TRANSPARENCIES, DEFAULT-LINK-TRANSPARENCIES |
| Boolean values | DELETED, DONT-DELETE, DONT-DUMP, DONT-REAP, DELETE-PROTECT, SUPERSEDE-PROTECT, NOT-BACKED-UP, OFFLINE, TEMPORARY, CHARACTERS, DUMPED, DIRECTORY |
| Strings | ACCOUNT, AUTHOR, LINK-TO, PHYSICAL-VOLUME, PROTECTION, VOLUME-NAME, PACK-NUMBER, READER, DISK-SPACE-DESCRIPTION, and all others by default. |

The semantics of the properties are not documented here; they are defined by the Symbolics operating system.

#### 16.8.15.2 Qfile Directory Options

The following options to DIRECTORY are recognized:

DELETED
: Treats soft-deleted files as though they were there. Without this option, they are not to be included among the files listed. Such files have the DELETED property indicated as "true" among their properties.

FAST
: Speeds up the operation and data transmission by not listing any properties, that is, a null set of `<prop-line>`s, for the files concerned. It has been proposed that this option cause listing of the DIRECTORY property as "true" in spite of this, for directories that are elements of the list.

NO-EXTRA-INFO
: Speeds up the File System Editor (FSEdit) when listing the top level of hierarchichal directory systems. This option affects the appearance of directories in the listing. When given, it shortens the set of properties listed for directories (as opposed to files and links). The set of properties is abbreviated by the following rule: Any property requiring that FSEdit go to the actual directory file to extract information out of it (as opposed to information extractable out of the directory's directory entry) need not be listed. This typically eliminates listing of directory-specific properties such as information about default generation counts and expunge dates.

DIRECTORIES-ONLY
: This subtle and powerful option changes the semantics of DIRECTORY fairly drastically. Normally, the server returns information about all files, directories, and links whose pathnames match componentwise with the supplied pathname. This means that for each file, directory, or link to be listed, *its* directory name must match the (potentially wildcarded) directory name in the

supplied pathname, its file name must match the file name in the supplied pathname, and so on.

When DIRECTORIES-ONLY is supplied, the server is to list only *directories*, not whose pathname matches componentwise with the supplied pathname, but whose pathnames expressed as *directory pathnames* match the (potentially wildcarded) *directory portion* of the supplied pathname. The PROBE-DIRECTORY option to the OPEN command discusses this: See the section "Qfile Open Options", page 172. What is more, the returned `<truename>s` in the directory list must be expressed as *directory pathnames* as well.

It is not yet established what servers on hosts who do not support this type of action natively are to do when presented with DIRECTORIES-ONLY and a pathname with a wildcard directory component.

### 16.8.16 Properties Qfile Command

```
PROPERTIES
        token args = [AFH]
        string args = [pathname]
        token results = <none>
        string results = <none>
```

This command finds out the properties of one file. It operates like the DIRECTORY command, requiring an input data channel and a CLOSE cycle. It is conceptually different from DIRECTORY in the following ways:

1. There is no wildcard matching. This makes lookup more efficient in some systems. It is an error if the file specified does not exist or is not accessible: An appropriate specific error is reported by the server in this case. It is not an error if a DIRECTORY does not match any files.

2. All possible information is returned and there are no abbreviation options.

3. The file can be identified by file handle as the file that is open through the open stream operation. This solves some timing problems and is needed by the Symbolics computer system.

4. The list of settable properties returned in the directory listing is definable, exact, and relevant to the one specific file listed.

The file handle in the `FH` field specifies an otherwise-free, user-side-selected input data channel for the listing transfer as with DIRECTORY. The file whose properties are sought is specified in one of two ways:

1. Its pathname, a host-syntax pathname via the pathname string argument. AFH must not be supplied. For a description of host-syntax pathnames: See the section "Open Qfile Command", page 170.

2. An active file handle that is given as the token argument AFH. If supplied, it is the same file, which is open via this handle, about which properties are requested. The pathname string argument must not be supplied.

The data transfer, closing, and aborting protocol is *identical* to that of the DIRECTORY command: See the section "Directory Qfile Command", page 191. The data format, however, is somewhat different. It has a <file-properties> in the following addition to the DIRECTORY syntax:

```
<file-properties>    = <settable-props-list> <file-info>
<settable-props-list> = [<property-name>]* <NL>
```

The <property-name>s given in the <settable-props-list> are the settable properties for the single file listed. For a description of settable properties: See the section "Qfile Directory Data Format", page 192.

### 16.8.17 Change-properties Qfile Command

```
CHANGE-PROPERTIES
        token args = <none>
        string args = [pathname] [<prop-line>]*
        token results = <none>
        string results = <none>
```

This changes one or more properties of a file. The properties that can be changed are those listed in the SETTABLE-PROPERTIES property of the commands DIRECTORY or PROPERTIES.

The file is specified either by the pathname string argument or the active file handle in the FH field of the command. The presence of a nonnull FH field implies that the pathname string argument is not present.

The <prop-lines> have the same syntax and semantics as for DIRECTORY and PROPERTIES. The values of the properties are expressed in the converted representations: See the section "Directory Qfile Command", page 191.

The server is to attempt to modify all the properties indicated to their values indicated. There is currently no definition about what it should do if it can change some and not others.

In the file handle case, there is some craft required in the setting of attributes that are modified by the act of closing the file. For example, the Symbolics computer system changes the creation date of files open for output in certain circumstances.

On most systems, closing a stream open for output implictly sets the creation date. If this is the case, the server must either inhibit the system from doing so, change the creation date of the open stream with CHANGE-PROPERTIES, or reset the creation date to its required value after closing the file.

### 16.8.18 Complete Qfile Command

```
COMPLETE
        token args = [option]*
        string args = default-pathname  string
        token results = status
        string results = new-string
```

This command performs file pathname completion.

No file handle must be supplied.

`string` is a partial filename typed in by the user and `default-pathname` is the default name against which it is being typed. The filename is completed according to the files present in the host file system and the possibly-expanded string is returned.

Allowed options are DELETED, READ, WRITE, OLD, NEW-OK. DELETED means not to ignore deleted files — this applies to TENEX. READ means the file is going to be read (this is the default). WRITE means the file is going to be written, and affects version number defaulting. OLD means an existent file is required (this is the default). NEW-OK means that it is permissible for the string to complete to the name of a file that does not exist.

The returned status is NIL, OLD, or NEW. NIL means that an error occurred, OLD means that the string completed to the name of an existent file, NEW means that the string completed to a legal filename that is not the name of an existent file.

## 16.9 Qfile Errors and Asynchronous Marks

### 16.9.1 Qfile Error Responses

If a command is rejected, the server returns an error response that is a Symbolics character set data packet with an opcode of **chaos:dat-op** (= 200 octal). It has the format:

```
TID <SP> FH <SP> ERROR <SP> erc <SP> c <SP> message
```

The `TID` and `FH` are the `TID` and file handle from the command that failed. The other fields (`erc`, `c`, and `message`) indicate the type of error in three different ways.

The erc part is a 3-character error code: See the section "Qfile Error Codes", page 198. (The error codes are subject to change and extension without notice.) c is a single-character that is either F (106 octal) for a "fatal" error or R (122 octal) for a restartable one. All errors in response to commands are "fatal". Errors occurring during data transfer that generate asynchronous marks are allowed to be restartable if they can be resumed. See the section "Qfile Asynchronous Marks", page 198. The message part is a human-readable (Symbolics character set) error message, usually obtained from the host operating system. The message can contain <NL> characters.

### 16.9.2 Qfile Asynchronous Marks

When a data channel process, in either direction, encounters an error condition, an asynchronous mark is sent. Typically, asynchronous marks indicate error conditions in the transfer, such as running out of disk space or allocation, or a bad disk record. Some of these might be recoverable; some of these might not be recoverable. When reading, the asynchronous mark is sent over the data channel over which the data were being read. When writing, it is sent on the *control* connection. This is the one case where an asynchronous event happens on the control connection. (The data connection is not used in this case because its two channels are unrelated and the receive half might be in use for another transfer.)

An asynchronous mark is a packet with opcode **fs:%file-asynchronous-mark-opcode** (= 202 octal). Its data content is identical to an error response: See the section "Qfile Error Responses", page 197. The Symbolics server tries to install the TID of the OPEN, DIRECTORY, or PROPERTIES command that started the transfer. The file handle, FH, must be the one for the data channel that encountered the error.

When a data channel process encounters an error, it should send the asynchronous mark and enter the "asynchronous marked" state. It should also be prepared to exit this state and resume the transfer if the error can be proceeded from and it is given instruction to do so by the receipt of a CONTINUE command from the control connection. The data channel process should also be prepared to exit this state in response to a CLOSE command (as is necessary with all "fatal" errors and might be requested for restartable ones) if the server receives a CLOSE command for the channel instead of a CONTINUE command. In this case, the data channel process must exit the "asynchronous marked" state into performance of the CLOSE cycle: See the section "Close Qfile Command", page 181.

### 16.9.3 Qfile Error Codes

The selection of the three letter code that represents the type of error encountered is a complex subject. The basic scheme is for the server to map operating-system dependent error representations into these three-letter codes shown in this section, which try to characterize all possible file-access-related errors.

We do not describe the semantics of these errors here, but just give the phrases for which these three-letter codes are abbreviations. The current intended semantics of these phrases are described fully elsewhere. See the section "File-system Errors" in *Reference Guide to Symbolics-lisp*. The server should attempt to map operating-system-specific errors into these codes as defined by the meaning of the phrases (encoded in the names error flavors) therein.

One of the largest unresolved problems in the choice of error codes is whether, for example, an "access violation causing a rename to fail" is to be reported as an access violation or rename failure and why. Basically, the space of errors is two dimensional (for example, RENAME/DELETE **x** DNF/ATD) and the current error flavor structure does not allow this.

In general, the *Cannot XXX File* codes (where XXX is the three letter code) are catchalls for things that do not have more specific codes.

Although this is what the Multics server does, it would be more helpful if it were more descriptive.

This is the current state of the error table:

| | |
|---|---|
| ATD | Incorrect access to directory |
| ATF | Incorrect access to file |
| DND | "Don't delete" flag set |
| IOD | Invalid operation for directory |
| IOL | Invalid operation for link |
| IBS | Invalid byte size |
| IWC | Invalid wildcard |
| RAD | Rename across directories |
| REF | Rename to existing file |
| WNA | Wildcard not allowed |
| ACC | Access error |
| BUG | File system bug. This includes all protocol violations. |
| CCD | Cannot create directory |
| CCL | Cannot create link |
| CDF | Cannot delete file |
| CIR | Circular link |
| CRF | Rename failure |
| CSP | Change property failure |
| DAE | Directory already exists |
| DAT | Data error |
| DEV | Device not found |
| DNE | Directory not empty |
| DNF | Directory not found |
| FAE | File already exists |
| FNF | File not found |
| FOO | File open for output |
| FOR | Filepos out of range |

| | |
|---|---|
| **FTB** | File too big |
| **HNA** | Host not available |
| **ICO** | Inconsistent options |
| **IP?** | Invalid password |
| **IPS** | Invalid pathname syntax |
| **IPV** | Invalid property value |
| **LCK** | File locked |
| **LNF** | Link target not found |
| **LIP** | Login problems |
| **MSC** | Miscellaneous problems |
| **NAV** | Not available |
| **NER** | Not enough resources |
| **NET** | Network lossage |
| **NFS** | No file system |
| **NLI** | Not logged in |
| **NMR** | No more room |
| **UKC** | Unknown operation |
| **UKP** | Unknown property |
| **UNK** | Unknown user |
| **UUO** | Unimplemented option |
| **WKF** | Wrong kind of file |

# 17. Interfacing to the Network System

This document describes the interface to the lower levels of the network system. It does not describe the user interface or the service interface. It will help you implement a network protocol and integrate it with the existing mechanisms provided by the system. Before reading this document, you should be familiar with the standard issues involved with implementing network protocols.

The functions described here are not intended to be used by application programs nor directly by the service mechanism. Application programs interact with the user interface provided by specific network protocols (for example, Chaosnet) or preferably with the generic service mechanism that chooses the appropriate network protocol. See the section "The Lisp Machine Generic Network System", page 37.

The core network provides a standard model for dealing with packets (the basic unit of communication), interfaces (software and hardware to move packets from one machine to another), and network protocol implementations. It is this standard model and the interactions with it that are described elsewhere: See the section "Subpackets and Coercing Packets", page 204.

## 17.1 Packets: Interfacing to the Network System

Packets are the basic unit of communication between network nodes. The Symbolics computer implements a packet as an array of fixnums, typically **art-8b** or **art-16b**, depending on what the packet is being used for. For example, a Chaosnet packet is an **art-16b** array, but a TCP packet might be an **art-8b** array.

**art-string** is another useful array type. The Chaosnet often views the data portion of the packet as a string, and it uses the subpacket mechanism to make an **art-string** "packet" out of the data portion of the Chaos packet.

### 17.1.1 The Packet Pool

Packets are the most volatile item of the network. They are allocated (and deallocated) at rates of possibly hundreds per second. It is inefficient and impractical in both time and space to create a new packet each time one is needed. Therefore, a pool of packets exists from which users request and to which they return packets.

Certain decisions have been made to make the network more efficient. You should understand these decisions when you implement network protocols.

The microcode operates under one restriction: the packets with which it deals must be wired (that is, not pageable), because it is not allowed to take a page fault during packet transmission or reception. This restriction leaves the network four ways to implement packets:

- Have two pools of packets: one wired that is acceptable to the microcode, and another for users and networks that is not wired. Unwired packets are copied to wired packets for transmission, and wired packets are copied to unwired packets after reception.

- Have one pool of packets. Some packets are wired and accessible to the microcode for reception, and are unwired after reception. The other packets are available to users and networks that are wired before transmission.

- Have one pool of packets that are always wired.

- Have two pools of packets: one that is wired that is acceptable to the microcode, and a second pool of packets that are created and wired as necessary. When a user requests a packet, the wired pool is checked first. If the wired pool is empty, the unwired pool is checked. If the unwired pool is empty, more packets are created (with restrictions) and put on the unwired pool. When a packet is taken from the unwired pool, it is wired and is considered part of the wired pool.

The first two possibilities allow for a large number of user packets, because these packets do not need to be wired in physical memory and can therefore be created if more are needed immediately. However, the first possibility (used before Release 5.0) requires copying between the wired and unwired packets. Copying can be a time-consuming operation and might take a page fault on the unwired packet. The second possibility does not require copying, but wiring and unwiring also take time.

The third possibility does not require extra time to copy or to wire and unwire, nor can it take page faults on the packets. It also removes the need to keep track of the exact state of each packet (copied, wired, or unwired). For these reasons, the core network system for Release 5 implemented one pool of always-wired packets.

This implementation had a few drawbacks. Because all packets were wired, there had to be a limited number so they would not take up too much physical space. Extreme had to be taken to ensure that applications and protocol implementations deallocated all packets.

The Release 6.0 Chaosnet implementation uses the fourth possibility. The rationale is that under extreme circumstances or heavy load, as on a file server, the preallocated number of wired packets might not be enough. However, to keep from wiring and unwiring packets continuously, the user still sees a wired packet.

The restriction for creating more packets is that not more than one-fifth of the physical memory is wired. Therefore, a server machine with four memory boards might have more packets than a user machine with one memory board.

To minimize the number of wired packets, the system unwires packets in an attempt to make the number of wired packets no greater than the value of **neti:*target-number-of-wired-packet-buffers***. Packets are created and wired as the need arises, and possibly unwired to minimize physical memory requirements.

You can use **unwind-protect** to be sure to deallocate all packets that are allocated. For example:

```
(defun do-something-eventually-freeing-packet (packet)
  (unwind-protect
    (progn ... do some things ...
           (pass-the-packet-along-eventually-freeing-packet
             (prog1 packet (setq packet nil)))
           ... do some more things ...)
    (when packet (deallocate-packet packet))))
```

If while *doing some things* an error occurs and the function is exited, the *unwind-protect* will free the packet, which is part of the function's contract. When the packet is passed along, the **prog1** arranges for the packet to be passed as an argument and the variable to be set to **nil** in the scoping of the outer function. It is now the responsibility of the called function to return the packet. *Doing some more things* is not allowed to use the packet (because it is supposed to have been freed) and the **unwind-protect** clause will not free the packet, both because the variable **packet** was set to **nil**.

### 17.1.2 Reference Material: Packets

**neti:allocate-packet-buffer** &optional *(wait-p* **t***)* *Function*

Gets a packet from the free pool if there is one available and returns it to the caller. If there is no available packet and *wait-p* is **nil**, then **allocate-packet-buffer** returns **nil**. Otherwise the function waits for an available packet and returns it. There is also an **:allocate-packet** message to interfaces, which might be useful in some applications. See the message **:allocate-packet**, page 212.

**allocate-packet-buffer** is the lowest level function to allocate a packet and is not normally the function for networks or applications to call directly. Networks usually define their own packet allocation routine which, in addition to calling **allocate-packet-buffer**, coerces the packet to its own format and fills in default fields. See the section "Example: Interfacing to the Network System", page 206.

The variable **neti:raw-packet-buffer-size** has the number of bytes in the array returned by the function. See the variable **neti:raw-packet-buffer-size**, page 204.

**neti:deallocate-packet-buffer** *packet-buffer* *Function*

Gives *packet-buffer* back to the free pool of packets. *packet-buffer* may be a packet or any of its subpackets. **deallocate-packet-buffer** is the lowest level function to deallocate a packet. Networks usually define their own packet deallocate routine, which can be a stub (that is, it just calls **deallocate-packet-buffer**) or which can adjust meters and do other internal bookkeeping.

**neti:raw-packet-buffer-size** *Variable*

The variable stores the number of bytes in the array returned by **neti:allocate-packet-buffer**. This is the maximum number of bytes that any packet can have. The value depends on the architecture of the machine and, to a lesser extent, on the particular system release. It is not guaranteed to be the same from one release to another. Nevertheless, since packet buffers can be used as temporary storage, knowing their size can be important.

**neti:*target-number-of-wired-packet-buffers*** *Variable*

The number of packet buffers the system tries to keep wired. Users may set this to a higher value on machines that have a need for many packets (for example, on a server machine).

**neti:*actual-number-of-wired-packet-buffers*** *Meter*

The number of wired packet buffers actually wired. When a packet is returned to the packet pool this is compared with **neti:*target-number-of-wired-packet-buffers*** to determine whether the packet should be unwired.

**neti:*number-of-unwired-packet-buffers*** *Meter*

The number of unwired packet buffers. This can be thought of as the number of extra packets needed during the most extreme use of the network.

### 17.1.3 Subpackets and Coercing Packets

The packet that **neti:allocate-packet-buffer** returns is an **art-8b** array of some length that is dependent on the architecture of the machine (currently 1498 bytes on the 3600 family; these numbers might change slightly in future releases). See the variable **neti:raw-packet-buffer-size**, page 204. Raw **art-8b** arrays are often insufficient for network purposes. For example:

- The Chaosnet views the packet as 16-bit words, so it prefers an **art-16b** array. The Chaosnet also views the data portion of the Chaos packet (that is offset 16 bytes from the beginning of the packet) as a string. Control information is associated with each packet that is not part of the packet data.

- It is desirable to give the array a name using the named-structure-symbol feature of arrays so the packet prints out nicely and **describe** prints out the fields of the packet.

The array type and byte offset can be done with displaced arrays. The extra control information can be stored in the array leader. The named-structure-symbol can also be stored in the array leader. We refer to an array of this type that is displaced to a packet as a *subpacket*. The function **neti:get-sub-packet** takes a packet or subpacket and returns a subpacket with the desired attributes.

**neti:get-sub-packet** *sub-packet array-type nbytes* &optional *leader-length named-structure-symbol* *Function*

Returns an array of type *array-type* that is displaced *nbytes* (*not* array elements) from the beginning of *sub-packet* with a leader length of *leader-length*, if supplied, and a named structure symbol of *named-structure-symbol*, if supplied. *Note: array-type* must be a symbol. For example, the following is wrong:

```
(neti:get-sub-packet sub-packet art-8b 0)
```

It should be:

```
(neti:get-sub-packet sub-packet 'art-8b 0).
```

The byte offset is from the beginning of the subpacket passed as the argument, which is not necessarily the beginning of the network packet. The byte offset is in bytes, not array elements. For example, a TCP packet is offset from the beginning of an Internet packet, and the data portion of the TCP packet is offset from the beginning of the TCP packet, *not* the beginning of the Internet packet. A simplified TCP/IP implementation might look like this:

```
(setq ip-packet  (neti:get-sub-packet packet 'art-8b 0))
(setq tcp-packet (neti:get-sub-packet ip-packet 'art-8b tcp-packet-offset))
(setq tcp-data   (neti:get-sub-packet tcp-packet 'art-string tcp-data-offset))
```

A common way to define the elements of an array leader is to use the **:array-leader** option of **defstruct**. This is not sufficient for subpackets. The system requires several array-leader elements for its own use. The proper method is to include the **neti:sub-packet** structure using the **:include** option of **defstruct**. See the section "Example: Interfacing to the Network System", page 206. You should also use the **:size-symbol** option to get the size of the resulting leader, which can then be used as the *leader-length* argument to **get-sub-packet**.

The *leader-length* argument to **get-sub-packet** is not required. If it is not supplied, the system will supply its own. Subpackets always have a fill-pointer that is available for general use. The *named-structure-symbol* argument to **get-sub-packet** is also not required.

**get-sub-packet** only creates new displaced arrays if it needs to. When it does have to create a new subpacket with specific attributes, it caches the information in the packet buffer. The next time the same attributes are requested, **get-sub-packet** will return the cached subpacket instead of creating a new one.

*Note:* When using **art-16b** arrays, the first byte is the least significant byte of the 16-bit word and the second byte is the most significant. This Symbolics computer byte ordering (known as little-ender) is the same as that of PDP-11s and VAX-11s, but is reversed from the big-ender ordering of PDP-10s, PDP-20s and 68000s, for example. Chaosnet is a little-ender protocol, but the DoD Internet Protocol (IP) and the DoD Transmission Control Protocol (TCP) are big-ender protocols, so care must be taken when forming multibyte words from a packet or depositing a multibyte word into a packet.

A negative byte offset can be used to get space for a header at the beginning of a sub-packet. When this is done, it is necessary to copy the packet if there is not enough space at the beginning for the new header. Unless the caller knows that enough space is available, it should call **neti:get-sub-packet-maybe-copying** instead of **neti:get-sub-packet**.

**neti:get-sub-packet-maybe-copying** *free-flag length sub-packet array-type nbytes* &optional (*leader-length* **neti:sub-packet-size**) (*named-structure-symbol* **nil**) *Function*

Returns an array of type *array-type* that is displaced *nbytes* (*not* array elements) from the beginning of *sub-packet* with a leader length of *leader-length*, if supplied, and a named structure symbol of *named-structure-symbol*, if supplied. It also returns a new value for the *free-flag*. If a negative offset (*nbytes*) forces copying of the data, *free-flag* indicates whether the old packet should be freed. In this case, T is returned as its new value.

### 17.1.4 Example: Interfacing to the Network System

In this example we define a packet named **my-packet** that we abbreviate to **mypkt**. **mypkt**s have a protocol header that is 16-bit words, so we view a **mypkt** as an **art-16b** array. We view the data, however, as a string (an array of type **art-string**). In order to link **mypkts** together, we define a **link** slot in the packet's array-leader. This avoids creating conses that are likely to be scattered throughout virtual memory and that will soon be discarded.

First we define the packet structure and the byte offset to the data portion. Note that **my-packet-leader** includes the structure **neti:sub-packet**. This is required for all packets that have a meaningful array leader.

```
(defstruct (my-packet :array (:conc-name mypkt-)
                      (:constructor nil) (:size-symbol mypkt-data-start))
  opcode                        ;packet opcode
  destination-address           ;protocol address of the packet's destination
  source-address                ;protocol address of the packet's origin
  number)                       ;packet number for sequencing

(defstruct (my-packet-leader (:include neti:sub-packet) (:constructor nil)
                             (:conc-name mypkt-)
                             (:size-symbol mypkt-leader-length))
  link)                         ;the link to the next packet in a list.
                                ;NIL means end of list, T means not on list.

;;; we multiply by 2 because we consider my-packet an art-16b array,
;;; which has two bytes per element.
(defconst mypkt-data-start-byte-offset (* mypkt-data-start 2))
```

We now define coercion routines to convert a packet given to us by somebody else into a **mypkt**. We also define a routine that, given a **mypkt**, extracts the data portion as a string. Note in **packet-my-packet** both the leader length and the named structure symbol are supplied. The leader length is required here since we define and use a **link** slot in the array leader. The named structure symbol is supplied so a packet will print as **#<MY-PACKET 7042346>** and so **describe** will print the header fields. **my-packet-data-string** supplies neither the leader length nor a named structure symbol because we have no immediate need for either of them. The string does have a fill-pointer, which we are allowed to modify.

```
(defun packet-my-packet (packet)
  (neti:get-sub-packet packet 'art-16b 0 mypkt-leader-length 'my-packet))

(defun my-packet-data-string (mypkt)
  (neti:get-sub-packet mypkt 'art-string mypkt-data-start-byte-offset))
```

Here we define allocation and deallocation meters, and a simple routine that allocates a **mypkt**.

```
;;; Allocation and deallocation meters.
(defvar *mypkts-allocated* 0)
(defvar *mypkts-deallocated* 0)

(defun get-mypkt ()
  (prog1 (packet-my-packet (neti:allocate-packet-buffer))
         (incf *mypkts-allocated*)))
```

Alternatively, if we want to wait optionally and fill in some extra fields, we could define **get-mypkt** this way:

```
(defun get-mypkt (&optional (wait-p t))
  (let* ((packet (neti:allocate-packet-buffer wait-p))
         (mypkt nil))
    (when packet
      (incf *mypkts-allocated*)
      (setq mypkt (packet-my-packet packet))
      (alter-my-packet mypkt
                       opcode initial-opcode
                       destination-address initial-destination-address
                       source-address initial-destination-address
                       number initial-number)
      (alter-my-packet-leader mypkt link T))    ;not on a list
    mypkt))
```

Finally, we create a routine to free a **mypkt**:

```
(defun return-mypkt (mypkt)
  (incf *mypkts-deallocated*)
  (neti:deallocate-packet-buffer mypkt))
```

### 17.1.5 Miscellaneous: Packets

**neti:packet-being-transmitted** *sub-packet* *Function*

Returns non-**nil** if *sub-packet* is on the transmit list of some interface and **nil** if not. A packet may be deallocated when it is on a transmit list (**deallocate-packet-buffer** is careful), but packets may not be queued for transmission more than once. This routine is commonly used by retransmission routines. If a packet is already on some transmit list, it may not be requeued for transmission.

**neti:map-packet-buffers** *function* &rest *other-function-args* *Function*

Applies *function* (with any given arguments *other-function-args*) to each packet buffer, not just allocated packet buffers, not just free packet buffers. For example:

```
(neti:map-packet-buffers #'print)
```

will print each packet buffer. This is primarily a debugging tool to scan all the packets. A network implementor might determine some module is not freeing packets. By scanning all existing packet buffers, the implementor might be able to find the missing packets and determine why and/or where they were not freed.

Because there are a limited number of packet buffers, and because some network implementations have internal packet buffering (for example, the Chaosnet buffers packets that arrive out of order), it is possible to run out of packets in the free pool. When this happens a deadlock is reached, since no packets can be allocated to cause communication to relieve the deadlock and no packets can be received by the microcode. **allocate-packet-buffer** is usually the first to notice when there are no packet buffers in the free pool. After too long a period of inactivity, connections might timeout, close down, and return packets. This might spark a complete recovery, but at the expense of losing one or more connections.

To try and recover before timeouts happen a *packet buffer panic* is triggered. A packet buffer panic informs all known networks and all known interfaces that a packet buffer panic is happening. Networks and interfaces then try to deallocate packet buffers in such a way that no information is lost. For example, interfaces that do not guarantee packet delivery might free packets on the transmit list, and networks that do not depend on reliable transmission might free packets on out of order lists. In both of these cases the packets will be retransmitted eventually so no information is lost.

Packet buffer panics can be triggered for two reasons:

- **allocate-packet-buffer** will trigger one if there are no packets in the free pool of packets.
- The free pool can be periodically checked and a packet buffer panic triggered if it is empty.

These are accomplished using the following two functions:

**neti:packet-buffer-panic** *Function*

Triggers a packet buffer panic. All known networks and all known interfaces are sent a **:packet-buffer-panic** message inside a **without-interrupts**. This function should not be called unless a packet buffer panic is needed.

**neti:maybe-packet-buffer-panic** *Function*

Triggers a packet buffer panic if the free pool of packets is empty. It is safe to call this function periodically; the Chaosnet does so every 15 seconds.

## 17.2 Interfaces: Interfacing to the Network System

An interface, here, means the software that communicates with an individual piece of hardware (or sometimes software) that causes packets to be moved from one host to another. An interface's contract is twofold. On transmit, an interface formats the packet so that it is acceptable to the hardware. For example, the 3600 family determines the Ethernet address, does some extra formatting of the packet, and puts the packet on the microcode's transmit list. On receive, an interface accepts a packet from the hardware, performs some validity checks, determines for what network the packet is, and delivers the packet to the network.

An interface can also be an *encapsulation interface*. For example, it is possible to put non-Chaosnet protocol packets in Chaos UNC packets and use the Chaosnet as the transmission medium. In this case the interface puts the non-Chaosnet packet in a Chaos UNC packet for transmitting. On reception it extracts the non-Chaosnet packet from the UNC packet (using **neti:get-sub-packet**) and delivers it to the appropriate network.

Interfaces (and networks) are represented as flavor instances. Interfaces and networks send messages to each other to agree on parameters, to determine state, and to transmit and receive packets.

### 17.2.1 Standard Communication with Interfaces

This section describes the common uses of interfaces. It does not describe how to write your own interface. The information here should be sufficient for you to make your network protocol implementation communicate correctly with the existing software.

All active interfaces are kept on the variable **neti:*interfaces***. Networks should use this list when they need to know about all the available interfaces. When a network is enabled it usually adds itself as one of the network users of each interface that supports the network protocol. This list can also be used to initialize routing information and to distribute routing information.

**neti:*interfaces*** *Variable*

The list of all active interfaces. Interfaces add themselves to this list as part of network initialization.

Interfaces such as the Symbolics 3600's Ethernet interface support many protocols.

Just as an interface might respond to a network's request for information, it might in turn query the network for certain parameters in order to determine whether or not it really can support the network. Ethernet interfaces currently send an **:address-resolution-parameters** message back to the network as part of this determination. See the section "Interfacing to Ethernets", page 218. To determine if an interface can support a network, the **:protocol-supported** message may be sent to an interface.

**:protocol-supported** *network* *Message*

Queries the interface whether it can support network *network*. Three classes of values can be returned.

1. The keyword **:unsupported** means the interface cannot transmit or receive packets for *network*.

2. A number or array (depending on the protocol address format of *network*) means the interface can support the network and that the interface insists the protocol address of the interface for *network* is the returned value.

3. The symbol **t** means the interface can support *network* but does not insist on a protocol address.

Interfaces and networks do not automatically start sending packets back and forth; they are explicitly informed about each other. Specifically, for each interface in ***interfaces*** a network should determine if the interface supports the network and if there is a local protocol address that can be assigned to the interface. If these conditions are met, the interface can add itself as one of the network users of the interface. This is done with the **:add-network** message to interfaces.

**:add-network** *network local-address* *Message*

Requests the interface to start receiving packets for, and to start accepting packets for transmit from, *network*. *protocol-address* is to be the interface's local protocol address for *network*.

If the network wishes, all of this can be performed automatically by the function **neti:find-network-interfaces**.

**neti:find-network-interfaces** *network* *Function*

Asks all known interfaces whether they support *network*. Returns a list of conses, one cons for each interface that supports *network*. Each cons is of

the form (*interface* . *protocol-address*). An interface that requests a specific address gets it if it is available; other interfaces are assigned the remaining addresses arbitrarily. **neti:find-network-interfaces** returns **nil** if no interface supports *network*. An **:add-network** message is sent to each interface that is assigned an address.

It is not necessary for networks to remember the protocol address of each interface. Instead, you can use the **:protocol-address** message to an interface. This can be useful for initializing and distributing routing information, and for determining if the interface is currently supporting the network.

**:protocol-address** *network* *Message*

Returns *network*'s local protocol address of the interface if the interface is currently supporting the network. Otherwise, **nil** is returned.

### 17.2.2 Sending a Packet to an Interface

After networks and interfaces negotiate and a network adds itself as one of the users of an interface, it is possible to receive and transmit packets on the interface. Networks transmit packets by sending a message to the appropriate interface, as described in this section. In the other direction, interfaces deliver packets to networks. See the section "Packet Reception: Interfacing to the Network System", page 216.

Simply asking an interface to transmit a raw (sub)packet is not sufficient. If the packet contains data that may need to be retransmitted, the interface should not free the packet. Networks also send control information that is not retransmitted, so it is allowable for the interface to free such a packet after transmission. Therefore, an interface needs to be told whether or not it must free the (sub)packet after transmission.

The interface must also know to whom to send the packet. A network is responsible for determining to what protocol address the packet should be sent, but it is *not* responsible for determining the hardware address of the foreign host. An interface is given both the network and the protocol address of the destination and does whatever is necessary to deliver the packet to the network implementation of the foreign host.

**:transmit-packet** *protocol-packet free-flag network protocol-address* *Message*

Causes *protocol-packet* to be transmitted on the interface. The destination of the packet is *protocol-address* within *network*'s addressing domain. It is the responsibility of the interface to convert the protocol address into a hardware address, if necessary. It uses *protocol-address*, *network*, and the information communicated during the **:add-network** message to do the conversion. If *free-flag* is **nil** the packet is not freed by the interface after it is transmitted. This is common for packets that might need to be retransmitted. If *free-flag* is not **nil**, the packet will be freed by the interface after transmission.

### 17.2.3 Miscellaneous: Interfaces

Some interfaces need to prepend bytes to a packet before transmission. A Chaosnet UNC encapsulation interface would require 16 bytes for the Chaosnet header. If it can be determined beforehand which interface will probably transmit a packet, it is desirable to allocate a packet with the necessary number of available bytes at the beginning. Otherwise, the packet would have to be copied in order to make room for the additional bytes. The **:allocate-packet** message to a network interface returns such a packet.

**:allocate-packet** &optional *(wait-p t)* *Message*
Similar to the **allocate-packet-buffer** function. It gets a packet from the free pool of packets if one is available, possibly waiting. The (sub)packet that is returned to the caller might have an addition byte offset, depending on the transmit needs of the interface.

## 17.3 Networks: Interfacing to the Network System

An implementor of a network protocol or protocols usually writes code for routing packets on output, processing packets on input, connection control, handling overdue events (timeouts), opening and closing of connections, and receiving packets from and delivering packets to users and applications. These issues are quite specific to the particular protocol(s) being implemented and are beyond the scope of this document. What is documented here are the conventions for integrating a network protocol implementation with the mechanisms of the system.

### 17.3.1 Defining a Network

Networks are represented as flavor instances. Networks that are in the namespace database are based on the **network** flavor. Each network flavor has a keyword associated with it that identifies the type of the network. The namespace system uses this to convert from the network type to the appropriate flavor to instantiate. The flavor the namespace system uses is stored on the **net:network-type-flavor** property of the type keyword.

**net:network** *Flavor*
The flavor on which networks that are in the namespace database are built.

**net:network-type-flavor** *Property*
A property given to keyword symbols. The symbol identifies the type of network; the value is the flavor to instantiate. If there is no such property, the namespace system defaults the flavor to **network**.

For example, the first step in the system's definition of the Chaosnet is:

```
(defflavor chaos-network () (network))
(defprop :chaos chaos-network net:network-type-flavor)
```

You can define a network that is not in the namespace database. This is useful when developing and debugging a network or when implementing a private network that does not need to be in the namespace database. You must define appropriate methods to sufficiently masquerade as a network based on the **network** flavor. As part of this masquerading, simply define a flavor without any base flavors. You need not define a type and give the type symbol a **net:network-type-flavor**, but it will not do any harm. For example:

```
(defflavor magic-network () ())
(defprop :magic magic-network net:network-type-flavor)
```

As an inverse of the **net:network-type-flavor** property, networks based on the **network** flavor can be sent a **:type** message that returns the keyword identifying the type of the network. By convention, a method should be defined for masquerading networks as well.

**:type** *Message*

Returns the type keyword of the network.

For our magic network, this would be defined as:

```
(defmethod (magic-network :type) () ':magic)
```

### 17.3.2 Network Addresses: Interfacing to the Network System

Humans usually refer to hosts by textual names. Applications usually convert the name into a host object by calling **si:parse-host**. The lower-level portions of networks, however, deal with *parsed addresses*. A parsed address is an object that represents the network address of a host in the form most convenient for the machine and network implementation. This representation is often not very useful for a human or for transmitting as text (for example, when transacting with a namespace server). The textual form of an address is the *unparsed address* and is a string. For example, the hexadecimal number #X+0A000006 is the parsed form of the unparsed Internet address "10.0.0.6". To convert between the two formats, methods for **:parse-address** and **:unparse-address** need to be defined.

**:parse-address** *address* *Message*

Returns a network address by parsing *address*, which is a string. *address* is a textual representation of a network address. The result may be any object and depends on the addressing format and needs of the network, and is usually a number or **art-8b** array. The method of the **network** base flavor returns the argument *address*.

**:unparse-address** *parsed-address* *Message*

Returns a string that is the textual representation of the network address *parsed-address*. The methods for **:parse-address** and **:unparse-address** should be inverses; **eq**-ness is not required. The method of the **network** base flavor returns the argument *parsed-address*.

For example, parsing "401" as a Chaosnet address returns the octal number 401, which in turn unparses as a string "401". This is accomplished by the following definitions.

```
(defmethod (chaos-network :parse-address) (string)
  (parse-number string 0 nil 8 t))

(defmethod (chaos-network :unparse-address) (address)
  (format nil "~O" address))
```

### 17.3.3 Interfacing to the Service Lookup Mechanism

The service lookup mechanism and the concepts of services, protocols, mediums, and desirability are discussed elsewhere. (See the section "The Lisp Machine Generic Network System", page 37.) Briefly, given a high-level goal such as remote login to a host, the system searches for possible ways to realize the goal. This involves such things as determining what services a host provides, what paths there are to the host, what protocols the Symbolics computer implements, what mediums those protocols require, and how those mediums are implemented. Interfacing a network to this search is rather straightforward. It is not necessary to define methods for the messages in this section if the network will not participate in the service lookup mechanism.

All networks are not created equal. Networks (and implementations) can differ in processing speed, amount of overhead, time to recover from lost packets or errors, size of packets, and supported features (for example, broadcast or existence of out-of-band signals). *Desirability* is the result of weighing these factors. The desirability is a floating-point number between 0.0 and 1.0. Most networks have a constant desirability, though a network may determine the desirability dynamically. For example, a network based on telephone calls might compute the desirability based on time of day.

**:desirability** *Message*

Returns a floating-point number between 0.0 and 1.0 that is the relative desirability of using the network as a medium.

Some networks can support broadcasting a request for a service throughout the network. Sometimes the ability to broadcast is based on the protocol. For example, it is often reasonable to broadcast a request for the current time, but it might not be reasonable to broadcast a request for login service.

**:supports-broadcast** *protocol-name* *Message*

Returns non-**nil** if *protocol-name*, a keyword, can be supported by broadcasting a request throughout the network. Otherwise, **nil** is returned. The method of the **network** base flavor returns **nil**.

The implementation of a protocol communicates over a medium. General protocols usually use a **:byte-stream** or **:datagram** medium. More specialized protocols can use more specialized mediums. To actually implement a protocol and its *base medium* over a particular network, the network-specific medium must be determined.

**:possible-medium-for-protocol** *protocol-name base-medium* *Message*

Returns the name of the medium to use to implement *base-medium* on the network. If *protocol* is not supported, or a medium cannot be determined from *base-medium*, then **nil** may be returned. The method of the **network** base flavor returns **nil**.

Some networks have services that all machines on the network are expected (though not required) to support.

**:default-services** *Message*

Returns a list of three element lists that are the default services that each host that implements the network is expected to provide. The elements of the lists are:

1. generic protocol name
2. network specific medium name
3. network specific protocol name

For example, the Chaosnet might return the following:

```
((:chaos-status :chaos-simple :chaos-status)
 (:uptime :chaos-simple :uptime-simple))
```

The method of the **network** base flavor returns **nil**.

### 17.3.4 Invoking Mediums: Interfacing to the Network System

The service lookup mechanism now has enough information to know what to do, but it is not quite able to do it yet. It can ask the network to convert a base medium for a protocol into a network-specific medium. It must also be able to invoke the specific medium. To do this, you use the **net:define-medium** macro. If the network medium implements a generic base medium (for example, **:byte-stream** or **:datagram**), then existing protocol implementations defined with **net:define-protocol** will be able to use the network medium. For nongeneric mediums you can use **net:define-protocol** to support high-level protocols in the ways specific to the network. **net:define-medium** and **net:define-protocol** are described elsewhere. See the section "The Lisp Machine Generic Network System", page 37.

### 17.3.5 Packet Reception: Interfacing to the Network System

After a network adds itself as a user of an interface, using the **:add-network** message to interfaces, the interface may start receiving packets on behalf of the network. When a packet arrives and the interface determines to which network the packet should be delivered, it sends the network a **:receive-packet** message with the packet as the first argument. The interface supplies two more arguments: the interface on which the packet was received, and the network's protocol address of the interface. These arguments might be useful in updating routing tables or implementing an interface keep-alive count. Additional arguments may be added in future releases.

The packet that is delivered to the network is just a packet. One of the first things that should be done is to extract the protocol packet from the packet by using **neti:get-sub-packet** or by using a function for that purpose as in the `packet-my-packet` example described elsewhere: See the section "Example: Interfacing to the Network System", page 206.

*Note:* There are some circumstances when the interface argument is **nil**. This usually happens when a network or an interface determines that the packet is destined for itself. In this case, the interface on which the packet was received does not really have a meaning since the packet was not really received. Even though the interface is **nil**, the network's protocol address of the intended interface is still supplied.

**:receive-packet** *packet interface interface-protocol-address* *Message*

Processes *packet* according to the definition of the network. *interface* is the interface from which the packet was received, or possibly **nil** if the packet was not really received by an interface. *interface-protocol-address* is the network's protocol address of the interface and is always valid even if *interface* is **nil**.

### 17.3.6 Packet Transmission: Interfacing to the Network System

The routing layer of a network determines the interface and the immediate destination host for a packet by using algorithms and databases defined by the particular network. The routing layer then sends the packet and immediate destination host as arguments in the **:transmit-packet** message to the interface. See the section "Sending a Packet to an Interface", page 211.

### 17.3.7 Network Errors: Interfacing to the Network System

For information on network errors: See the section "Conditions" in *Reference Guide to Symbolics-lisp*.

### 17.3.8 Initialization, Reset, and Enable: Interfacing to the Network System

Once a network is fully defined, instances of it can be made. This is often done automatically by the namespace system as needed. Of all the known networks, only *local networks*, networks to which the machine is attached, actually receive and transmit packets. They must be initialized when the machine is cold or warm booted. You may also reinitialize individual networks or the entire network system manually.

The first part of initializing local networks is for the networks to be declared local. This is done by putting them on the list **neti:*local-networks***. When Lisp is initialized during booting, the system scans the network addresses of the local machine, as determined by the namespace database, and puts the networks it finds there on ***local-networks***.

**neti:*local-networks*** *Variable*

The list of networks to which the local machine is directly attached.

If a network is local but is masquerading as a namespace object then it will not be automatically put on ***local-networks***. To interact with global network operations, the network should add itself to ***local-networks***. The proper time to do this is after the primary network is enabled but before the system enables all other local networks. This is done by adding an initialization to the following list.

**net:after-network-initialization-list** *Variable*

This variable is an initialization list that contains initializations that are performed after the primary network is determined and enabled.

For example (remember, this is only for masquerading networks):

```
;;; make an instance that we always consider to be local
(defvar *magic-network* (make-instance 'magic-network))

;;; put it on *local-networks* when the file is loaded
(push *magic-network* neti:*local-networks*)

;;; and make sure it gets on *local-networks* when the
;;; machine is warm or cold booted.
(add-initialization "Add Magic Network"
  '(push *magic-network* neti:*local-networks*)
  nil 'neti:after-network-initialization-list)
```

You can perform two major operations on networks: *reset* and *enable*. There is also a minor operation that some networks support optionally or internally: *disable*. Resetting a network completely shuts down the operation of the network and everything associated with it. Enabling a network initializes databases, attaches the network to interfaces that support it, and makes the network available for use. Disabling a network puts it in a quiescent state where packets are not processed.

The network can later be enabled and should continue operation from the point at which it was disabled. As part of the system's initialization of the network system it sends each network on ***local-networks*** a **:reset** message followed by an **:enable** message.

**:reset** *Message*

Requests the network to reset itself. This normally involves closing down connections, freeing queued packets awaiting processing, entering a state that refuses to receive or transmit packets, and perhaps informing users and applications of the network that it is shutting down.

**:enable** *Message*

Requests the network to enable itself. This normally involves (re)initializing databases, attaching to interfaces that support the network, and perhaps announcing to users and applications that the network is now available.

**:disable** *Message*

Requests the network to disable itself. This normally involves freeing queued-up packets and entering a state that refuses to receive or transmit packets. It does not affect connections. If the network is then enabled, all connections should be intact (provided timeout intervals did not expire) and the network should be able to continue from the point just before disabling. If disabling is supported, it is usually the first step in a reset operation.

### 17.3.9 Byte Stream Conventions: Interfacing to the Network System

If the network provides a byte stream interface, the stream should support some additional messages in addition to the standard stream messages.

**:foreign-host** *Message*

Returns the host object of the foreign side of the connection.

**:accept** *Message*

Accepts a request for connection.

**:reject** &optional *reason* *Message*

Rejects a request for connection. Reason, if supplied, is a textual reason for refusal and should be communicated to the requestor if the network is able to do so.

### 17.3.10 Interfacing to Ethernets

To convert from protocol addresses to Ethernet hardware addresses, Symbolics uses the address resolution scheme as described in An Ethernet Address Resolution Protocol, ARPA document RFC 826. Part of the initial negotiation between Ethernet interfaces and networks is for the interface to determine what the value of

the Ethernet type field is for the network and other relevant parameters for address resolution.

**:address-resolution-parameters** *Message*

Returns multiple values describing the network's Ethernet attributes. Inapplicable values need not be returned or may be returned as **nil**. The values are:

1. The 16-bit Ethernet type field as assigned to this network protocol by Xerox. *Note:* The first byte that is transmitted is the *most* significant byte of this 16-bit word. This is the opposite of the normal Symbolics byte ordering within words.

2. The number of bytes in a protocol address for the network.

3. A keyword describing the format of an address for the network. This may be **:little** if the address is a number and the first byte is the least significant byte of the address, **:big** if the address is a number and the first byte is the most significant byte of the address, **:array** if the address is an **art-8b** array, or **:fixnum-big** if the address is a fixnum and the first byte is the most significant.

4. The network protocol address that should cause hardware broadcast if the interface supports hardware broadcast and if the interface is asked to transmit a packet to this protocol address.

For example, the Chaosnet defines this method as:

```
(defmethod (chaos-network :address-resolution-parameters) ()
  (values #x+0804 2 ':little 0))
```

### 17.3.11 Interaction with Peek Network Mode

The Peek program can maintain visual information about networks and interfaces.

Networks that are not based on the **network** base flavor may define methods for the following messages that return **nil**.

**:peek-header** *Message*

Returns a scroll item that is the header display for the network. The method of the **network** base flavor returns a scroll item that enables one to reset, enable, describe or inspect the network. It is usually unnecessary to provide a primary method.

**:peek** *Message*

Returns a scroll item (usually a list of scroll items) detailing various parts of the network. This can include details of connections, meters, debugging information, and routing tables. The method of the **network** base flavor returns **nil**.

## 17.4 Starting Servers: Interfacing to the Network System

Most of the discussion so far has assumed that the Symbolics computer is requesting a service. The Symbolics computer can also be a server, that is, the provider of a service. A request for a service is handled in a way specific to the definition of the network. The code that provides the service is often general. Servers are usually defined with **net:define-server**. To invoke a server, a network must do several things.

### 17.4.1 Finding a Server Description

The network first converts the network specific request (for example, contact name in Chaosnet or port number in TCP) into a protocol keyword. This is done in a network-dependent manner using a database defined and maintained by the network.

The network next finds a *server description* for the protocol. In this discussion a server description is a structure that identifies what protocol the server implements, what medium the implementation uses, the function to call to provide the service, the number and type of arguments the function expects, and a list of additional properties associated with the server. Server descriptions are kept in the list **neti:*servers*** and the protocol the server implements can be obtained by calling **neti:server-protocol-name** with the server as the argument.

If a server is found for the protocol, it is customary to spawn a process at this point (using **process-run-function**). This allows the network to continue its duties independently of server establishment and operation. One of the properties on the property list of the server description is **:process-name**. Its value is the suggested name for the process.

### 17.4.2 Calling the Server Function

At this point, things get involved. The function that eventually gets called to set up for calling the server function is **neti:funcall-server-internal-function**. The first argument is the server description. The rest of the arguments are keyword-value pairs. Some of the pairs are based on the property list of the server, some are based on which medium the server uses, and some are based on the arguments to the server. It is acceptable to supply pairs that are not necessarily needed. Arguments to the server that are needed but not supplied default to **nil**.

#### 17.4.2.1 General Arguments: Starting Servers: Interfacing to the Network System

If the **:reject-unless-trusted** property is not **nil** and the host requesting the service is not trusted, the request for the service should be refused.

If **:trusted-p** is one of the arguments to the server, then **:trusted-p** and a determination of the requested host's trustedness should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

If the **:who-line** property is not **nil** and if the network supports noting the establishment and closing of servers, then **net:note-server-established** should be called. When the connection is closed the network should call **net:note-server-closed**.

If **:host** is one of the arguments to the server, then **:host** and the host object for the foreign host should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

If **:network** is one of the arguments to the server, then **:network** and the network invoking the server should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

#### 17.4.2.2 Medium Arguments: Starting Servers: Interfacing to the Network System

The major dispatch is based on which medium the server uses. Networks may support two generic mediums: **:byte-stream** and **:datagram**. A network may also implement network-specific mediums and network-specific servers that use them.

If the server uses the **:byte-stream** medium, **:stream** and a stream should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**. Unless there is an explicit **:accept-p nil** pair in the **:stream-options** property of the server, the request for connection is automatically accepted. If the **:accept-p** property is **nil**, the server is responsible for accepting or rejecting the request by sending either the **:accept** or **:reject** message, respectively, to the stream. If the server returns normally and if the **:no-eof** property of the server is **nil** or not specified, the stream should be closed synchronously. Otherwise, the stream should be closed in abort mode.

If the server uses the **:datagram** medium, a different set of arguments is passed to **neti:funcall-server-internal-function**. Three keyword-value pairs are always supplied. The server does not need to accept these keywords.

- **:response-array** is an **art-8b** or **art-string** array for the response
- **:response-array-start** is the first array index available for the response
- **:response-array-end** is the last array index (exclusive) available for the response

If **:request-array** is one of the arguments to the server, then three additional keyword-value pairs are supplied.

- **:request-array** is an **art-8b** or **art-string** array that contains the request
- **:request-array-start** is the first array index that contains the request
- **:request-array-end** is the last array index (exclusive) that contains the request

Server functions for datagram protocols return two values. The first is a success flag. If this is **nil**, the request is refused. If it is not **nil**, a reply is generated. The second value is either a number that is the number of bytes in the response

array that are valid, or a string that is the response and that must be copied into the response array.

If the server uses a network-specific medium, the network should supply whatever keyword-value pairs it determines are needed by the server.

Remember, it is acceptable to supply keyword-value pairs to **neti:funcall-server-internal-function** that are not needed by the server. This might make setting up the argument list to **neti:funcall-server-internal-function** easier.

### 17.4.3 Reference Material: Starting Servers

**neti:*servers*** *Variable*
The list of all supported servers, as defined by the **net:define-server** macro.

**neti:server-protocol-name** *server* *Function*
Returns the keyword that identifies the protocol the server implements.

**neti:server-medium-type** *server* *Function*
Returns the keyword that identifies what medium the server uses.

**neti:server-function** *server* *Function*
Returns the function that gets called to perform the service.

**neti:server-number-of-arguments** *server* *Function*
Returns the number of arguments the function expects.

**neti:server-argument-descriptions** *server* *Function*
Returns a list of keywords that identify the expected arguments. For example, the list (**:stream :host**) means the first argument is a stream and the second argument is the host object of the requesting host.

**neti:server-property-list** *server* *Function*
Additional properties of the server. This might include a suggested process name and stream options.

**neti:funcall-server-internal-function** *server* &rest *arguments* *Function*
This is the general function for invoking a server after the network has determined the necessary arguments for the server function. *server* is a server description structure. *arguments* are keyword-value pairs off the possible information the server may need to know.
**funcall-server-internal-function** matches the supplied keywords with the argument descriptions in *server* and invokes the server function. This function is just an argument matcher and does not close byte streams or handle the result of a datagram server.

*Note:* In the following two functions, *server-description* is a description of an instance of a server, *not* a description of the characteristics of a server.

**net:note-server-established** *network protocol-name foreign-host connection* &rest *plist* *Function*

Informs the system that a new server has been established. The server will appear in Peek Server mode and might appear in the status line. A *server description*, which is a copy of the arguments in a structure, is returned.

**net:note-server-closed** *connection by-foreign-host* &optional *server-description* *Function*

Informs the system that the server for *server-description* has closed. If *server-description* is not given, the system searches for *connection* in the list of active servers. *by-foreign-host* is **nil** if the connection was closed locally, or not **nil** if the connection was closed by the foreign host.

# Index

## B

## D

## E

## F F F

## G

## G

## G

## H

H H

**K** **K** **K**

# M

M M

## N

## N

## N

## P P P

# Q Q Q

## R R R

## S

# T T T

## U U U

## V V V

# W



# X