Table of Contents

Page

1 Concepts of Symbolics Networks **1**

1.1 Design Goals of the Network System 1
1.2 What is a Network? 2
1.3 What is a Network Service? 2
1.4 What is a File Server? 3
1.5 Concepts of Service, Medium, and Protocol 3
1.5.1 Common Protocols 4
1.6 Networks Supported by Symbolics Computers 6
1.7 Concept of Network Addresses 7
1.7.1 Setting the Chaosnet Address 7
1.8 Concepts of the Namespace System 8
1.9 Concept of Namespace Objects 9
1.10 Concept of **service** Attribute 9
1.11 A Sample Host Object in the Namespace Database 10
1.12 Glossary of Networking Terminology 10

2 Using the Network **13**

2.1 Commands That Use the Network 13
2.2 Activities That Use the Network 14
2.3 Using the Terminal Program 14
2.3.1 Connecting to a Remote Host over the Network 14
2.3.2 Connection Keywords in the Terminal Program 15
2.3.3 Dynamic Window Features of The Terminal Program 16
2.4 Using Peek to Get Information on Networks 16
2.5 Recovering From a Network Problem 17

3 Remote Login **21**

3.1 The Remote Login Capability for ASCII Terminals 21
3.2 Using the Remote Login Facilities for ASCII Terminals 22
3.3 Functions Used in Remote Login for ASCII Terminals 24

4 Network Addressing **27**

4.1 Format of Chaosnet Addresses 27
4.2 Format of Internet Addresses 27
4.3 Format of DNA Addresses 29
4.4 The Dialnet Subnets File 30
4.5 Choosing a Network Addressing Scheme 34
4.5.1 How to Obtain an Internet Address 34
4.5.2 Mapping an Internet Address into a Chaos Address 35

4.5.3 Mapping a Chaos Address into a DNA Address 36

5 Symbolics Generic Network System 39

5.1 Network Users and Servers 39
5.2 Service Attributes in the Namespace Database 40
5.3 Network Mediums 41
5.3.1 Generic and Specific Mediums 41
5.3.2 Descriptions of Defined Mediums 42
5.4 Generic Network Services 44
5.4.1 Protocols Supported by all Symbolics Computers as Users 44
5.4.2 Protocols Supported by all Symbolics Computers as Servers 45
5.4.3 TCP and UDP Protocols Supported by Symbolics Computers as Users 46
5.4.4 TCP and UDP Protocols Supported by Symbolics Computers as Servers 47
5.4.5 TCP and UDP Protocols Supported by SUN Computers as Servers 47
5.4.6 DNA Protocols Supported by Symbolics Computers as Users 48
5.4.7 DNA Protocols Supported by Symbolics Computers as Servers 48
5.4.8 Descriptions of Defined Generic Services 48
5.5 Enabling and Disabling Network Services 51

6 The Remote Procedure Call Facility 53

6.1 Overview of Symbolics RPC 53
6.1.1 Symbolics RPC and Sun RPC 54
6.1.2 Differences Between Local and Remote Function Calling 54
6.1.3 Basic Concepts of RPC 55
6.2 Symbolics RPC Facilities in Lisp 56
6.2.1 Extensions to Lisp Syntax for RPC 56
6.2.2 Macros for Defining RPC-based Programs 57
6.2.3 The RPC Data Representation Layer 64
6.2.4 Predefined RPC Data Types 65
6.2.5 The RPC Data Type Extension Language 70
6.2.6 Tracing RPC Transactions 77

7 Sync-Link Gateways 79

7.1 Hardware Requirements for Sync-Link Gateways 79
7.2 Configuring Sync-Link Gateways 79
7.2.1 Configuring Chaos-only Sync-Link Gateways 80
7.2.2 Configuring Sync-Link Gateways with Both Chaos and Internet 80
7.3 Bootstrapping Sync-link Gateways 81
7.3.1 Bootstrapping Sync-Link Gateways with a Single Namespace 82

7.3.2 Bootstrapping Sync-Link Gateways with Separate Namespaces 82

8 Software Interface to the Namespace System 85

8.1 Namespace System Lisp Data Types 85
8.2 Namespace System Variables 85
8.3 Namespace System Functions 86
8.4 Messages to Namespace Names and Objects 88
8.4.1 Messages to **neti:name** 88
8.4.2 Messages to **net:object** 88
8.5 Namespace Server Access Paths 89
8.6 Defining Namespace Classes 90

9 Interfacing to the Generic Network System 91

9.1 How a Network Service is Performed 91
9.2 Invoking Network Services 93
9.2.1 Service Access Path 93
9.2.2 File Access Path 94
9.2.3 Functions for Invoking Network Services 95
9.3 Defining a New Network Service 97
9.3.1 Example of Defining a Server Using Network Server Code 99
9.3.2 Summary of Functions for Defining Users and Servers 101
9.3.3 Functions for Defining Users and Servers 102
9.4 Finding a Path to a Service on a Remote Host 109
9.4.1 Finding a Path to a Local Service 109
9.4.2 Determining What Kinds of Connections a Symbolics Computer Can Make 110
9.4.3 Determining What Kinds of Connections a Remote Host Can Make 110
9.4.4 Finding the Possible Paths to a Host 111
9.4.5 Example of Finding a Path to a Host 112
9.4.6 Desirability of Network Protocols 115

10 Implementation of the Generic Network System 117

10.1 Packets 117
10.1.1 The Packet Pool 117
10.1.2 Functions Related to Packets 119
10.1.3 Subpackets and Coercing Packets 120
10.1.4 Example of Programming with Packets 122
10.2 Network Interfaces 125
10.2.1 Standard Communication with Interfaces 125
10.2.2 Sending a Packet to an Interface 126
10.2.3 Miscellaneous Notes on Interfaces 127
10.3 Implementation of Networks 127
10.3.1 Defining a Network 128
10.3.2 Implementation of Network Addresses 128

10.3.3 Invoking Mediums 129
10.3.4 Packet Reception 129
10.3.5 Packet Transmission 130
10.3.6 Initializing, Resetting, and Enabling Networks 130
10.3.7 Byte Stream Conventions 132
10.3.8 Interfacing to Ethernets 132
10.3.9 Interaction with Peek Network Mode 133
10.4 Implementation of Network Mediums 133
10.5 Implementation of the Service Lookup Mechanism 138
10.5.1 Summary of Functions for Service Lookup and Invocation 138
10.5.2 Functions for Service Lookup and Invocation 139
10.5.3 Messages Related to Service Lookup 141
10.6 Starting Network Servers 142
10.6.1 Finding a Server Description 142
10.6.2 Calling the Server Function 143
10.6.3 Functions Related to Starting Servers 144

11 Network, Medium, and Protocol Descriptions 147

11.1 Internet Networks 147
11.1.1 Introduction to Internet Networks 147
11.1.2 Internet Domain Names 148
11.1.3 References to IP/TCP Protocol Specifications 156
11.2 DNA Networks 157
11.2.1 Introduction to DNA Networks 157
11.2.2 References to DECnet Protocol Specifications 158
11.3 Dial Network Medium 159
11.4 BYTE-STREAM-WITH-MARK Network Medium 159
11.4.1 Introduction to BYTE-STREAM-WITH-MARK Network Medium 159
11.4.2 BYTE-STREAM-WITH-MARK Abortable States 161
11.4.3 Interfacing to the Lisp Machine Byte-Stream-With-Mark 163
11.5 Token List Transport Layer 165
11.5.1 Introduction to the Token List Transport Layer 165
11.5.2 Token List Stream 166
11.5.3 Token List Data Stream 174
11.6 NFILE File Protocol 177
11.6.1 Introduction to NFILE 177
11.6.2 Starting to Use NFILE 178
11.6.3 Reference Information on NFILE 178
11.7 Namespace Protocols 231
11.7.1 Network Namespace Protocol 231
11.7.2 Namespace Timestamp Protocol 232
11.8 Chaosnet 233
11.8.1 Introduction to Chaosnet 233
11.8.2 Overview of the Chaosnet Software Protocol 234
11.8.3 Technical Details of the Chaosnet Software Protocol 245

11.8.4 Application-Level Chaosnet Protocols 254
11.8.5 Using Foreign Protocols in Chaosnet 257
11.8.6 Symbolics Implementation of Chaosnet 259

List of Tables

Page
1 Translations Between Symbolics Characters and Standard ASCII 181
2 Translations in SUPER-IMAGE Mode 183

1. Concepts of Symbolics Networks

Networking capabilities are an essential part of Symbolics computers. Via networks, Symbolics computers communicate with each other and with different kinds of computers at a site. The goal of that communication is for one computer to *provide a service* for another computer. This allows a site to share its resources among the users at the site. For example, a network enables many users to share printers, tape drives, and disks. This reduces redundancy and often saves money.

This section gives a general description of what the networking capability provides and how to use the network services.

1.1. Design Goals of the Network System

In designing the network capability, Symbolics had three major goals. The network system should:

- Do its job automatically for the user.

  There is no special program to learn in order to use the network. Instead, you use familiar commands from the editor, Zmail, or the Command Processor; these commands use the network for you, when needed. Many commonly used commands and functions use the network, such as Show Users, Find File (m-X), Copy File, and Get Inbox. Many complex interactions must occur for these commands to do their jobs successfully , but they happen quickly, reliably, and automatically.

- Provide a way for programmers to deal uniformly with computers that run different operating systems and different networking software.

  For example, programmers use a single set of functions (such as **with-open-file**) to access files, whether they are stored locally or on a remote host, regardless of the type of operating system or networking software supported by that host. The same principle applies to programs such as the Mailer, the Terminal program, Tape, and so on.

- Be easily extensible by the programmer.

  The extensibility of the networking software should prove valuable to programmers who want to add new networking capabilities. The software is divided into layers of protocol so that programmers can build new applications on the foundation of a chosen layer of protocol. This unique design of a networking system is called the Symbolics Generic Network System: See the section "Symbolics Generic Network System", page 39.

1.2. What is a Network?

When a site has more than one computer, it is often desirable for the computers to be able to communicate. The goal of that communication is for one computer to perform a service for another computer, such as transferring files, sending mail, and so on.

A network consists of hardware and software that allow two or more computers to communicate with each other. The hardware provides a physical link and the software governs the communication. Computers that are connected by a network are often called *hosts*.

There are different types of computer networks, but they all have these things in common:

- Each host on the network must have a network address.
- One host must know the address of another host to communicate with it.
- Every host can communicate with any other host on the network.

Networks differ in three main ways:

- Types of services supplied.
- Format of network addresses.
- The way that data is transmitted from one host to another.

A "type of network" does not refer specifically to hardware. The hardware used by Symbolics computers is an Ethernet cable. One Ethernet cable can be used to support a Chaosnet network, an Internet network, or both.

1.3. What is a Network Service?

When computers are connected to a network, each computer gains certain new capabilities. That is, one computer is capable of performing a *service* for another computer. Here are some examples:

Name of Service	*Description of Service*
FILE	Ability to access files on a remote host.
MAIL	Ability to send electronic mail to a user on a remote host.
LOGIN	Ability to log in to a remote host.

SEND — Ability to send conversational messages to a user on a remote host.

HARDCOPY — Ability to hardcopy a file on a printer attached to a remote host.

The names of the services are in capital letters, as they appear in the namespace database. In other networking environments, the same services are called by different names.

1.4. What is a File Server?

One important capability of the network is that of transferring files from one machine to another. This capability is called *FILE service*. FILE service also enables a user to perform file operations on a remote host, such as copying and deleting files, probing for the existence of a file, listing and expunging directories.

A machine that provides FILE service to other machines is called a *file server*. Most sites designate one or more machines as file servers for all machines at the site. Users can store their files on the file server machine rather than on the Symbolics computer they use every day. This shared file system retains the traditional advantages of a timesharing system, such as:

- Users can have access to the same files and programs.
- Resources at the site are shared, such as disks.
- Users benefit from centralized backups and maintenance.

File servers can be Symbolics computers or any computer accessible via the network. For example, a timesharing computer such as a VAX can be a file server for the Symbolics computers at the site if it has the necessary hardware and software to be connected to the network.

Some other important servers are:

Print Server — A computer that is attached to a hardcopy device; it offers to print (or spool) files for other computers on the network.

Namespace Server — A Symbolics computer that stores the namespace database.

1.5. Concepts of Service, Medium, and Protocol

The Symbolics networking system has three important and related concepts: *service*, *medium*, and *protocol*. These are the conceptual layers of the networking software.

Service A capability provided by one host for another. Examples include: FILE, LOGIN, MAIL.

Protocol A particular high-level type of stylized dialogue supported by one computer that provides a particular service to another computer. Examples include: TCP-FTP, 3600-LOGIN, SMTP.

Medium A definition of what types of paths are adequate for providing a service using a particular protocol. Examples include: TCP, CHAOS, DNA.

Requesting a service and getting served involves a stylized dialogue between two hosts. The details of that dialogue are called the *protocol*. For example, if we imagine that food is a service, the protocol for requesting food is different if you are at a restaurant or a vending machine:

RESTAURANT protocol for FOOD service	VENDING-MACHINE protocol for FOOD service
1. Ask for food.	1. Pay with coins.
2. Receive food on plate.	2. Press buttons.
3. Pay with cash or credit.	3. Receive food in package.

A *medium* is a system underlying the protocol; it defines the low-level details of the communication. We can extend the food analogy to include mediums: You can request food service using restaurant protocol with the IN-PERSON medium or the TELEPHONE medium.

The concepts of service, medium, and protocol are described in more detail elsewhere:

See the section "Concept of **service** Attribute", page 9.
See the section "Service Attributes in the Namespace Database", page 40.
See the section "How a Network Service is Performed", page 91.

1.5.1. Common Protocols

Internet supports several levels of protocol. The lowest level protocol provides for transport of datagrams (raw blocks of data), and serves as the skeleton for which all other Internet protocols are built:

IP (Internet Protocol)
Defines the essentials for datagram transport including a standard address format, fragmentation/reassembly of data not fitting into a single packet, and packet checksum format. This protocol does not provide flow of control, retransmission, or error control.

The next level of protocol provides minimal routing, flow of control, and error control:

ICMP Provides minimal routing, flow of control and error control.

You can use the following protocols for various administrative functions:

- ARP (Address Resolution Protocol)
  Enables hosts to obtain a hardware (Ethernet) address corresponding to a known protocol address (for example, Internet or Chaos address).

- EGP (Exterior Gateway Protocol)
  Used for exchanging routing information between gateways.

- IGMP (Internet Group Multicast Protocol)
  Provides the tools necessary for defining multicast addresses and for associating a host with a multicast address.

- RARP (Reverse Address Resolution Protocol)
  Enables hosts to determine a protocol address (for example, Internet or Chaos address) corresponding to a known hardware (Ethernet) address.

You can use the following host to host data transmission protocols as a basis for application specific protocols:

- UDP (User Datagram Protocol)
  A minimal transaction-oriented protocol providing source and destination port addressing. This protocol distinguishes different application protocols, and also differentiates multiple connections between hosts. UDP does not protect against duplicate or out-of-order packets, or transport failure.

- TCP (Transmission Control Protocol)
  Provides process-to-process, end-to-end data transfer between hosts. TCP supports sour ce and destination ports (sockets), ordered transmission of data, and provides flow of control.

The most common IP/TCP protocols are:

- Domain Name Protocol
  Provides name service for Internet hosts. Two forms exist;

  ° Domain Simple
    Used over UDP.

  ° Domain
    Used over TCP.

- File Transfer Protocol
  A general file transfer mechanism using TCP; referred to as TCP-FTP by the Symbolics implementation.

- Simple Mail Transfer Protocol
  The format for electronic mail using TCP.

- TELNET
  Remote login protocol based on TCP.

1.6. Networks Supported by Symbolics Computers

A computer *supports* a type of network if the computer has the hardware and software required for that network. The hardware that physically links Symbolics computers together is called an *Ethernet*. This is a coaxial cable of the type used for cable television. Each Symbolics computer has a hardware interface to the ethernet. In the following model, a site has four Symbolics computers connected to an Ethernet:

```
Mickey  Donald   Minnie   Pluto  <- Symbolics computers
  |       |        |       |    <- hardware interface
==============================  <- the Ethernet
```

A Symbolics computer can be on many different networks even if it has only one hardware interface; it requires the software to support the different networks. Symbolics computers have the software (some of which is purchased separately) to support the following types of networks:

Chaos
: All Symbolics computers support Chaosnet, which was originally developed at M.I.T.

Dial
: Any Symbolics computer with a modem can support Dialnet, the international telephone network. This is the only network that does not use Ethernet hardware; it uses a modem and the existing telephone network. The function of Dialnet is to provide a reliable transport medium over possibly unreliable common carrier facilities. The primary uses of Dialnet are mail transfer and remote login.

Internet
: Symbolics makes available optional software (the IP/TCP software package) that enables Symbolics computers to support Internet networks.

DNA
: Symbolics makes available optional software (the Digital Network Architecture software package) that enables Symbolics computers to support DECnet, also called DNA.

SNA
: Symbolics makes available optional software (the System Network Architecture facility) that enables Symbolics computers to support a subset of SNA capabilities.

Typically, Symbolics computers use Chaosnet to communicate with one another. When a site has other kinds of computers, often those computers are already connected to a network, such as an Internet or a DECnet network. The optional software packages enable the Symbolics computer to be connected to the network already in use at a site.

1.7. Concept of Network Addresses

Each host on a network needs a unique *network address*. The network address is an identifier for the host. For example, when an electronic mail message is sent over the network, the sending host must include the network address of the destination host in the message. When you send a letter through the postal system, you write an address on the envelope for the same purpose.

One computer can support two different networks (for example, Chaosnet and Internet), if it has the necessary software. Such a host needs to have both a Chaosnet address and an Internet address.

The following examples show typical addresses on different networks:

Chaosnet	402
Internet	192.10.41.21
Dialnet	16175551234
DECnet	3.1

For one host on the network to communicate with another, it must know or be able to find out the address of that host. This information is stored in the *namespace database*. For an introduction to the namespace database: See the section "Concepts of the Namespace System", page 8.

For more detailed information: See the section "Network Addressing", page 27.

1.7.1. Setting the Chaosnet Address

Every Symbolics computer on a Chaosnet needs to set its Chaos address in its boot file. This is a line resembling:

```
Set Chaos-address octal-value
```

The default value of *octal-value* is the previous Chaosnet address, which is set to zero when the FEP is started.

The FEP checks for an acceptable Chaosnet address before starting Lisp. If none is specified as argument to this command, it warns you, asks whether the current setting is acceptable, and allows you to change it if necessary.

1.8. Concepts of the Namespace System

When computers are connected by a network to form a distributed computing environment, the computers should all be able to share information that describes that environment. The type of information typically needed by computers on a network includes:

- The names of other computers with which they can communicate
- The network addresses of those computers
- What printers are available on the various server computers
- Which host stores the mailbox for a particular user

Most network implementations have some method for storing and updating such information; in general, this is called a network database. The Symbolics implementation of a network database is called the *namespace database*.

The namespace database is maintained by a computer designated as the *namespace server*. Only Symbolics computers can be namespace servers.

All computers on the network can query or make changes to the namespace database by communicating over the network with the namespace server. The *namespace editor* is the tool for viewing and altering information stored in the namespace database. You can invoke the namespace editor by choosing it from the System menu, or giving the command Edit Namespace Object.

The database is structured to understand that there can be many different networks in a distributed environment. 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.

Summary of Namespace Terminology

namespace database
: The Symbolics implementation of network databases.

namespace server
: The computer on which the namespace database is stored.

namespace system
: The namespace database itself and the tools to use it.

namespace editor
: The tool used to view or alter objects in the namespace database.

1.9. Concept of Namespace Objects

The namespace database consists of a collection of *objects*. The namespace database has several different kinds of objects for different purposes. For example, the namespace database has a *host object* for each host on the network.

Examples of Namespace Objects

host object	Contains information on a computer on the network, such as its name, its network addresses, and the services it provides.
user object	Contains information on a user of the network, such as the user's login name and mail address.
printer object	Contains information on a printer connected to the network, such as the printer's name, its type, the host to which it is attached, and the options it supports.

1.10. Concept of service Attribute

Each service is implemented on a network medium using a protocol. Hosts that are on two networks can often provide a service over two network mediums using two different protocols. For example:

On Internet:

FILE service is implemented on the TCP medium using the TCP-FTP protocol. Often FILE service is implemented also on the UDP medium using the UDP-FTP protocol.

On Chaosnet:

FILE service is implemented on the CHAOS medium using the NFILE protocol.

The namespace database stores information on the services, media, and protocols that each host supports. The information is stored in the service attributes of each host object. A service attribute has three parts: the service, the medium, and the protocol. If you view a host object, you might see these entries:

```
Service: Set: FILE CHAOS NFILE
Service: Set: FILE TCP TCP-FTP
```

When one computer needs a service from another computer, it consults the namespace database to determine:

- Does the computer provides the requested service?
- What is the best route to get that service? The medium and protocol are part of the route.

Finding a path to a host can be a complicated procedure, but it is all done automatically by the Symbolics Generic Network System. The necessary information is stored in the namespace database, and the namespace system provides tools that use the information to find the best route. For more information: See the section "Finding a Path to a Service on a Remote Host", page 109.

For more details on how services are requested and performed: See the section "How a Network Service is Performed", page 91.

For more details on service entries: See the section "Service Attributes in the Namespace Database", page 40.

1.11. A Sample Host Object in the Namespace Database

By viewing a sample host object in the namespace database, many of the concepts of Symbolics networking become clearer. Host objects can contain much more information than shown here; however, this example illustrates the most important attributes of a host object.

```
System Type*: LISPM
Machine Type: 3600
Address:  Pair: CHAOS 24460
Address:  Pair: INTERNET 192.10.41.48
Service: Set: FILE TCP TCP-FTP
Service: Set: FILE CHAOS NFILE
```

This host is a Symbolics 3600-family computer that is on two networks: Chaos and Internet. The host therefore has two network addresses.

The **service** attributes show that this host can provide FILE service in two ways: across the Internet network (using the TCP medium and TCP-FTP protocol), and across the Chaos network (using the CHAOS medium and the NFILE protocol).

1.12. Glossary of Networking Terminology

This section gives brief definitions of the terms used frequently in the networking documentation.

Host	Used interchangeably with *computer* and *machine*. Examples are: Symbolics computers and VAX computers.
Machine	Used interchangeably with *computer* and *host*. Examples are: Symbolics computers and VAX computers.
Medium	Defines how one computer can provide a service using a given protocol; that is, defines what type of paths are adequate for a given protocol. Examples are: TCP, CHAOS, DNA.

Namespace database
: The Symbolics implementation of network databases.

Namespace editor
: The set of tools used to view or alter objects in the namespace database.

Namespace server
: The computer on which the namespace database is stored.

Namespace system
: The namespace database itself and the tools to use it.

Network
: The hardware and software that enables two computers to communicate. The goal of that communication is for one computer to provide a service for the other computer.

Network type
: There are many different types of networks; each type has a designated way of transmitting data, format of network addresses, and types of services supplied. Examples are: Internet, Chaos, Dial.

Protocol
: A stylized dialogue between two computers that takes place when one computer requests a service from another computer. Examples are: TCP-FTP, 3600-LOGIN.

Service
: A capability that one computer provides for another computer on the network. Examples are: FILE, LOGIN, MAIL.

Site
: A collection of computers located in one small geographic location; usually the computers are connected to one another by means of a network. A site can also be a single computer; these sites have no need for a network. Examples: the Symbolics Corporate Research Center, ACME Corporation building 21.

User host
: A computer that requests a service from another computer on the network.

Server host
: A computer that provides a service to another computer on the network.

2. Using the Network

Symbolics designed the network to be used by commands, functions, and activities, instead of being invoked directly by a user. This section describes some of the commands and activities that use the network automatically, when needed.

The only time you need to do anything special to use the network is when logging in to a remote host. Then you use the Terminal program. See the section "Using the Terminal Program", page 14.

You can connect to a remote Symbolics computer from an ASCII terminal or another Symbolics computer. For more information: See the section "Remote Login", page 21.

2.1. Commands That Use the Network

The following Zmacs, Zmail, and Command Processor commands provide some examples of the use of the network. Many other functions and programs also use the network.

Show Users	This CP command requests the SHOW-USERS service from a given host on the network, or from all hosts reachable on the network.
Find File (m-X)	This Zmacs command (c-X c-F) requests the FILE service from the host on the network where the given file is stored. The file is copied from that host to an editor buffer.
Save File (m-X)	When you later save the file (c-X c-S), Zmacs again requests FILE service to copy the altered contents of the file from your editor buffer to the host on the network where the file is stored.
Mail	This Zmail command requests STORE-AND-FORWARD-MAIL service on the host where the recipient receives mail. STORE-AND-FORWARD-MAIL handles the mail delivery.
Get Inbox	Many Zmail commands use the network. When you use the Get Inbox command, Zmail requests FILE service from your mail host. Your inbox is copied from your mail host to a Zmail buffer.
Hardcopy File	This command requests HARDCOPY service from a print server. Your host sends the contents of the file to the print server, which in turn sends it to the printer.

2.2. Activities That Use the Network

The following activities use the network for you:

SELECT C	The Converse facility requests SEND service on one or more hosts on the network, to send your conversational message to its recipients.
SELECT T	The Terminal facility requests LOGIN service from the given host, enabling you to log in to that host over the network. See the section "Using the Terminal Program", page 14.
SELECT D	Document Examiner frequently requests FILE service from the host that stores the online documentation files. Commands like Show Candidates, Show Documentation, and Show Table of Contents make use of the network.

2.3. Using the Terminal Program

2.3.1. Connecting to a Remote Host over the Network

If your Symbolics computer is on a network and configured properly, you can access other hosts on the network with the Terminal program.

To use the Terminal program, press SELECT T. The prompt is:

```
Connect to host:
```

Type the name of the host to which you want to connect. The network system makes a connection, and you will see the prompt of the remote host displayed on the screen. You are now communicating directly with the remote machine.

When you are connected to a remote host, the NETWORK key provides several useful commands. For example:

NETWORK HELP	Displays the list of options for the NETWORK key.
NETWORK L	Logs out of the remote host, and breaks the connection.
NETWORK D	Disconnects without logging out first.

See the section "NETWORK Key" in *Genera Handbook*.

If you want to use the Terminal program to log in to a remote Symbolics computer when someone is logged in to that machine, you must first enable remote login by evaluating the form **(net:remote-login-on)** on that machine. See the function **net:remote-login-on**, page 24.

See the section "Connection Keywords in the Terminal Program", page 15. See the section "Dynamic Window Features of The Terminal Program", page 16.

Remote Terminal Commands

Set Remote Terminal Options
: Enables you to toggle MORE processing on and off. Additionally, you can specify whether a status line appears at the bottom of your screen, and also how often the status line updates.

Show Remote Terminal Options
: Enables you to view the current settings of your remote terminal options. This command also displays the height and width of your screen.

Halt Remote Terminal
: Enables you to halt your remote terminal.

2.3.2. Connection Keywords in the Terminal Program

In most cases you need only enter the name of the host to the "Connect to host" prompt in the Terminal program. However, there are optional keywords that let you further specify some aspect of the connection. These keywords include:

:Login protocol
: The name of the protocol to use to interpret output from the remote host. If this keyword is not supplied, a protocol is chosen automatically by the Generic Network System. You can enter any protocol for LOGIN service defined on your host. Some examples are: TELNET, SUPDUP, 3600-LOGIN, CTERM. The HELP key lists the LOGIN protocols defined on your host.

:Connection Protocol
: The name of the protocol to use to establish the connection. If this keyword is not supplied, a protocol is chosen automatically by the Generic Network System, and it will be a protocol for LOGIN service. The HELP key lists the connection protocols defined on your host. You can specify a protocol for a service other than LOGIN if you want to debug that server.

:Echo
: If yes, echo all characters locally. If no, let the remote host echo the characters. The default is no.

:Overstrike
: If yes, when the host outputs a backspace and you type another character in its place, the second character overstrikes the first. This behavior is similar to that of a printing terminal. If no, the backspace erases characters instead of overstriking them.

:Terminal Simulator
: Specifies the name of a terminal you wish to emulate. The choices are: VT100, Ambassador, IMLAC, and Glass TTY.

:Wallpaper File
: The pathname of a file to which output should be sent. This is sometimes called a journal file. By default there is no wallpaper file.

2.3.3. Dynamic Window Features of The Terminal Program

The Terminal program offers some Dynamic Window features. First, the window is scrollable, so you can scroll forward and backward over the history of input and output that has appeared on the screen. Second, you can mark a region of the screen and do something to it, such as: enter the marked region as input, save it on the kill ring, or hardcopy it.

The Terminal program does not use presentation types, nor does it support features of Dynamic Windows that use the SUPER or META keys. For example, m-W, m-V, and m-SCROLL do not work in the Terminal program.

Marking and Using Regions:

The marking features are available with the CONTROL key pressed down. If you press CONTROL, the mouse documentation line shows which commands are available. Typically you first mark a region of the screen and then do something with that region. To mark a region, position the cursor at the beginning of the region of interest. Press c-Left and move the cursor to the end of the region of interest. The marked region is underlined. Press c-Right for the *Marking and Yanking Menu*, which lists the things you can do with the region.

Entering a Region as Input:

You can position the cursor over a single word (not separated by spaces) and press c-Middle to enter that word as input. If you want to enter a longer command that is separated by spaces and lines, mark the region, click c-Right for the menu, and choose [Yank Marked Text].

Using the scroll bar:

The scroll bar on the left side of the screen allows you to scroll backward and forward. Any cursor motion or graphics display occurs relative to the current viewport.

When you are using the TELNET or CTERM protocol, all input and output history is saved, and you can access it by scrolling. Most terminal connections to UNIX hosts use TELNET, and connections to VAX/VMS hosts use CTERM.

Note that when you are using the 3600-LOGIN or SUPDUP protocol, or emulating a VT100 terminal, only one screenful of input and output is saved, so you cannot scroll backward or forward. Most terminal connections between two Symbolics computers use 3600-LOGIN.

2.4. Using Peek to Get Information on Networks

The Peek facility displays and updates status information on various aspects of the network. The best way to find out what information Peek offers is to experiment with it. Press SELECT P.

Peek has four network-related options: [Networks], [File System], [Servers], and [Hostat]. [Networks] and [File System] are the most interesting. Click on one of

those headings at the top of the screen. When you move the mouse over the different parts of the display, the mouse documentation line offers options that are appropriate to that mouse-sensitive area of the screen.

For more information: See the section "Using Peek" in *Genera User's Guide*.

2.5. Recovering From a Network Problem

In general, the symptom of a network problem is the inability of your Symbolics computer to communicate with other hosts on the network. This section describes how to recognize some common network problems, some possible causes of them, and suggestions for solving the problem.

In brief, the first step is to isolate the problem. A network problem could be a problem in the software or hardware of your local machine, the software or hardware of the remote machine, the information stored in the namespace database, or the hardware of the network itself. The Reset Network command is useful for resetting the network software in your machine, but it cannot solve any problems in the remote host, the network itself, or the hardware.

Once you have located the problem, you can take steps to solve it. If the problem is the remote host, the namespace, or the network itself, you should probably consult with your Site Administrator for help.

Symptoms of Network Problems

- File transfer is stuck or slow.

  When a file is being transferred, the pathname is displayed in the bottom right corner of the screen, along with the number of bytes and the percentage of the file that has been copied. If the percent and byte-count figures do not change, the file transfer seems to be stuck.

  The local program might be running slowly. If the status line is in Run state, at least you know that the program is running. Another possible cause of a stuck or slowed-down file transfer is that the server on the remote host is responding slowly. It is also possible that the network is highly congested. In any of these cases, little can be done other than just waiting.

  If the file transfer remains stuck for a long time (several minutes), sometimes the connection is broken and you are offered some choices in a debugging menu. You can choose to restart or abort the file transfer.

  A hardware problem could also halt a file transfer. See below.

- Broken Terminal connection.

When you are using the Terminal program and are connected to a remote host, the connection can be broken. An error message is displayed, and the prompt "Connect to host:" is redisplayed. This can happen if the remote host goes down unexpectedly or for scheduled maintenance, or if someone resets its network interface. Similarly, if you give the Reset Network command, this would break all your network connections. Once the connection is broken, the only thing you can do is try to open another connection by answering the Terminal prompt with the name of the desired host. If you cannot log in to that host, you should check with the Site Administrator for that host to see if there is a problem with that host.

A hardware problem could also break a Terminal connection. See below.

- No network operations work successfully.

  Occasionally, you will notice several problems with network-related tasks. For example, a file transfer gets stuck, the Terminal program stops responding, and you cannot queue a file to a printer. To test the network software on your host, give some simple commands, such as Show Users and Show Hosts for several hosts. If you do not get the expected response, it is possible that the network software is somehow compromised. You can give the Reset Network command. This resets much of the networking software, breaks any outstanding network connections, and restarts the network again. Once you have done this, try the Show Users command again.

  It is also possible that the network itself is causing the problems. Check with other users at the site to see if they are also having trouble with network operations. If so, the problem probably lies in the network itself.

  If other users are not having problems, but your host still cannot communicate over the network, it is probably a hardware problem specific to your host. One common cause of this is the transceiver cable somehow falling out of the back of the Symbolics computer. If this has happened, plug it in again. If the network does not immediately work, use the Reset Network command.

- Error message: *Host* does not have services enabled.

  Sometimes the remote host is up and running, but does not have its network services enabled. This is often true when a host is just coming up and is not yet fully initialized. It is also possible that a user of that host has decided to disable services. You can either wait and try again later, or call the host's Site Administrator to see why services are not enabled.

  This symptom does not indicate a hardware problem.

- Error message: *Host* does not support this service.

This error message indicates that the target host does not support the network service you requested. Sometimes the network system offers to try another protocol for the same service; you can try that. In a heterogeneous networking environment, there are some services that you cannot obtain from some hosts.

It is also possible that the host does have the capability of performing that service, but the information in its host object in the namespace is incorrect. You can ask the person who is responsible for maintaining the namespace database if that is the case.

This symptom does not indicate a hardware problem.

Hardware Problems

A hardware problem usually halts all network operations. There are two categories of problems: a problem that is isolated to your machine, and a problem that affects all users of the network.

If only your machine is affected, the first thing to check is that the transceiver is properly connected to the back of your machine. If it has been dislodged, plug it in again. If it is properly connected and the network still does not work, the transceiver hardware might be the problem.

If the whole site is affected, the cause of the problem could be one of these:

- Ethernet cable is not terminated at both ends.
- Ethernet cable is broken in the middle.
- Ethernet cable is shorted.
- A network host is jamming the cable by transmitting continuously.

3. Remote Login

3.1. The Remote Login Capability for ASCII Terminals

The remote login facilities allow up to four ASCII terminals to be connected directly via a Symbolics computer's serial ports. See the section "The Serial I/O Facility" in *Internals*.

Also, 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 3600-LOGIN, TELNET, and SUPDUP. 3600-LOGIN is used only in communication between two Symbolics computers. TELNET and SUPDUP are standard protocols used on the Internet.

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

- Lisp Listener (not a Dynamic Window)
- Input editor
- Debugger (not the Display 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:

See the section "Using the Remote Login Facilities for ASCII Terminals", page 22.

See the section "Functions Used in Remote Login for ASCII Terminals", page 24.

3.2. Using the Remote Login Facilities for ASCII Terminals

This section discusses how to prepare to use the remote login facilities for ASCII terminals. The server host is the Symbolics computer to which you want to connect remotely.

Preparing the Server Host for Remote Login

If the server host has no user logged in, there are no restrictions on logging into it from a remote terminal. However, if a user is logged in, remote login connections are rejected by default. To change this, use the function **net:remote-login-on** on the server host. You cannot do this step remotely; you must evaluate that form on the server host itself.

Editing the Namespace

If you are not connecting via the serial line, you need to decide which generic network service, medium, and protocol you want to use. Edit the host object of the server host to add the appropriate service attribute.

To connect to a Symbolics computer from another Symbolics computer, the service attribute is one of these:

```
Service: LOGIN CHAOS 3600-LOGIN
Service: LOGIN TCP 3600-LOGIN
```

To connect to a Symbolics computer from a terminal attached to a host that is on the same network as the Symbolics computer, or from a terminal attached to a terminal concentrator that is on the network, you need to know which protocol the host or terminal concentrator uses. These are the possibilities:

```
Service: LOGIN CHAOS SUPDUP
Service: LOGIN CHAOS TELNET
Service: LOGIN TCP TELNET
Service: LOGIN TCP SUPDUP
```

Preparing to Connect via a Serial Line

To use a terminal connected via a serial line, use the function **neti:enable-serial-terminal** on the server host. There is no need to edit the namespace database when connecting directly to a serial line.

Describing the Characteristics of the Terminal

This step is required when you use the TELNET protocol or the serial line. You need to use either the function **neti:ask-terminal-parameters** or the function **neti:set-terminal-parameters** on the server host to describe the terminal. (If the terminal automatically echoes a newline when a character is printed in the rightmost column, then decrement the width by one.)

When the SUPDUP or 3600-LOGIN protocol is used, terminal information is communicated automatically.

Additional Notes

- The SUPDUP server works only if the terminal supports character insertion and deletion.

- Only one interactive process is allowed per remote terminal.

- If you are logging in from an ASCII keyboard, an escape prefix exists to allow you to refer to Symbolics special function keys that do not exist on an ASCII keyboard. The special keys are typed as single characters following the escape, which is ascii code 31. Different keyboards have different schemes for typing ascii code 31. c- is a common one. Others include c-←, c-?, and c-→. The single characters to send the special function keys are:

```
H  HELP
E  END
A  ABORT
S  SUSPEND
R  RESUME
C  COMPLETE
I  CLEARINPUT)
X  ESCAPE
L  LINE
P  PAGE
F  REFRESH
B  BACKSPACE
N  NETWORK
1  SQUARE
2  CIRCLE
3  TRIANGLE
```

- If you are logging in from one Symbolics computer to another, the keyboard operation is identical except that when you use these keys, they are not transmitted through to the server:

```
NETWORK
LOCAL
FUNCTION
SELECT
```

Set Remote Terminal Options Command

Set Remote Terminal Options

Prompts for the options to set up a remote terminal. On a serial terminal, keyword arguments to **neti:enable-serial-terminal** are used to determine reasonable defaults.

Set Remote Terminal Options is only available from a remote terminal.

3.3. Functions Used in Remote Login for ASCII Terminals

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

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	Allows remote login connections even when the main console is in use.
nil	Rejects remote login requests.
:notify	Allows remote login requests but send the main-console user a notification.

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 do not change.

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 do not change.

neti:enable-serial-terminal &rest *options* &key *(:top-level* **'si:lisp-top-level1***) (:herald* **t***) :x3.64 (:width* **79***) (:height* **1073741824***) (:unit* **1***) :share-kill-history :status-line-p (:status-line-update-frequency* **300***)* &allow-other-keys *Function*

Allows an ASCII terminal to communicate with a Symbolics computer process through one of the machine's serial ports (specified by the *unit* argument). *unit* can be **1**, **2**, or **3** to indicate one of the bulkhead ports (these are DTEs); or **0** to indicate the serial I/O port located at the back of the console (a DCE). For more information on the serial I/O ports: See the section "The Serial I/O Facility" in *Internals*.

The argument **:x3.64** specifies whether the terminal supports escape sequences meeting this ANSI standard. **:width** and **:height** are the terminal's width and height in characters, respectively. If you supply **nil** for height and width, the current settings do not change. *top-level* specifies the process. **:herald** specifies whether the herald is displayed on the terminal. **:status-line-p** specifies whether or not to display the status line and **:status-line-update-frequency** controls the frequency of updates on the status line in sixtieths of a second.

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.

neti:disable-serial-terminal *unit* *Function*

Kills the Genera process associated with a terminal connected to a serial port, closes the stream, and clears the serial port so it can be used again. *unit* specifies the serial port to which the terminal is connected. *unit* can be **0**, **1**, **2** or **3**.

Communication between the terminal and the Symbolics computer is begun with the **neti:enable-serial-terminal** function.

Sample use:

```
(neti:disable-serial-terminal 2)
```

4. Network Addressing

This section describes the format of Chaosnet addresses, DNA addresses, and Internet addresses.

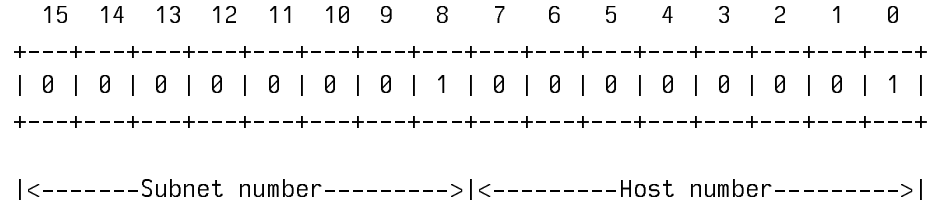
We propose that all sites choose network addresses for their hosts with the perspective that they might eventually support another type of network, or connect to another existing network. Thus we recommend coordination among sites that might later be connected via a gateway. We also propose a scheme for choosing DNA and Chaosnet addresses based on a valid Internet address.

The recommendations are described at the end of this section: See the section "Choosing a Network Addressing Scheme", page 34.

4.1. Format of Chaosnet Addresses

A Chaos address is a 16-bit quantity, in which the high-order 8 bits represent the subnet number, and the low-order 8 bits represent the host number on that subnet. Neither the subnet number nor the host number can be zero. Chaos addresses are expressed in octal.

Example: Chaos Address 401



The subnet number is 1.
The host number is 1.
The Chaos address is 401 octal.

For technical details on how the Chaosnet address is used: See the section "Chaosnet Addresses and Indices", page 235.

4.2. Format of Internet Addresses

Internet addresses are expressed in decimal, in four octets separated by periods. Each octet is 8 bits long. There are three kinds of Internet addresses: Class A, Class B, and Class C.

Examples of Internet addresses:

- 10.2.0.7 is host 2.0.7 on Class A network 10.
- 139.41.0.3 is host 0.3 on Class B network 139.41.
- 192.10.0.200 is host 200 on Class C network 192.10.0.

Note that the host number cannot be zero or 255, because those are considered broadcast addresses.

Interpreting Internet Addresses

Internet addresses consist of network and host fields. The network field identifies the network, and the host field identifies the host on that network. This size of the Internet address depends on the address and the configuration of the network.

You can use a subnet field for networks containing subnets. Using a subnet field divides the address into three fields. A subnet mask determines the bits used for selecting a subnet. Note that the rules for determining a subnet field vary for each network.

Note: You cannot fill a field (network, subnet, or host field) with all zeros or ones for representing a network, subnet, or host.

Class A Addresses

A Class A Internet address is a 32-bit number, in which the high-order octet (8-bits) represents the network number and the following three octets represent the host number. The first octet is less than 128.

Example of Class A Internet Address: 10.2.0.7

```
+--------+--------+--------+--------+
|00001010|00000010|00000000|00000111|
+--------+--------+--------+--------+

|<-net-->|<--------host------------>|
```

Class B Addresses

A Class B Internet address is a 32-bit number, in which the two high-order octets represent the network number and the following two octets represent the host number. The first octet of a Class B network is greater than or equal to 128 and less than 192.

Example of Class B Internet Address: 139.41.0.3

```
+--------+--------+--------+--------+
```

```
|10001011|00101001|00000000|00000011|
+--------+--------+--------+--------+

|<---network----->|<-----host------>|
```

Class C Addresses

A Class C Internet address is a 32-bit number, in which the three high-order octets represent the network number and the low-order octet represents the host number. The first octet of a Class C network is greater than or equal to 192, and less than 224.

Example of Class C Internet Address: 192.10.0.200

```
+--------+--------+--------+--------+
|11000000|00001010|00000000|11001000|
+--------+--------+--------+--------+

|<-------network---------->|<-host->|
```

Internet Subnet Number

The Internet subnet number is the Internet address resulting from replacing the host field of an Internet address with zeros. You can determine the Internet subnet number of a network by determining the class of a host address and replacing the host portion of the address with zeros. For example, the class B address 128.81.38.232 corresponds to the Internet Subnet number 128.81.0.0.

Subnet Masks

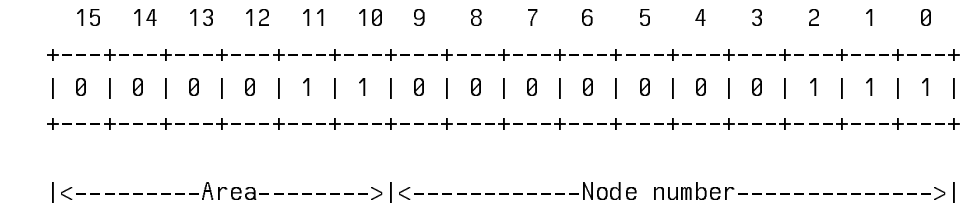
A subnet mask determines the field of the Internet address specifying the subnet on the network. A subnet mask is a 32-bit quantity containing one in every bit corresponding to the official Internet subnet number. Additionally, the subnet mask contains a zero in every bit selecting a host on a subnet. For example, A class B network (128.81.0.0) is broken into many subnets, using the third octet of the address for selecting a subnet. The Class B default mask is 255.255.0.0; since the third octet determines a subnet, you have to fill it with ones. The resulting subnet mask is 255.255.255.0.

4.3. Format of DNA Addresses

DNA addresses have two components: an area and a node number in that area. For example, a DNA address of 3.7 indicates the host is node 7 in area 3. Hosts with different area numbers cannot communicate with each other.

DNA addresses are 16 bit quantities, where the high-order 6 bits constitute the area, and the low-order 10 bits constitute the node number. DNA addresses are expressed in decimal notation.

Example: DNA Address 3.7



Bits 0-9 represent the node number, in this example 7.
Bits 10-15 represent the area number, in this example 3.

You can choose DNA addresses for your hosts in any way you like, as long as:

- Each host that will use DNA protocols, whether the machine is a VAX or a Symbolics computer, has a valid and unique DNA address.
- Any two hosts that want to communicate with each other are in the same area. For example, the Symbolics computer area numbers must be the same as the area number for any VAX that is a server machine.
- The area number is in the range of 1 to 63 inclusive.
- The node number is in the range of 1 to 1023 inclusive.

Some sites choose to assign DNA addresses sequentially, from 1.1, 1.2, 1.3 and so on.

4.4. The Dialnet Subnets File

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.

The mailer always identifies Dialnet hosts by their fully-specified addresses, meaning that the address is represented by its country code, area code, exchange, and so forth. Any given Dialnet address has only one fully-specified form, unique worldwide, regardless of the local conventions for how one dials the phone to connect to that address.

It is not generally appropriate to 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 Subnets file is used to tell the mailer, for a given telephone number, what actual number to dial in order to connect to that number.

There can only be one Dialnet subnets file at any given site, called SYS: SITE; SUBNETS.LISP. This file consists of some number of Lisp forms. Each form is always a list of alternating keywords and values like this:

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

All three keywords must appear, and they must appear in this order. No other keywords are accepted.

The attribute after the **:subnet** keyword specifies a pattern that must match in order to consider the rest of the particular form. If the match succeeds, the actual telephone number that the modem should dial is described by the attribute after the **:dial** keyword, which may contain modem control characters as well as pattern-matching characters and literal digits. Finally, the attribute after the **:cost** keyword specifies how expensive this call is, and is used to select the cheapest way to route the call if more than one of the **:subnet** patterns matches.

:Subnet Keyword in the Dialnet Subnets File

The Mailer may know the Dialnet addresses for a large number of hosts. It is not necessary to specify every possible binary combination of world-wide phone exchanges and their associated prefixes in order for the mailer to know how to dial the phone. Instead, the attribute following the **:subnet** keyword in the subnets file 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 represent 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 clarifies this. 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, as is 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 the one with the most minimal 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.

:Dial Keyword in the Dialnet Subnets File

The attribute following the **:dial** keyword in a subnets file is used if the pattern match in a **:subnet** attribute succeeds. This **:dial** attribute is a sequence of digits, letters, and punctuation. Digits in this attribute are simply dialed literally. Non-digits are more complicated, and may either stand for digits in the number to be dialed, or for modem control characters.

For example, suppose we are calling from 16175777348 to 14155200142. Given the subnet pattern 1xxxyyyyyyy>1zzzwwwwwww, we would get the successful matches:

```
xxx      617
yyyyyyy  5777348
zzz      415
wwwwwww  5200142
```

The **:dial** attribute corresponding to this pattern match might be 91zzzwwwwwww. The non-digits in this attribute will be filled in from the values obtained from the pattern-match of the **:subnet** attribute, meaning that we will actually dial 914155200142. We would do something like this if the modem went through a PBX that required dialing 9 to get to an ordinary outside telephone line.

Some telephone systems, such as PBX's, may require you to dial a number to get to an outside line, wait for a second dial tone, and then continue dialing. Many modems support this sort of dialing by allowing you to embed punctuation characters in the string of numbers to dial which cause the modem to take some special action.

To allow you to specify this, if you specify characters in the **:dial** attribute that are not matched by the right side of the **:subnet** attribute, those characters will be sent literally to the modem, rather than eliciting an error message. (Before Genera 8.0, unmatched characters in the right side would cause an error message.) For example,

```
(:subnet "1xxxyyyyyyy>1xxxzzzzzzz" :dial "T9,WPzzzzzzz" :cost "0")
```

might tell a Hayes modem that, in order to dial an outside number from a PBX that is in the same area code as the number to be dialed, it must DTMF-dial a 9, wait for a second dial tone, then pulse-dial the rest of the number.

Note that because unmatched non-digits in the **:dial** attribute will be sent directly to the modem instead of causing an error, typographical errors in this attribute are difficult to catch.

:Cost Keyword in the Dialnet Subnets File

The attribute following the **:cost** keyword should be a small integer which somehow reflects the cost of the call in some convenient metric. Typically this is related to how expensive it is to make the call. If more than one pattern matches a particular address, Dialnet uses the match with the lowest cost. This typically comes into play when an 800 number matches some address that is also matched by a "normal" long-distance line. If there is only one way to reach the given number (only one pattern matched), the cost is ignored.

Here is an example of a typical subnets file:

```
;;; -*- Mode: Lisp -*-

(: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, you do not have to use a 1 prefix for long-distance calls.

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.

6. Note that a typical subnets file that may already be suitable for your telephone system is included in SYS: DIALNET; PROTOTYPE-SUBNETS.LISP. (This is not distributed as SYS: SITE; SUBNETS.LISP, which is where the data must eventually be stored, because it may not be correct for your site and the consequences of misdialing can be expensive.)

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.

4.5. Choosing a Network Addressing Scheme

This section proposes a scheme for convenient handling of network addresses in a multi-networking environment, and recommends coordination among sites that might in the future be connected via gateways. It is not necessary or required that you follow the suggestions in this section.

The primary intent of this section is to advise site administrators to consider the possibility that the site might want to connect to an existing network, or support another type of network sometime in the future. Many sites already support more than one type of network. Some sites support Chaosnet and Internet networks; other sites support Chaosnet and DNA networks.

A standalone site can set up the network addressing in such a way that the transition to a larger networking environment will go smoothly in the future. For example, consider the requirement that each network host (for Chaosnet, DNA, or Internet types of networks) must have a unique address. If your site intends to connect to another existing network, it is to your advantage to coordinate with the site administrator of that network to ensure that no two hosts on either network have the same address. This type of coordination would obviate the need for changing the network addresses of hosts when the two networks become connected.

We also recommend choosing network addresses by a scheme of mapping one type of address into another, such that if you know the Internet address of a host, you can derive its Chaosnet address and vice versa. We propose a similar mapping between Chaosnet and DNA addresses. When a site uses such a scheme, the site administrator has one method for assigning network addresses for hosts. This should reduce the complexity of assigning two or three types of addresses to each host.

As a general note, all sites might consider requesting a valid Internet address. If you set up your site based on a valid Internet address, it is unlikely that your addresses will collide with the addresses of other sites. You can receive a valid Internet address without being connected to the Internet. If your site ever does connect to the Internet in the future, the transition will go smoothly if your site is already using valid Internet addresses.

Once you have an Internet address, you can use the mapping schemes to derive a Chaos address and a DNA address based on the Internet address.

4.5.1. How to Obtain an Internet Address

If your site does not already have an Internet network number, you can request one by contacting:

Joyce Reynolds
USC - Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90292
(213) 822-1511
ARPANET: jkreynolds@usc-isi.arpa

The Internet address you receive is the network part of the address. You assign the host number part of the address yourself. Each host on the local network must have a unique host number.

4.5.2. Mapping an Internet Address into a Chaos Address

Once you have an Internet address for a host, you can map that address into a Chaos address. You can then assign sequential Chaos addresses for all Chaos hosts on the network. If you are on the Internet, you can use each host's Internet address to derive a Chaos address.

The mapping process is best explained by example. The following two examples show the mapping of a Class B and Class C Internet address into a Chaosnet address:

Class C Internet address: 192.10.41.48 decimal.

Step 1: Get the Chaos subnet number and host number.

```
192.10.41.48  is the Internet address.
192           is unused in the mapping.
    10        is unused in the mapping.
       41     is the Chaos subnet number.
          48  is the Chaos host number.
```

Step 2: Convert the decimal subnet and host numbers to octal.

The subnet number (41 decimal is 51 octal.)
The host number (48 decimal is 60 octal.)

Step 3: Insert subnet and host numbers into two eight-bit bytes.

```
  15  14  13  12  11  10  9   8   7   6   5   4   3   2   1   0
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

|<-------Subnet number--------->|<---------Host number--------->|
```

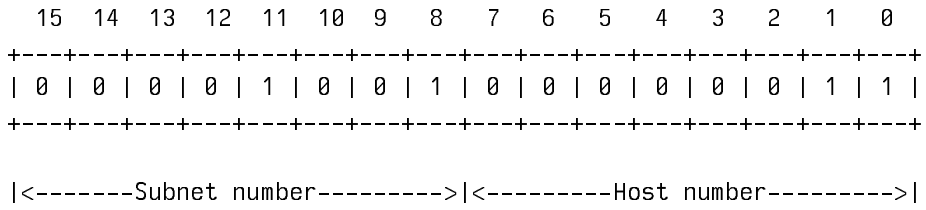
Step 4: Express the quantity in octal notation; this is the Chaos address.

```
0 010 100 100 110 000    (binary representation)
   2   4   4   6   0     (octal representation)
```

The resulting Chaos address is 24460 octal.

Class B Internet address: 139.41.9.3 decimal.

The subnet number is 9 decimal.
The host number is 3 decimal.



The resulting Chaos address is 4403 octal.

4.5.3. Mapping a Chaos Address into a DNA Address

We recommend that you choose DNA addresses for the hosts at your site based on the Chaos addresses. Each Symbolics computer already has a unique Chaos address. By choosing a DNA address derived from the Chaos address, you can always determine a DNA address from the Chaos address (thus assuring that the DNA address is unique), and you can derive the Chaos address from the DNA address.

It is not necessary or required that you derive DNA addresses based on the Chaos addresses. This is just a suggestion.

Some sites cannot use this mapping scheme. If your site has several VAX/VMS hosts that are already using DNA protocols, they already have DNA addresses assigned to them. In that case, you must be sure to assign DNA addresses to the Symbolics computers that have the same DNA area number as the VAX/VMS hosts on the network. These addresses must be unique within the DNA database.

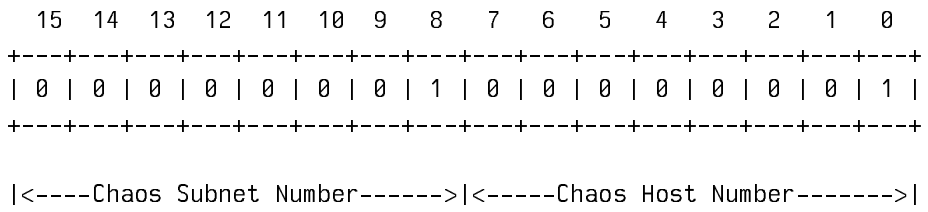
If you use this mapping scheme, keep in mind that the node numbers of each host must be below the VAX's limit, which is the MAX ADDRESS parameter of the NCP. The NCP does not accept network communication from hosts with node numbers higher than MAX ADDRESS. By default, MAX ADDRESS is 32. It is an easy matter to set the MAX ADDRESS higher.

Start by figuring out the Chaos address of the first host to have DNA installed on it. You can do this by entering the namespace database (choose it from the System menu): use [View], then use [Host], then enter the name of the host. Each Symbolics computer host object should contain a Chaos address (expressed in octal notation) that resembles:

```
Address: Pair: CHAOS 401
```

To map a Chaos address into a DNA address, first determine the Chaos host number and subnet number from the address. The Chaos host number is the DNA node number. The Chaos subnet number is the DNA area number.

Chaos Address 401



The Chaos subnet number is 1.
The Chaos host number is 1.
The Chaos address is 401 octal.

In this example, the Chaos subnet number is 1, so the DNA area number is 1. The Chaos host number is 1, so the DNA node number is 1. The Chaos address 401 maps into a DNA address of 1.1.

Note that this mapping of Chaos subnet number to DNA area number works only if the Chaos subnet number uses six or less of the available eight bits, that is, if the Chaos subnet number is 128 or less. Any Chaos address that is 37777 or less can be fully mapped into a DNA address. Chaos addresses greater than 37777 can be partially mapped into DNA addresses, by mapping only the Chaos host number into the DNA node number.

5. Symbolics Generic Network System

This section provides information useful to anyone who is maintaining the namespace database and wants to understand more about how it fits into the networking system. In brief, this section describes some of what goes on automatically when a network service is requested by one host and performed by another host.

The *generic network system* is the conceptual framework of Symbolics' implementation of network communications. This section describes some key aspects of network communication, including: the roles of the two computers, the service entries stored in the namespace database, network addresses, and the process of finding a path to a desired service on a remote host.

This section describes mediums and defines the terms *generic* and *specific medium*. This section also lists the mediums and protocols supported by Symbolics computers.

5.1. Network Users and Servers

When a network service is performed, the work is done in a dialogue between two hosts. A protocol is a specification of the dialogue that occurs over the network. The host that requests the service is called the *user host*, and the host that performs the service is the *server host*.

Each network protocol has two implementations, a *user side* and a *server side*. The user side is a program that runs on the user host; the server side is a program that runs on the server host. A service is obtained by a user side using a protocol to communicate via a network medium with a server side.

In many cases, a host provides both a user side and a server side for the same protocol. Sometimes the Symbolics computer supports a protocol with a user side but no server side. This means that the Symbolics computer can use the service if another host provides it. The **:tcp-gateway** protocol is one example of this.

In other cases, the Symbolics computer supports a protocol with a server side but no user side. If another host supports a user protocol, that host can take advantage of the server on the Symbolics computer. Or, you could write such a user program on another host.

Some services are provided locally. The medium of such a service is **:local**. These services are performed without using the network when the user host is the same as the server host.

5.2. Service Attributes in the Namespace Database

This section describes the role of the namespace database service attributes.

Purpose of Service Attributes

Typically, host objects contain one or more service attributes. The purpose of each attribute is to inform all hosts on the network that this host can provide a given service, and the details of how it can provide the service (the protocol and medium).

When you request a generic network service, your machine is the user host. The user host consults the namespace database and looks at the host object of the server host to determine if it provides the desired service. Therefore, every host at the site that is expected to perform network services should have information on all services it can provide entered in the service attributes of its host object.

Thus, a computer that acts as a file server must contain a **:file** service attribute for each medium and protocol for which it provides **:file** service in its host object. Similarly, a computer that acts as a namespace server must have service attributes for the **:namespace** and **:namespace-timestamp** services in its host object.

Three Parts of a Service Attribute

A service attribute has three parts: service, medium, and protocol. Each generic network service is implemented by a protocol, communicating through a medium. The service attribute of a host object resembles:

`Service:` *a triple of service, medium, and protocol*

Although the names of services, mediums, and protocols are keywords, you should not enter the colon when editing the namespace database.

service is the name of the generic network service. Some services are implemented on more than one medium or protocol. For example, a host might contain the following service attributes:

```
Service: FILE TCP NFILE
Service: FILE CHAOS NFILE
Service: FILE CHAOS QFILE
```

medium is a specific medium in the namespace database, even if the protocol is defined to be built on a generic medium. For example, **:file** service is defined for the generic **:byte-stream-with-mark** medium, using the **:nfile** protocol. **:byte-stream-with-mark** is implemented over two specific mediums: **:chaos** and **:tcp**. Therefore, the host object has two separate service attributes that contain the two specific mediums for **:file** service and **:nfile** protocol. To match a generic medium with the specific medium or mediums that implement it, see the section "Descriptions of Defined Mediums", page 42.

Some generic network services are implemented on the **:local** medium. It is not necessary to have a service attribute for any service implemented on **:local**. A host that provides a **:local** service stores that information internally and does not consult the namespace when such a service is requested and performed.

protocol is the name of the protocol that the server offers. In some cases, the names of the service and the protocol are the same, as in this service attribute:

```
Service: SEND CHAOS SEND
```

Symbolics computers are capable of providing many generic network services. The services themselves are described elsewhere: see the section "Descriptions of Defined Generic Services", page 48.

5.3. Network Mediums

A *medium* is one of the layers of abstraction in the network paradigm. Each protocol is associated with a medium. The medium provides a way for the information of the protocol to be communicated; it fills in some lower-level details of the communication. For example, the medium knows how to open a connection to a remote host. Because there are different ways to open connections to hosts, there are different mediums. Some examples of mediums are: **:chaos**, **:tcp**, and **:dna**.

5.3.1. Generic and Specific Mediums

The network system has two types of mediums: *generic mediums* and *specific mediums*.

Examples of Generic Mediums	*Examples of Specific Mediums*
:byte-stream	**:chaos**
:byte-stream-with-mark	**:chaos-simple**
:datagram	**:tcp**
	:dna
	:dial

Generic mediums are useful because some protocols are written in such a way that they require only a generic byte stream or generic datagram medium, and do not care about the details of how those things are implemented. Generic mediums can operate over many kinds of network. Each generic medium is implemented by one or more specific mediums, because the generic medium does not understand the lower-level details that are necessary to communicate over a particular kind of network.

The specific mediums sometimes take advantage of the features peculiar to a specific network in order to provide higher performance or special services.

It is not possible to make a strictly dualistic distinction between generic and specific mediums, because one medium can be implemented by another, which is implemented by a third, and so on. The structure is really a directed graph rather than a pair of layers.

Here are the definitions of two generic mediums, **:byte-stream** and **:datagram**:

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

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

When a specific medium is defined, it usually implements one more more generic mediums. Thus the specific medium provides a specific implementation of the generic medium. The second subform of the **net:define-medium** form contains the generic mediums on which this medium is built. The following form defines the **:chaos** medium, which is built on two generic mediums, **:byte-stream** and **:byte-stream-with-mark**:

```
(define-medium :chaos (:byte-stream :byte-stream-with-mark)
  (((:network :chaos)) lambda-list
   body))
```

Similarly, the definition of the **:chaos-simple** medium shows that it is built on the **:datagram** generic medium:

```
(define-medium :chaos-simple (:datagram)
  (((:network :chaos)) lambda-list
   body))
```

Generic mediums never 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 mediums often appear in server and protocol definitions. When a service is requested, a specific medium is chosen based on what is found in the service attribute of the host object of the server host.

5.3.2. Descriptions of Defined Mediums

It is customary that user and server sides of protocols are defined to use a generic medium (in the **net:define-server** and **net:define-protocol** forms). Each generic medium is supported by one or more of the specific mediums listed below.

Generic Mediums:

:byte-stream Delivers bytes reliably from one end of the connection to the other. The bytes arrive intact and in the original order. This medium is used for protocols that require a stream of data bytes, such as the **:nfile** protocol.

:byte-stream-with-mark
Provides the same functionality as **:byte-stream**, with the additional feature that either side may safely interrupt the flow of data. This medium has a mark that makes it possible to resynchronize the connections between the two hosts, should it be required. See the section "BYTE-STREAM-WITH-MARK Network Medium", page 159.

:datagram A datagram is some small number of bytes of data. The datagram arrives at the destination intact, but might arrive multiple times or fail to arrive at all. If you send two datagrams, they might not arrive in the order that they were sent. This medium is used by protocols that provide their own error checking, or do not require error checking. **:datagram** is appropriate for protocols that perform simple tasks, such as requesting the time of day.

Specific Mediums:

:chaos Supports the **:byte-stream** and **:byte-stream-with-mark** generic network mediums. All Symbolics computers support the **:chaos** medium, which is used by the Chaosnet type of networks. Chaosnets usually use Ethernet hardware.

:chaos-simple Supports the **:datagram** generic network medium. All Symbolics computers support the **:chaos-simple** medium, which is used by the Chaosnet type of networks.

:dial Supports communications over the international telephone network. All Symbolics computers support the **:dial** medium software; however, they require a modem to physically connect to the telephone network. **:dial** supports the **:byte-stream** medium. The primary use of the **:dial** medium is mail transfer. See the section "Dial Network Medium", page 159.

:local Enables a host to provide a service locally, without using the network.

:tcp Supports the **:byte-stream** and **:byte-stream-with-mark** generic mediums. It is used to communicate with hosts on IP/TCP networks, such as the ARPA Internet. This medium is supplied with the optional IP/TCP software package. **:tcp** is the Transmission Control Protocol medium as described in ARPA RFC 793, available from ARPA Network Information Center.

:udp Supports the **:datagram** generic medium. It is used to communicate with hosts on IP/TCP networks, such as the ARPA Internet. This medium is supplied with the optional IP/TCP software package. **:udp** is the User Datagram Protocol medium as described in ARPA RFC 768, available from ARPA Network Information Center.

:dna Supports the **:byte-stream** generic medium. Provides communications using DECnet protocols, as described in *DECnet Digital Network Architecture (Phase IV) General Description*, available from Digital Equipment Corporation. This medium is supplied with the optional DNA software package.

5.4. Generic Network Services

For information on how to write application programs built on the foundation of the generic network system, see the section "Defining a New Network Service", page 97.

5.4.1. Protocols Supported by all Symbolics Computers as Users

This chart lists the generic services that are supported by user sides on all Symbolics computers, and the specific medium and protocol on which each service is implemented. For related information: See the section "Descriptions of Defined Generic Services", page 48.

The optional software packages support additional capabilities; these are listed separately.

The variable **neti:*protocol-list*** is a list of user-side descriptions.

Service	*Medium*	*Protocol*
BAND-TRANSFER	CHAOS	BAND-TRANSFER
CHAOS-STATUS	CHAOS-SIMPLE	CHAOS-STATUS
CONFIGURATION	CHAOS	CONFIGURATION
DOMAIN	CHAOS	DOMAIN
ECHO-XCN-TOKEN-LIST	CHAOS	ECHO-XCN-TOKEN-LIST
EXPAND-MAIL-RECIPIENT	CHAOS	EXPAND-MAILING-LIST
EXPAND-MAIL-RECIPIENT	CHAOS	SMTP
FILE	CHAOS	NFILE
FILE	CHAOS	QFILE
HARDCOPY-STATUS	CHAOS	LGP-QUEUE
HARDCOPY	CHAOS	LGP
HARDCOPY	CHAOS	PRINTER-QUEUE
LISPM-FINGER	CHAOS-SIMPLE	LISPM-FINGER
LOGIN	CHAOS	3600-LOGIN
LOGIN	CHAOS	SUPDUP
LOGIN	CHAOS	TELNET
LOGIN	CHAOS	TELSUP
LOGIN	CHAOS	TTY-LOGIN
LOGIN	DIAL	TELNET
MAIL-PROBE	DIAL	MAIL-PROBE

MAIL-TO-USER	CHAOS	CHAOS-MAIL
MAIL-TO-USER	CHAOS	SMTP
MAIL-TO-USER	DIAL	SMTP
NAMESPACE-TIMESTAMP	CHAOS-SIMPLE	NAMESPACE-TIMESTAMP
NAMESPACE	CHAOS	NAMESPACE
NOTIFY	CHAOS	NOTIFY
PRINTER-QUEUE-CONTROL	CHAOS	PRINTER-QUEUE
PRINTER-CONTROL	CHAOS	PRINTER-QUEUE
RESET-TIME-SERVER	CHAOS-SIMPLE	RESET-TIME-SERVER
SEND	CHAOS	CONVERSE
SEND	CHAOS	SEND
SEND	CHAOS	SMTP
SHOW-USERS	CHAOS	NAME
STORE-AND-FORWARD-MAIL	CHAOS	CHAOS-MAIL
STORE-AND-FORWARD-MAIL	CHAOS	SMTP
STORE-AND-FORWARD-MAIL	DIAL	SMTP
TAPE	CHAOS	RTAPE
TIME	CHAOS-SIMPLE	TIME-SIMPLE
UPTIME	CHAOS-SIMPLE	UPTIME-SIMPLE
WHO-AM-I	CHAOS-SIMPLE	WHO-AM-I

5.4.2. Protocols Supported by all Symbolics Computers as Servers

This chart lists the generic services that are supported by server sides on all Symbolics computers, and the medium and protocol on which each service is implemented.

See the section "Descriptions of Defined Generic Services", page 48.

The variable **neti:*servers*** is a list of server-side descriptions.

Service	*Medium*	*Protocol*
BAND-TRANSFER	CHAOS	BAND-TRANSFER
CHAOS-STATUS	CHAOS-SIMPLE	CHAOS-STATUS
CONFIGURATION	CHAOS	CONFIGURATION
DOMAIN	CHAOS	DOMAIN
EXPAND-MAIL-RECIPIENT	CHAOS	SMTP
FILE	CHAOS	NFILE
FILE	CHAOS	QFILE
HARDCOPY-STATUS	CHAOS	LGP-QUEUE
HARDCOPY	CHAOS	LGP
HARDCOPY	CHAOS	PRINTER-QUEUE
LISPM-FINGER	CHAOS-SIMPLE	LISPM-FINGER
LOGIN	CHAOS	3600-LOGIN
LOGIN	CHAOS	SUPDUP
LOGIN	CHAOS	TELNET

LOGIN	CHAOS	TTY-LOGIN
LOGIN	DIAL	TELNET
MAIL-PROBE	DIAL	MAIL-PROBE
MAIL-TO-USER	CHAOS	CHAOS-MAIL
MAIL-TO-USER	DIAL	SMTP
NAMESPACE-TIMESTAMP	CHAOS-SIMPLE	NAMESPACE-TIMESTAMP
NAMESPACE	CHAOS	NAMESPACE
NOTIFY	CHAOS	NOTIFY
PRINTER-QUEUE-CONTROL	CHAOS	PRINTER-QUEUE
PRINTER-CONTROL	CHAOS	PRINTER-QUEUE
RESET-TIME-SERVER	CHAOS-SIMPLE	RESET-TIME-SERVER
SEND	CHAOS	CONVERSE
SEND	CHAOS	SEND
SEND	CHAOS	SMTP
SHOW-USERS	CHAOS	NAME
STORE-AND-FORWARD-MAIL	CHAOS	CHAOS-MAIL
STORE-AND-FORWARD-MAIL	DIAL	SMTP
TAPE	CHAOS	RTAPE
TIME	CHAOS-SIMPLE	TIME-SIMPLE
UPTIME	CHAOS-SIMPLE	UPTIME-SIMPLE
WHO-AM-I	CHAOS-SIMPLE	WHO-AM-I

The server protocols related to the mailer are available only if the mailer is installed. The server protocols related to hardcopy and printers are available only if the print spooler is installed.

5.4.3. TCP and UDP Protocols Supported by Symbolics Computers as Users

The IP/TCP software package enables Symbolics computer users to access the following services provided by other hosts:

Service	*Medium*	*Protocol*
CONFIGURATION	TCP	CONFIGURATION
DOMAIN	TCP	DOMAIN
DOMAIN	UDP	DOMAIN-SIMPLE
EXPAND-MAIL-RECIPIENT	TCP	SMTP
FILE	TCP	NFILE
FILE	TCP	TCP-FTP
FILE	UDP	TFTP
LISPM-FINGER	UDP	LISPM-FINGER
LOGIN	TCP	3600-LOGIN
LOGIN	TCP	SUPDUP
LOGIN	TCP	TELNET
MAIL-TO-USER	TCP	SMTP
SEND	TCP	SMTP

SHOW-USERS	TCP	ASCII-NAME
STORE-AND-FORWARD-MAIL	TCP	SMTP
TCP-GATEWAY	CHAOS	TCP-GATEWAY
TIME	TCP	TIME-MSB
TIME	UDP	TIME-SIMPLE-MSB

5.4.4. TCP and UDP Protocols Supported by Symbolics Computers as Servers

The IP/TCP software package enables Symbolics computers to provide the following services:

Service	*Medium*	*Protocol*
CONFIGURATION	TCP	CONFIGURATION
DOMAIN	TCP	DOMAIN
DOMAIN	UDP	DOMAIN-SIMPLE
EXPAND-MAIL-RECIPIENT	TCP	SMTP
FILE	TCP	NFILE
FILE	TCP	TCP-FTP
FILE	UDP	TFTP
IEN-116	UDP	IEN-116
LISPM-FINGER	UDP	LISPM-FINGER
LOGIN	TCP	3600-LOGIN
LOGIN	TCP	SUPDUP
LOGIN	TCP	TELNET
MAIL-TO-USER	TCP	SMTP
SEND	TCP	SMTP
SHOW-USERS	TCP	ASCII-NAME
STORE-AND-FORWARD-MAIL	TCP	SMTP
TIME	UDP	TIME-SIMPLE-MSB
UNIX-RWHO	UDP	UNIX-RWHO

The server protocols related to the mailer are available only if the mailer is installed.

5.4.5. TCP and UDP Protocols Supported by SUN Computers as Servers

Service	*Medium*	*Protocol*
FILE	TCP	TCP-FTP
FILE	UDP	NFS
HARDCOPY	TCP	UNIX-LPD
PRINTER-QUEUE-CONTROL	TCP	UNIX-LPD
PRINTER-CONTROL	TCP	UNIX-LPD

LOGIN	TCP	TELNET
RPC	UDP	UDP-RPC
RPC	TCP	TCP-RPC
RPC	TCP	RPC
SEND	TCP	SMTP
SHOW-USERS	TCP	ASCII-NAME
TAPE	TCP	UNIX-REXEC
TIME	UDP	TIME-SIMPLE-MSB
UNIX-REXEC	TCP	UNIX-REXEC
X-WINDOW-SYSTEM	TCP	X-WINDOW-SYSTEM

5.4.6. DNA Protocols Supported by Symbolics Computers as Users

The DNA software package enables Symbolics computer users to access the following services provided by other hosts:

Service	*Medium*	*Protocol*
FILE	DNA	DAP
LOGIN	DNA	CTERM
MAIL-TO-USER	DNA	DNA-MAIL
SHOW-USERS	DNA	ASCII-NAME
TAPE	DNA	RTAPE
TIME	DNA	DNA-LMTIME
UPTIME	DNA	DNA-LMUPTIME

5.4.7. DNA Protocols Supported by Symbolics Computers as Servers

The DNA software package enables Symbolics computers to provide the following services:

Service	*Medium*	*Protocol*
FILE	DNA	DAP
LOOPBACK	DNA	DNA-LOOPBACK-MIRROR
MAIL-TO-USER	DNA	DNA-MAIL

5.4.8. Descriptions of Defined Generic Services

:band-transfer

The user side requests that a copy of a world load be transferred. This transfer can be in either direction. The Copy World command uses this service.

:configuration
The server reports its hardware configuration to the user. The Show Machine Configuration command uses this service.

:domain The server is capable of being an Internet Domain Server. This is used when parsing host names. See the section "Internet Domain Names", page 148.

:expand-mail-recipient
The server returns the elements of a mailing list. The Show Expanded Mailing List (M-X) Zmail command uses this service.

:file The user host performs operations on files stored on a remote host. The server host responds to requests from the user host relating to file access. File access can include these file operations: open, close, read, write, probe, directory, and so on.

:hardcopy-status
The server sends a description of the current status of a local hardcopy device and its spooler to the user. This is used by sites that have one or more Symbolics computers.

:hardcopy The server prints a file on a local hardcopy device. The Hardcopy File command and the **hardcopy:make-hardcopy-stream** function use this service. This is used by sites that have one or more Symbolics computers.

:lispm-finger
The server host provides information on the users currently logged in to this host. Returns a list of (*host-name user-id host-location idle-time personal-name group*). The Show Users command uses this. If you prefer to keep certain fields of your user object private, such that the **:lispm-finger** protocol does not return them: See the section "Censoring Fields for lispm-finger and name Services" in *Genera User's Guide*.

:login The server permits a user to log in remotely. The Terminal program uses this service.

:mail-to-user
The server delivers an electronic mail message to the mailbox of the recipient of the message. **:mail-to-user** service performs delivery only if the mailbox is stored locally on the server host.

:namespace-timestamp
This service is used to determine whether the data in the namespace database has changed. The server returns a timestamp of the last update to the database. It is necessary for any namespace server to provide this service.

:namespace The namespace system uses this service to query and update the namespace database. It is necessary for any namespace server to provide this service. For information on the protocol used to provide this service: See the section "Network Namespace Protocol", page 231.

:notify The server issues an asynchronous message to a local user or users. **net:notify** and **chaos:notify-all-lispms** use this service.

:printer-control
The server manipulates a local hardcopy device, as requested by the user. The Halt Printer command uses this service.

:printer-queue-control
The server manipulates the queue of a local hardcopy device, as requested by the user. The Delete Printer Request command uses this service.

:reset-time-server
The server host resets its own internal time to the time returned by one of the network hosts.

:send The server host sends an interactive message to a designated user (person) on that host. The Converse program uses this service.

:show-users The server returns information on the users currently logged in to this host The Show Users command uses this service. If you prefer to keep certain fields of your user object private, such that the **:name** protocol does not return them: See the section "Censoring Fields for lispm-finger and name Services" in *Genera User's Guide*.

:store-and-forward-mail
The server participates in the delivery of an electronic mail message. The message is forwarded to another host on the network which is closer to the target host. If the next host in the path is down, the server holds the message (hence the "store" in the name of the service) and retransmits it when the host is up.

:tape The server side transfers data between a tape and the user side. The transfer can be in either direction. **tape:make-tape-stream** uses this service.

:time The server returns the current universal time, or **nil** if it cannot find the current time. See the section "Representation of Dates and Times" in *Programming the User Interface*.

:tcp-gateway
The server host is capable of being a TCP gateway, which means it can create TCP connections on behalf of the user side. This is useful when the user host has no IP-TCP medium directly connected to it.

:uptime The server returns the amount of time it has been up, in sixtieths of a second.

:who-am-i The server provides information about itself. Returns three values: the keyword that names the namespace of this host; the host name (or **:unknown**); and the host that responded with this information. This is used by Symbolics computers at boot time.

5.5. Enabling and Disabling Network Services

If a network service is enabled on your host, your host performs the service when requested to do so by another network host. If a service is not enabled, your host refuses to perform the service when it is requested.

When you cold or warm boot your machine, the function **sys:enable-services** is called. It enables the network services indicated by the variable **neti:*standard-services-enabled***.

You can enable or disable selected network services using **sys:enable-services** and **sys:disable-services**.

sys:enable-services &optional (*services* **neti:*standard-services-enabled***) *Function*

Enables selected network services. *services* can be a symbol that names a single service to enable, or a list of symbols naming services to enable, or **:all**, to enable all services. If no argument is provided, only those services indicated by the variable **neti:*standard-services-enabled*** are enabled.

If the keyword symbol that names a service has a **sys:enable-services** property, that function is called with the name of the service as its sole argument.

sys:disable-services &optional (*particular-services* **':all**) *Function*

Disables network services. *particular-services* can be a symbol that names a service, or a list of symbols to disable. If no argument is provided, all services are disabled. For example:

```
(sys:disable-services ':send)
```

If the keyword symbol that names a service has a **sys:disable-services** property function, that function is called with the name of the service as its sole argument.

neti:*standard-services-enabled* *Variable*

Contains the services that are enabled by **sys:enable-services** by default. This variable is one of:

:all All services are enabled; this is the default.

nil No services are enabled.

list Only the services in *list* are enabled.

neti:*new-services-enable* *Variable*

A non-**nil** value ensures that when a new service is defined it is also enabled (if any services are enabled). The default is **nil**.

neti:service-enabled-p *protocol-name* *Function*

protocol-name is a keyword symbol that names a protocol. If the service implemented by that protocol is currently enabled, the list of enabled services is returned. *protocol-name* is the first element of the list.

Returns **nil** if the service is not currently enabled.

For example:

```
(neti:service-enabled-p ':send)
```

net:*services-enabled* *Variable*

Contains a list of the network services currently enabled on this host.

sys:enable-services *Property*

Server name symbols can have a **sys:enable-services** property. This function is called when the function **sys:enable-services** is called; the function should enable the service. The argument is always the name of the service. For example:

```
(defun (:property service sys:enable-services) (arg)
  body...)
```

sys:disable-services *Property*

Server name symbols can have a **sys:disable-services** property. This function is called when the function **sys:disable-services** is called; the function should disable the service. The argument is always the name of the service. For example:

```
(defun (:property service sys:disable-services) (arg)
  body...)
```

6. The Remote Procedure Call Facility

6.1. Overview of Symbolics RPC

Symbolics RPC is an implementation of industry-standard RPC that underlies Sun Microsystems' NFS and other programs (see Request for Comments (RFC) #1057 "RPC: Remote Procedure Call Protocol specification version 2"). The distinguishing characteristic of Symbolics RPC is that it uses Lisp technology to provide a very clean and easy-to-use interface for defining RPC-based programs. The form of data transmitted over the communications medium is fully compliant with the standard.

Remote Procedure Call (RPC) is a facility that allows a function executing on one processor to call a function executing on another processor. The two functions can be written in the same language or in different languages, such as Lisp and C. The two processors can be of the same type or of different types; for example, a function executing on an Ivory can call a function that executes on an MC68020.

RPC allows a program executing on one processor to access facilities that are available on another processor. For example, an Ivory embedded in a host can use RPC to make use of hardware devices controlled by that host, to call facilities of the host operating system, and to call program libraries that are available for the host but not for the Ivory. Similarly, a program running on a host can use RPC to call symbolic processing facilities such as Joshua that run on the Ivory.

Using RPC, you can segment a program into pieces and run each piece on a different processor. This can improve performance through parallel processing. More importantly, this allows each part of the program to execute on the processor and under the operating system best adapted to support that part. Benefits include both performance improvement and ease of programming.

For example, a program for a MacIvory system can run its user interface on the Macintosh and its knowledge processing on the Ivory. It is not necessary to have such a large granularity in the segmentation of a program; the same program might be improved by running the high-level "policy" portion of its user interface on the Ivory, with the low-level "mechanism" portion running on the Macintosh. Dynamic Windows on MacIvory work precisely this way.

Another reason to use RPC is when you want to run a program on processor A but it needs to cooperate with an existing program that is available only on processor B. Processor A might be an Ivory, which you are using because of its ease of programming, while processor B might be a non-Symbolics processor, with a large library of available programs. The main part of your program runs on processor A and it includes an appendage that runs on processor B; the appendage communicates with the existing program using the interfaces defined by the existing program. The main part of your program and the appendage communicate through RPC. The existing program is unaware of RPC and does not have to be modified or adapted. (The Genera interface to HyperCard on MacIvory works this way.)

RPC provides communication between two processors in a single system, as when a Symbolics Ivory is embedded in a non-Symbolics platform such as a Macintosh or Sun.

In this case communication is through shared memory and is quite efficient, although of course calling a function remotely is never as fast as calling it locally. RPC can also be used for communication between two processors in separate systems, which might be physically located side by side or at a great distance from each other. RPC operates through local-area and wide-area networks and through RS232 serial lines. Using RPC over a network is slower than using RPC in an embedded system.

Symbolics RPC provides a transparent interface; calling a function remotely looks the same as calling a local function. When you call a function, you do not have to know whether its body executes on the local processor or on a remote processor. This is true regardless of whether you program in Lisp or in C. The RPC system implements this by automatically defining a *stub function* that acts as a local representative of the remote function. The stub takes care of all the housekeeping required to transmit the arguments to the remote function and receive back the values. Symbolics RPC provides a transparent interface for the callee as well. You write the body of a remotely callable function in Lisp or C in the usual way; the RPC system automatically adds code to receive the arguments, puts them in variables with the names you specified, and sends back the results.

6.1.1. Symbolics RPC and Sun RPC

Symbolics RPC is a fully compliant implementation of the RPC and XDR *(eXternal Data Representation)* standards described in RFC (Request for Comments) #1057 "RPC: Remote Procedure Call Protocol specification version 2" and RFC #1014 "XDR: External Data Representation standard."

As such, it is completely compatible and can interoperate with any other compliant RPC implementation, such as the one supplied with Sun Microsystems computers. (See the Sun Microsystems document *Network Programming* for further information.) For instance, a program written in Symbolics RPC can make RPC calls to a program written in SunRPC language, and vice versa.

Symbolics RPC language differs from SunRPC language in many ways, most notably in that Symbolics RPC can simultaneously generate code in two programming languages, C and Lisp. Symbolics RPC language cannot generate code in Sun RPC language. Users of the Symbolics UX can choose either for programming. Symbolics RPC language is likely to make the code-maintenance task easier for programs that will run on both Ivory-based systems and a C-based system.

6.1.2. Differences Between Local and Remote Function Calling

An important and necessary difference between local and remote function calling is that functions executing on separate processors have separate memory address spaces and cannot share any data. All arguments and values must be passed by value, not by reference. For this reason, unlike a locally callable function, a re-

motely callable function uses special functions (**rpc:rpc-values** and **rpc:rpc-error** in Lisp, RPCValues and RPCError in C) to return its results.

Because there is no call by reference, the data types that can be used with RPC are limited. For example, in Lisp you cannot pass an arbitrary symbol as an argument. If you pass a flavor instance, the callee sees a copy of the instance. If the callee modifies the instance, those modifications are not passed back to the caller. On the other hand, a benefit of call by value is that the caller and callee can use different data representations. For example, the caller can pass a Lisp flavor instance, which the callee will see as a C struct.

You construct an RPC-based program by using a set of Lisp macros to define the remotely callable functions. These Lisp macros are somewhat unusual in that they expand into both Lisp code and C code. The Lisp expansion is processed in the normal way. The C expansion is written to a file that can be compiled by the Symbolics C compiler or shipped to another processor and compiled by its own C compiler. Once the interface has been defined and compiled, you call the stub functions using ordinary Lisp or C function calls. The callee or server half of the interface is loaded together with any other programs it calls.

6.1.3. Basic Concepts of RPC

The basic concepts of RPC include *remote modules, remote entries, remote errors*, and *remote types*, explained in the following table:

remote entry	A remotely callable function.
remote module	A collection of related remote entries that are treated as a unit for bookkeeping purposes.
remote error	An exceptional condition that can arise while executing a remote entry.
remote type	A type of data that can be used as an argument to a remote entry or a remote error, and can be returned as a value by a remote entry. A remote type defines the possible data values, their representation in Lisp and C, their representation for interprocessor transmission, and the methods for converting between these representations.

Some of these concepts have nonstandard names. These names were chosen to avoid any confusion with other concepts in Genera with names similar to the standard names. Other systems call remote modules "remote programs" and call remote entries "remote procedures."

The RPC facility consists of three layers:

- The *call layer* is in charge of identifying remote entries to be called, transmitting the arguments to them, matching up the returned values with the caller who is awaiting the results, and reporting errors.

- The *data representation* layer is in charge of defining a common representation for data and translating representations used by different machines and by different programming languages to and from the common representation.

- The *transport layer* is in charge of moving raw bits between machines and dealing with bit-ordering issues. There are three different transport layers, selectable at run time. One is based on the embedding substrate's inter-processor communication mechanism, the others are based on the byte-stream and UDP/IP media of the generic network system.

6.2. Symbolics RPC Facilities in Lisp

6.2.1. Extensions to Lisp Syntax for RPC

The RPC facility involves several Lisp macros that expand into C code. To make these macros easier to write, we have extended Lisp syntax with a facility that amounts to a C version of Lisp's backquote. This minimizes the syntactic clumsiness of constructing C programs with Lisp code, as compared to doing a lot of explicit string manipulation or using a package for manipulating C parse trees.

To enable this syntax, you must specify `-*- Syntax: Lisp+C -*-` in the file's attribute list. This is the same as `-*- Syntax: Common-Lisp -*-` except for the addition of a **#{** reader macro. Text between **#{** and the balancing **}** is C source code with one exception noted below. The result of reading a **#{...}** expression is a Lisp form which, when evaluated, produces a list of tokens. There is a function **rpc:write-c-token-list** that prints a list of tokens to a file, producing a valid and more or less legible C program. Tokens are Lisp integers, strings, characters, and symbols, with most C tokens being represented as symbols in the **rpc** package.

Inside a **#{...}** the ↓ character allows you to drop in some tokens produced by a Lisp form. This is the equivalent of comma in Lisp's backquote. A ↓ is followed by a Lisp form that evaluates to a list of tokens. A ↓↓ is followed by a Lisp form that evaluates to a single token (a less common case in practice). When using this syntax, you need to be careful to terminate the Lisp form in a Lisp way, not a C way; it is usually best to leave a space after the form before resuming C syntax.

Syntax Examples

1. A simple C program:

```
#{ printf("Hello, world.\n"); }
```

2. A Lisp function that generates variations on that C program:

```
(defun hello (&optional (whom "world"))
  #{ printf("Hello, %s.\n", ↓↓whom); }
  )

(hello "my fellow Americans")
```

Note how the close parenthesis is not placed on the same line as the semicolon, to avoid confusing Zmacs' Lisp expression parser. At present, Zwei does not understand #{...} syntax, so you must be careful about C semicolon characters, which will be interpreted as comments. Do not put a close parenthesis on the same line after a semicolon.

3. A Lisp function that takes C programs as both input and output:

```
(defun print-out (C-expression)
  (let ((string (with-output-to-string (s)
                  (rpc:write-c-token-list C-expression s))))
    #{ printf("The value of %s is %d\n",
              ↓↓string , ↓C-expression ); }
))
```

For additional information: See the macro **rpc:define-remote-type**, page 70. See the macro **rpc:define-remote-c-program**, page 62.

6.2.2. Macros for Defining RPC-based Programs

rpc:define-remote-module *module-name* &rest *options* *Macro*

Defines a module. A module is a collection of related entry points. These are called "modules" rather than "programs" to avoid confusion with **dw:define-program-framework**.

module-name is a symbol that identifies this module. Some information about the module is kept in an object that is stored as the **remote-module** property of the symbol. If there is a Lisp client for this module, the special variable ***module-name*-remote-module*** holds the object also, so the stubs can get at it quickly.

Valid options are:

(:version *integer***)** Version of the interface. This option is mandatory. Change the version number when you change the interface. Only equal versions are compatible.

(:number *integer***)** A number that uniquely identifies the remote module. The identifying *integer* for a remote module must be unique world-wide. For information on how to choose a number, see the section "Remote Module Numbers", page 58. This option is mandatory.

(:process *keyword value keyword value...***)** or **(:process nil)**
Controls what process a server runs in. The default is for each incoming call to start a new process, in which the server for that call runs. The first form of the **:process** option can be used to specify options for this process, such as **:name** and **:priority**.

The **(:process nil)** option causes the server to run in the RPC dispatcher process. This ensures that asynchronous calls are processed in order, but is dangerous if the server runs for a long time or can block, since all RPC service is delayed until the server finishes.

(:server *languages...***)** or **(:client** *languages...***)**
These options tell the macros what stubs and handlers to generate. At least one must be specified. *languages* are keywords **:lisp** and **:c**.

The **:server** option lists the languages in which entries of this module can execute. The **:client** option lists the languages from which entries of this module can be called.

(:allow-untrusted-access *boolean***)**
If *boolean* is nil, then network access (TCP or UDP) to this RPC module from untrusted hosts is not permitted. Any such network connections will be rejected by stream-based RPC, and ignored by datagram-based RPC. The default is **nil**.

(:authentication *authentication-flavors***)**
Authentication-flavors specifies the flavor of authentication that entries in this module will use. The recognized values are **:null, :unix,** and **:des**. The default is **:null** authentication. For related information, see the **:authentication** option to **rpc:define-remote-entry**.

:unix authentication requires that the client host be trusted by Genera on the server host, regardless of the setting of the **:allow-untrusted-hosts** option. When more then one authentication flavor is listed, the server allows the least secure and the client uses the most secure. If both **:des** and **:unix** are specified, the client tries des first, then, because **:des** authentication is as yet unimplemented, it falls back to **:unix**. The server will accept requests that use either unix or des authentication (assuming always that they have valid credentials), except that it will only allow **:unix** authentication from trusted clients.

6.2.2.1. Remote Module Numbers

In interprocessor communication, modules are identified by their number, not by name. Choose module numbers following the conventions in the file SYS:EMBEDDING;RPC;ASSIGNED-NUMBERS.TEXT . All module numbers for a particular site must be unique.

rpc:define-remote-entry *entry-name module-name* &rest *options* *Macro*

Defines an entry point to a remote module. This function does three things: defines the interface to a remotely callable entry, creates zero or more stubs for calling this entry from various languages, and creates zero or more handlers for implementing this entry in various languages. When a stub is called by a normal function call, it contacts a server on some other machine, sends the arguments, and receives and returns the values. A handler is called by the server and its body performs the desired action on behalf of a stub running on some other machine.

entry-name is a symbol that names the Lisp stub function. The C stub's name is derived from this by a simple character mapping (hyphen to underscore, uppercase to lowercase). The function name for a handler is *entry-name*-handler.

module-name is a symbol that identifies what remote module this entry belongs to.

Valid options are:

(:number *integer***)** An identifying number. This option is mandatory. *integer* must be unique within the module. Because this number is also used to index an array, it is more efficient to number entries in order starting with 1.

(:arguments (*name type*)...**)**
Declares the names and types of the arguments. *name* is a symbol that can name a Lisp variable. *type* is a symbol that names a previously defined remote type, or a list of such a symbol and parameters. Only required arguments are allowed, not `&optional`, `&rest`, or `&key` arguments. This option is mandatory.

An argument specifier can also be (*name type* **:output** *value*). This means that the argument *name*, besides being transmitted with the call in the normal way, supplies storage that can be reused for the value named *value*. This reuse occurs only on the client side; the server knows nothing of it.

(:arguments (*name type* **:extent :dynamic** ...) ...**)**
An argument that is declared with **:extent :dynamic** may be stack-consed in the Lisp server. For further information on this topic, see the section "Consing Lists on the Control Stack" in *Internals*.

(:values (*name type*)...**)**
Declares the names and types of the values. A Lisp stub returns these values in the normal way. A C stub takes pointer parameters after the normal arguments and stores the values through those parameters. This option is mandatory unless **:asynchronous** is used.

A value specifier can also be (*name type* **:overwrite** *argument*), which means that the argument *argument* supplies storage that can be reused for this value. Also note that *argument* is not transmitted with the call; that is, it is an output argument, not an input/output argument.

(:future *boolean***)** If *boolean* is non-nil, this suboption generates start-*name*-future, finish-*name*-future, and abort-*name*-future stubs that allow RPC futures programming in Lisp. This option is only meaningful if the **:asynchronous** option is nil. RPC futures programming is not supported in C.

(:asynchronous *t/nil***)**
If **t**, calls do not wait for a reply and no values can be returned.

(:whostate *string***)** *String* is the whostate for a process executing this RPC entry while waiting for a reply. This option is meaningful only if the **:asynchronous** option is nil.

(:authentication *authentication-flavors***)**
Like the **:authentication** option to **rpc:define-remote-module**, which it overrides. This option is useful for modules that require some level of authentication, but that have an entry that allows null authentication, for example.

(:lisp *suboptions...***)** Suboptions for a Lisp-language stub and/or handler:

(:server *body...***)**
The body of a handler.

In the **(:lisp (:server** *body...***))** suboption, the body is a group of Lisp forms. The arguments are available as Lisp variables, values can be returned with the **rpc:rpc-values** macro, and errors can be reported by means of the **rpc:rpc-error** macro. Arguments to **rpc:rpc-error** are the name of the error (not evaluated) and argument forms.

(:c *suboptions...***)** Suboptions for a C-language stub and/or handler:

(:server *body***)**
The body of a handler.

In the **(:c (:server** *body***))** suboption, the body is a Lisp form that evaluates to a sequence of C tokens (use the **#{ ... }** reader syntax). The resulting C program will be enclosed in a

block (thus it can start with declarations). The arguments are available as C variables, values can be returned with the RPCValues macro, and errors can be reported by means of the RPCError macro. Errors can also be reported by returning an operating system error code with a return statement.

(:server-cleanup *statement***)**
Causes *statement* to be executed when the server exits, whether it exits normally or due to an error. This provides the same functionality for C that **unwind-protect** provides for Lisp.

Notes:

When storage is reused by **:output** or **:overwrite**, the type of the argument and the type of the value must agree. In Lisp, a vector must have a fill-pointer, which will be adjusted. In C, the vector is always a structure with length and elements, the length is modified, and there is no checking that sufficient storage was allocated. **:output** is for input/output arguments, while **:overwrite** is for output-only arguments.

rpc:define-remote-error *error-name module-name* &rest *options* *Macro*

Defines an error that can be reported from a server back to a client. In Lisp, the error is reported by signalling a condition. In C, the remote call returns an error code.

error-name is a symbol that names the error. In Lisp, this symbol can be used with the **rpc:rpc-error** macro to report the error; in C, it can be used with RPCError.

module-name is a symbol that identifies what remote module this error belongs to. Use **nil** for global errors.

Valid options are:

(:number *integer***)** An identifying number. This option is mandatory. *integer* must be unique within the module, or if *module-name* is **nil**, *integer* must be unique system-wide and must be negative. Because this number is used to index an array, it is more efficient to number errors in order starting with 1 or -1.

(:arguments (*name type*)...**)**
Declares the names and types of the arguments, which are specified after the *error-name* when calling **rpc:rpc-error** or RPCError.

(:condition *name***)** The name of the condition to be signalled by a Lisp stub when this error is reported. This defaults to a reasonable error condition.

(:handler *body***)** Code folded into the Lisp stub, responsible for selecting and signalling the appropriate error condition. The error arguments are lexically apparent to *body*. When **:handler** is specified, **:condition** (if any) is ignored.

rpc:rpc-values &rest *forms* *Macro*

Returns values. This is used in a remotely callable function to return values to its caller. This macro can only be called from within the body of a handler; that is, the **:server** suboption of the **:lisp** option of **rpc:define-remote-entry**.

rpc:rpc-error *error-name* &rest *arg-forms* *Macro*

Reports errors. This is used in a remotely callable function to report errors to its caller. Arguments to **rpc:rpc-error** are the name of the error (not evaluated) and argument forms. This macro can only be called from within the body of a handler; that is, the **:server** suboption of the **:lisp** option of **rpc:define-remote-entry**. See the macro **rpc:define-remote-error**, page 61 for related information.

rpc:define-remote-c-program *module-name* &rest *options* *Macro*

During compilation, writes out the C source file(s) for the remote module, based on information recorded by the macros **rpc:define-remote-module**, **rpc:define-remote-entry**, and **rpc:define-remote-error**. The expansion of the macro is an **(eval-when (compile eval) ...)**.

Valid options are:

(:client *suboptions...***)**
Specifies which side of the program to generate. At least one of **:client**, **:server**, **:client-extern**, or **:server-extern** must be specified.

(:server *suboptions...***)**
Specifies which side of the program to generate. At least one of **:client**, **:server**, **:client-extern**, or **:server-extern** must be specified.

(:client-extern *suboptions...***)**
Specifies which side of the program to generate. **:client-extern** is an include file with function prototypes for the remote entries of a client. At least one of **:client**, **:server**, **:client-extern**, or **:server-extern** must be specified.

(:server-extern *suboptions...***)**
Specifies which side of the program to generate. **:server-extern** is an include file with function proto-

types for the remote entries of a server. At least one of **:client**, **:server**, **:client-extern**, or **:server-extern** must be specified.

Suboptions are:

(:file *filename***)** Specifies what file to write. This option is mandatory. If you specify a filename only, the file is put in the directory of the file containing the **rpc:define-remote-c-program** form.

(:include *strings...***)** Specifies what include files are included.

(:prefix #{...}) Specifies the text to go at the front of the file.

(:suffix #{...}) Specifies the text to go at the end of the file.

(:symbolics-trade-secret t**)**
If specified, includes standard boilerplate.

(:init #{...}) Designates initialization code — valid inside **:server** only.

(:errors *remote-error-names***)**
Reports RPC errors. A suboption to the **:client** and **:client-extern** options only.

(:type *module-type***)** Declares the type of module. This option is valid only as a suboption of **:server**. For information on use of this option with MacIvories, see the section "Types of RPC Servers for MacIvory" in *MacIvory User's Guide*. For information on use of this option with UX400, see the section "Overview of RPC for the Symbolics UX" in *Open Genera User's Guide*.

module-type is one of:

:linked — The module is linked into the RPC program. Call initialize_*module_name*_server to set it up. This is the default.

:auto-load — The module is automatically loadable and cannot use any static data. The **:init** option cannot be used with this type module. This module type is valid only for the MacIvory.

:auto-load-with-static-data — The module is automatically loadable and can have static data. This module type is valid only for the macIvory.

You can split the remote entries of a module into several submodules. Each submodule is written to a separate .c file, reducing the size (but increasing the number) of .c source files. Use an **rpc:define-remote-c-program** form for each submodule, including the following suboptions:

(:entries-only *remote-entry-name ...***)**
Include only code for the named remote entries. You should also use the **:include** suboption with this suboption to specify the inclusion of the server's function prototypes header file that was generated using the **:server-extern** option.

(:glue-only t**)** Include only the "glue" code. You should use the **:server-extern** option when specifying this suboption to generate the server's function prototypes header file. Be sure to specify that this file is included in the other source files by using the **:include** suboption when generating those files.

(:submodule-name *submodule-name***)**
Name to include in the comment written at the front of the file.

rpc:define-remote-error-number *system-type number string* *Macro*

Defines a translation from operating system error codes to strings that can be used in error messages.

system-type is **:symbolics** or a keyword symbol that can be used as the system type of a host namespace object, such as **:unix42** or **:macintosh**.

With the **:unix42** system type, *number* is the UNIX error number potentially returned by a server function running on the UNIX system.

With the **:macintosh** system type, *number* is an error code used by that type of operating system and potentially returned (as the function value, not by means of RPCValues or RPCError) by a server function running on that system.

In the **:symbolics** case, *number* is an error code offset relative to first_Symbolics_error_code — these are error codes used by the Symbolics RPC software or by Symbolics RPC-based servers (these are distinct from operating system error codes). The operating-system-dependent values first_Symbolics_error_code and last_Symbolics_error_code indicate the range of error code numbers used for this purpose.

string is a description of the situation, which will be included in an error message.

6.2.3. The RPC Data Representation Layer

This layer defines a common representation for data. It translates representations used by different machines and by different programming languages to and from the common representation.

The data representation layer provides a mechanism for defining a data type by defining its common representation, its Lisp representation, and its C representation, along with code to translate between these representations. Several types are

predefined using this mechanism, and you can define additional types yourself. See the macro **rpc:define-remote-type**, page 70 for further information on defining remote types.

6.2.4. Predefined RPC Data Types

Atomic Types

rpc:integer-32 signed 32-bit integer

rpc:cardinal-32 unsigned 32-bit integer

rpc:integer-16 signed 16-bit integer (packed vector element)

rpc:cardinal-16 unsigned 16-bit integer (packed vector element)

rpc:integer-8 signed 8-bit integer (packed vector element)

rpc:cardinal-8 unsigned 8-bit integer (packed vector element)

rpc:character-8 ASCII character (packed vector element)

rpc:cardinal-4 unsigned 4-bit integer (packed vector element)

bit *array* &rest *subscripts*
unsigned 1-bit integer (packed vector element). Possible values are 0 or 1.

boolean *true* or *false*: in Lisp, **t** or **nil**, in C, 1 or 0.

single-float IEEE single-precision floating point

rpc:enumeration Similar to the remote type **member**, except the XDR value is explicitly given. In Lisp, this is done with an ALIST, in C, with an "=" in the enum declaration.

The common representation of each atomic type is 32 bits, except for those commented as packed. These are represented as 32 bits normally, but as 16, 8, 4, or 1 bits when elements of a vector.

Examples of Using the bit remote type and the rpc:enumeration remote type

Note that before you can use the following examples, you have to compile the following form:

```
(RPC:DEFINE-REMOTE-MODULE TPC-TEST
        (:NUMBER #X7F0080001) ; (or other appropriate number)
        (:VERSION 1)
        (:CLIENT :C :LISP)
        (:SERVER :LISP :C)
        (:ALLOW-UNTRUSTED-ACCESS NIL))
```

Example of a remote entry that uses the **bit** remote type:

```
(rpc:define-remote-entry bit-entry rpc-test
  (:number 13)
  (:arguments (x bit))
  (:values (y bit))
  (:lisp (:server (rpc:rpc-values x)))
  (:c (:server #{ RPCValues(x); }
  )))
```

This can be called by `(bit-entry 0)` or `(bit-entry 1)`.

Example of the **rpc:enumeration** remote type:

```
(defvar *coins* '((penny 1) (nickle 5) (dime 10) (quarter 25)
                  (half-dollar 50) (silver-dollar 100)))

(rpc:define-remote-type us-coin ()
  (:abbreviation-for
    `(rpc:enumeration ,@*coins*)))

(rpc:define-remote-entry enumeration-entry rpc-test
  (:number 15)
  (:arguments (coin us-coin))
  (:values (n-coins-make-a-dollar rpc:integer-8))
  (:lisp
    (:server
      (rpc:rpc-values (case coin
                        (penny 100) (nickle 20) (dime 10) (quarter 4)
                        (half-dollar 2) (silver-dollar 1) (t -1)))))
  (:c
    (:server
      #{ short int n = -1;
         switch(coin) {
           case penny: n=100;
             break;
           case nickle: n=20;
             break;
           case dime: n=10;
             break;
           case quarter: n=4;
             break;
           case half dollar: n=2;
             break;
           case silver dollar: n=1;
             break;
         }
         RPCValues(n);
       }
      ))
  )
```

This can be called by `(enumeration-entry 'penny)` or `(enumeration-entry 'quarter)`, for example.

Compound Types

structure *(field-name field-type) (field-name field-type)... (language structure-name sub-options...)...* *Remote Type*

A heterogeneous sequence of named fields each of any type. The common representation is simply a sequence of field representations in the order declared. Each field occupies an integral number of 32-bit words. The Lisp representation is a structure or an instance and the C representation is a struct.

The *language* options allow additional control over the representation in each language. Note that no suboptions are currently defined for the **:lisp** and **:c** options.

If no **:lisp** option is specified, the Lisp representation is a vector with one field per element. If the **:lisp** option is specified, the Lisp representation is a **defstruct** structure or a flavor instance, with slots (fields or instance variables) with the given field names. The remote type system automatically finds the correct accessor and constructor functions. The **defstruct** or **defflavor** of *structure-name* must be done before compiling anything that uses the remote type.

If no **:c** option is specified, the C representation is a struct automatically declared with the specified field names and types. If the **:c** option is specified, *structure-name* is a **typedef** already defined elsewhere (usually in an include file) with the specified field names and types. *structure-name* must be a **#{...}** expression.

Example a remote entry that uses the **structure** remote type:

```
(defflavor box ((height 0) (depth 0) (width 0)) ()
  :initable-instance-variables
  :readable-instance-variables)

(rpc:define-remote-type box ()
  (:abbreviation-for '(structure (:lisp box)
                                 (height rpc:cardinal-32)
                                 (depth rpc:cardinal-32)
                                 (width rpc:cardinal-32))))

(rpc:define-remote-entry structure-entry rpc-test
  (:number 16)
  (:arguments (b box))
  (:values (x box))
  (:lisp (:server (rpc:rpc-values s)))
  (:c (:server #{ RPCValues(s); } )))
```

This can be called by `(structure-entry (make-instance 'box))`, for example.

vector *element-type* &optional *length* *Remote Type*

(vector *element-type***)**
A variable-length sequence of elements, each of the same type. The common representation is a 32-bit word containing the number of elements, followed by the element representations. The Lisp representation is a vector. The C representation is a pointer to a struct with fields named length and element; element is an array of elements.

(vector *element-type length***)**
A fixed-length sequence of elements, each of the same type. The common representation is simply the element representations. The Lisp representation is a vector. The C representation is a pointer to an array of elements.

rpc:spread-vector *element-type* *Remote Type*

The same as **(vector** *element-type***)** except that in C this is passed around as two separate values, a pointer and a length, rather than as a single value, a struct with fields named length and element.

or *type type...* *Remote Type*

A discriminated union of several types. The common representation is a 32-bit word containing the zero-origin ordinal number of the type selected, followed by the representation of the value.

In Lisp, the types must be distinguishable by **typep**, as the representation is simply the value.

In C, the representation is a struct containing fields named type, the discriminant, and value, which is a union. The discriminant is of type enum with constants named *type*_type after each of the types. You have to be careful how you use this. Because C does not scope enumeration constants properly, it is possible to get name conflicts.

string &optional *length* *Remote Type*

(string) A variable-length string of ASCII characters. The common representation is a 32-bit word containing the number of characters, followed by the characters packed into 8-bit bytes. The Lisp representation is a thin-string. The C representation is a pointer to a struct with fields named length and element where element is an array of chars.

(string length**)** A fixed-length sequence of ASCII characters. The common representation is the characters packed into 8-bit bytes. The Lisp representation is a thin-string. The C

representation is a pointer to an array of chars.

Only the 95 ASCII characters should be used. In particular, do not use the carriage return character. Non-ASCII characters require more elaborate treatment, and there is no predefined type for them.

rpc:c-string *Remote Type*

Same as **string** except that the C representation is a pointer to a null-terminated string.

rpc:pascal-string *Remote Type*

Same as **string** except that the C representation is a pointer to a string whose first element is a character count.

rpc:opaque-bytes *options...* *Remote Type*

Data in its foreign-language representation.

In Lisp, this is a vector of unsigned 8-bit bytes. The option **(:length** *n-bytes***)** specifies a fixed length. If this option is not present, the length is variable. The option (*language typename*) specifies that the representation in *language* is the type *typename*. If this option is omitted, the representation in *language* is the same as in Lisp, for example, **(rpc:opaque-bytes (:length 8) (:c #{Rect})**.

Example of the **rpc:opaque-bytes** remote type:

```
(rpc:define-remote-entry opaque-bytes-entry rpc-test
  (:number 19)
  (:arguments (bytes (rpc:opaque-bytes (:length 3))))
  (:values (bites (rpc:opaque-bytes (:length 3))))
  (:lisp (:server (rpc:rpc-values bytes)))
  (:c (:server #{ RPCValues(bytes); } )))

(opaque-bytes-entry (make-array 3 :element-type '(unsigned-byte 8)
                                  :initial-contents '(1 2 3)))
```

list *element-type* *Remote Type*

Provides a way to transmit variable length lists. The Lisp representation is a list. The C representation is a series of linked `structs` with slots called `element` and `rest`. The element slot holds an object of type *element-type*. The rest slot holds a pointer to the next struct, or 0 if it is the end of the list.

Example of the **list** remote type:

```
(rpc:define-remote-entry list-entry rpc-test
  (:number 20)
  (:arguments (list (list rpc:cardinal-16)))
  (:values (l (list rpc:cardinal-16)))
  (:lisp (:server (rpc:rpc-values list)))
  (:c (:server #{ RPCValues(list); } )))

(list-entry '(100 200 300))
```

member *sym sym...* *Remote Type*

Similar to the **rpc:enumeration** remote type. In Lisp the representation is a symbol. In C, the representation is an enum. Note that, because C does not scope enumeration constants properly, name conflicts are possible.

Example of the **member** remote type:

```
(rpc:define-remote-entry member-entry rpc-test
  (:number 21)
  (:arguments (item (member foo :bar baz)))
  (:values (index rpc:cardinal-4))
  (:lisp
    (:server
      (rpc:rpc-values (ecase item
                        (foo 0) (:bar 1) (baz 2)))))
  (:c
    (:server
      #{ short int n;
         switch(item) {
           case foo: n=0; break;
           case bar: n=1; break;
           case baz: n=2; break;
         }
         RPCValues(n); }
      )))

(member-entry 'foo)
```

6.2.5. The RPC Data Type Extension Language

rpc:define-remote-type *name arglist* &body *options* *Macro*

Tells the data representation layer about a data type to be used for RPC arguments and/or values.

name is a symbol. *arglist* destructures the **cdr** of the remote type specifier when the remote type specifier is a list. When a remote type specifier is a symbol, it is the same as a list with a null **cdr**. The default for unsupplied optional arguments is **nil** (not * as in Lisp's **deftype**).

The variables bound by *arglist* are available within all forms in the options and suboptions of **rpc:define-remote-type**. The variables **type** and

rpc:original-type are also available; their values are remote type specifiers after and before **:abbreviation-for** expansion, respectively. The variable **rpc:language** is also available; its value is the current language.

The general options for **rpc:define-remote-type** are:

(:abbreviation-for *type***)**
Allows inheritance from another type. *type* is a Lisp form, usually a backquote expression, that evaluates to a remote type specifier. If this option is omitted, all the suboptions for all the languages you need must be included.

(:size *n-words***)** Specifies the number of 32-bit words occupied by the common representation of this type, if it is fixed, or **nil** if it is variable. Fixed-size types are a little more efficient because some computations can be done at compile time. *n-words* is a form that evaluates to a non-negative integer or to **nil**. If this option is omitted, it defaults to **nil**.

(:packed *unit***)** Specifies that values of this type can be elements of a packed vector. This implies that the C and Lisp representations are bit for bit identical to the XDR representation so that block-move techniques can be used.

unit is a Lisp form that evaluates to one of **:bit, :nibble, :byte, :halfword,** or **:word**, that is, 1, 4, 8, 16, or 32 bits respectively, or to **nil**, which disables packing. If this option is omitted, it defaults to **nil**.

(:signed *form***)** *form* evaluates to true if the values of this type can be packed negative numbers. The default is false.

(:prologue ((*var val*)...) *forms*...**)**
Before doing anything else, each variable *var* is bound to the result of evaluating the form *val*, and then the *forms* are evaluated. The *forms* should signal an error if the type parameters are not good.

The macro **rpc:type-error** may be helpful. It automatically inserts a comment about original-type into the error message if necessary. The variables bound here are available while evaluating forms in other options.

Two special options are

(:lisp (*sub-option arguments*...)...**)**

and

(:c (*sub-option arguments*...)...**)**

whose suboptions define the characteristics of this type specific to the Lisp or C programming language.

Suboptions for Lisp

Suboptions for Lisp all contain an argument list followed by a form that is evaluated to produce a Lisp form to perform some action. Backquote is typically used. The argument list receives arguments that are specific to the particular suboption.

(:size (*value*) *n-words*)
: Computes the number of XDR words needed to encode *value*. This suboption must be specified instead of the **:size** option if the type has a variable size common representation. If unspecified, this defaults from **:size** (the option), then from **:abbreviation-for**.

(:send (*value*) *code*)
: Stores the common representation of *value* in the transport medium.

  The form returned by the **:send** suboption of the **:lisp** option of **rpc:define-remote-type** can call any of the following macros to send data in the common representation.

  rpc:send-word *word*
  rpc:send-words &rest *words*
  rpc:send-word-vector *vector* &optional *start end*
  rpc:send-halfword-vector *vector* &optional *start end*
  rpc:send-signed-halfword-vector *vector* &optional *start end*
  rpc:send-byte-vector *vector* &optional *start end*
  rpc:send-signed-byte-vector *vector* &optional *start end*
  rpc:send-nibble-vector *vector* &optional *start end*
  rpc:send-bit-vector *vector* &optional *start end*
  rpc:send-char-vector *vector* &optional *start end*
  rpc:send-single-float-vector *vector* &optional *start end*

(:encode (*value*) *code*)
: Converts *value* to a single-word common representation and returns it. This is a convenient abbreviation for **:send**. If neither **:send** nor **:encode** is specified, suboptions inherited from **:abbreviation-for** are used.

(:receive (*variable storage-mode*) *code*)
: Receives the common representation of a value of this type from the transport medium and returns it. The arguments are normally ignored. However, if the type is an array, then *storage-mode* controls how storage is allocated. *storage-mode* is a compile-time test, not a runtime test. Possible values are:

nil — the value must be allocated in the heap, that is, normally.

:stack — the value is allowed to have dynamic lifetime (be allocated in the stack or share storage with the call block).

:overwrite — *variable* is already initialized to a value, so overwrite that storage. This is the only case in which the *variable* argument is not ignored.

The form returned by the **:receive** suboption of the **:lisp** option of **rpc:define-remote-type** can call any of the following macros to receive data in the common representation.

rpc:receive-word
rpc:receive-word-vector *vector* &optional *start end*
rpc:receive-halfword-vector *vector* &optional *start end*
rpc:receive-signed-halfword-vector *vector* &optional *start end*
rpc:receive-byte-vector *vector* &optional *start end*
rpc:receive-signed-byte-vector *vector* &optional *start end*
rpc:receive-nibble-vector *vector* &optional *start end*
rpc:receive-bit-vector *vector* &optional *start end*
rpc:receive-char-vector *vector* &optional *start end*
rpc:receive-single-float-vector *vector* &optional *start end*

(:decode (*word*) *code***)**

Converts *word*, a single-word common representation, into a Lisp value and returns it. This is a convenient abbreviation for **:receive**. If neither **:receive** nor **:decode** is specified, suboptions inherited from **:abbreviation-for** are used.

(:typep (*value*) *test***)** Tests whether *value* is a member of this type. This is used by a union (the **or** remote type specifier) to determine which remote type should be used when sending a Lisp value.

(:preprocess (*variable*) *code***)**

Does something to the value of *variable* that has to be done before **:send** and **:size**. *code* typically involves a setq of *variable*. If unspecified, this is inherited from **:abbreviation-for**, but if there is no inherited **:preprocess**, preprocessing does nothing.

(:optimizable-common-subexpressions (*value*) *form***)**

form evaluates to a list of Lisp forms that are common subexpressions that may be evaluated multiple times in computing the size and transmitting the value. A typical *form* would be **'((length ,:value))**. The RPC system

binds temporary variables to these forms, if necessary, and substitutes the variables for occurrences of the forms, in order to improve efficiency.

Suboptions for C

Suboptions for C contain an argument-list followed by a Lisp form that is evaluated to produce C code to perform some action. **#{...}** is typically used. For each suboption we specify whether the C code it produces is a statement or an expression. Statements must include a trailing semicolon. A "statement" can actually be multiple statements separated by semicolons, and can be no statements at all (NIL, an empty token list).

The argument-list receives arguments that are specific to the particular suboption; irrelevant trailing arguments can be omitted from the argument-list.

The Lisp form can call the function **rpc:declare-c-variable** to add declarations for temporary variables to the C function being constructed. Temporary variables have to be declared by side-effect, rather than just being included in the code being returned, because of the irregular structure of the C language — declarations cannot be nested inside expressions.

(:declare (*name*) *dcl***)**

A C declaration for the variable named *name* without a trailing semicolon. *name* is a token list, not a single token. Defaults from **:abbreviation-for**.

(:size (*value*) *n-words***)**

Computes the number of XDR words needed to encode *value*. *n-words* evaluates to a C expression. *n-words* can return a second value, which is a statement that must be executed before the expression can be evaluated. This suboption must be specified instead of the **:size** option if the type has a variable size common representation. If unspecified, this defaults from **:size** (the option), then from **:abbreviation-for**.

(:send (*value*) *statement***)**

Stores the common representation of *value* into the transport medium.

The statement returned by the **:send** suboption of the **:c** option of **rpc:define-remote-type** can call any of the following macros to send data in the common representation.

send_word *word*
send_word_vector *vector length*
send_halfword_vector *vector length*
send_signed_halfword_vector *vector length*

send_byte_vector *vector length*
send_signed_byte_vector *vector length*
send_nibble_vector *vector length*
send_bit_vector *vector length*
send_char_vector *vector length*
send_single_float_vector *vector length*

(:encode (*value*) *expression***)**

Converts *value* to a single-word common representation and return it. This is a convenient abbreviation for **:send**. If neither **:send** nor **:encode** is specified, suboptions inherited from **:abbreviation-for** are used.

(:receive (*variable storage-mode*) *statement***)**

Receives the common representation of a value of this type from the transport medium and stores the C value into *variable*. If the value is a pointer as opposed to a scalar value (some remote types use pointers as their C representation), then *storage-mode* controls how storage is allocated. *storage-mode* is a compile-time test, not a run-time test. Possible values of *storage-mode* are**:**

nil — the value must be allocated in the heap.

:stack — the value is allowed to have dynamic lifetime (be allocated in the stack or share storage with the call block).

:overwrite — the variable is already initialized to a value, so overwrite that storage.

The statement returned by the **:receive** suboption of the **:c** option of **rpc:define-remote-type** can call any of the following macros to receive data in the common representation.

receive_word
receive_word_vector *vector length*
receive_halfword_vector *vector length*
receive_signed_halfword_vector *vector length*
receive_byte_vector *vector length*
receive_signed_byte_vector *vector length*
receive_nibble_vector *vector length*
receive_single_float_vector *vector length*
receive_char_vector *vector length*
receive_bit_vector *vector length*
get_receive_pointer
advance_receive_pointer *n-words*

(:decode (*word*) *expression***)**

Converts *word*, a single-word common representation,

into a C value and returns it. This is a convenient abbreviation for **:receive**. If neither **:receive** nor **:decode** is specified, suboptions inherited from **:abbreviation-for** are used. **(:free** (*value*) *statement***)**
Disposes of heap storage, if any, occupied by *value*. This is used in combination with **:receive** with *storage-mode* = **:stack**. Thus, if **:receive** does not use the heap when *storage-mode* = **:stack**, **:free** can do nothing.

If unspecified, **:free** defaults from **:abbreviation-for**, and if no default is found that way, it defaults to a null statement (rather than signaling an error).

(:optimizable-common-subexpressions (*value*) *form***)**
form evaluates to a list of C expressions that are common subexpressions that may be evaluated multiple times in computing the size and transmitting the value.

Examples Using rpc:define-remote-type

Here is how the built-in type **rpc:cardinal-16** is defined:

```
(define-remote-type cardinal-16 ()
  (:size 1)
  (:packed :halfword)
  (:lisp (:encode (value) value)
         (:decode (value) value)
         (:typep (value) `(typep ,value '(unsigned-byte 16))))
  (:c    (:declare (name) #{unsigned short ↓name })
         (:encode (value) #{(long)↓value })
         (:decode (value) #{(unsigned short)↓value }
)))
```

Here is a very simple remote type definition that is just used as an abbreviation:

```
(rpc:define-remote-type answer ()
  (:abbreviation-for `(member :yes :no :maybe)))
```

Here is a MacIvory example of a simple remote type definition that just expands into the predefined **structure** type with appropriate arguments:

```
;; Macintosh points.  The Lisp representation is just (VECTOR V H).
(rpc:define-remote-type point ()
  (:abbreviation-for `(rpc:structure (v rpc:integer-16) (h rpc:integer-16)
                       (:c #{ Point }))))
```

Here is an example of a more complex remote type definition that uses the **:send** and **:receive** clauses. It does Macintosh points again, but in a less automatic way. The C representation is the predefined Point structure, the Lisp representation is a list of x and y, and the common representation is

two 32-bit words, first x then y. It's preferable to use **:abbreviation-for** to expand into a remote type that provides the **:send** and **:receive** routines, rather than writing your own, when possible.

```
(rpc:define-remote-type point ()
   (:size 2)
   (:lisp
     (:send (pt)
       `(progn (rpc:send-words (first ,pt) (second ,pt))))
     (:receive ()
       `(list (rpc:receive-word) (rpc:receive-word)))
     (:typep (pt)
       `(typep ,pt 'cons)))
   (:c
     (:declare (name)
       #{ Point ↓name })
     (:send (pt)
       #{ send word(↓pt .h);
          send word(↓pt .v); }
        )
     (:receive (pt)
       #{ ↓pt .h = receive word();
          ↓pt .v = receive word(); }
       )))
```

6.2.6. Tracing RPC Transactions

rpc:show-rpc-trace *n-newest-to-show* &optional *(n-newest-to-skip* **0***)* *Function*

Prints the RPC operations that were previously recorded in the trace buffer while tracing was enabled. If tracing is still enabled, this disables it before printing anything. *n-newest-to-show* is the number of events to print; if *n-newest-to-skip* is 0, these are the most recent events, otherwise the most recent *n-newest-to-skip* events are skipped and the *n-newest-to-show* older events are printed.

Note that even while the machine appears idle, RPC events to manage the screen are occurring. The best way to use this is to set up a process that calls **rpc:enable-rpc-trace**, does the operation of interest (or sleeps while another process does it), then calls **rpc:disable-rpc-trace**. This should minimize the intrusion of irrelevant screen management operations into the RPC trace buffer. At this point the information has been captured and **rpc:show-rpc-trace** can be called in any process, if necessary more than once, to print it out.

Each event is an outgoing call, an incoming call, an incoming reply to an outgoing call, or an outgoing reply to an incoming call. The information

printed for a call looks like:

```
5993 O Timer@151          [12]: Call 15943 to RPC-CONSOLE-SYNCH: 1 35111332245
```

5993 is the number of microseconds since the previous event. O (outgoing) is the direction of transmission. Timer is the name of the process. 151 is the depth in that process's stack. 12 is the size of the block of data being transferred, including control information. 15943 is the RPC transaction ID; use this to match calls and corresponding replies. RPC-CONSOLE-SYNCH is the name of the function that was called. What follows is the external data representation of the arguments, printed in octal and truncated to fit.

The information printed for a reply looks like:

```
1436 I RPC dispatch@97           [7]: Reply to 15943: 0 35111332245
```

1436, I (incoming), RPC dispatch, 97, 7, and 15943 are as explained above. What follows is the octal external data representation of the success/failure status (0 means success) followed by the values (if success) or error information (if failure).

rpc:enable-rpc-trace &optional *(buffer-size* **100***)* *Function*

Enables tracing of RPC operations. *buffer-size* is the number of operations to fit in the buffer; only the most recent calls and returns that happened while tracing is enabled are saved.

rpc:disable-rpc-trace *Function*

Disables tracing of RPC operations.

7. Sync-Link Gateways

This is a light-duty synchronous-link gateway facility for linking two Symbolics sites into a single network that can share files, send mail, copy worlds (albeit slowly), and exchange Converse messages using the Chaos and Internet (IP/TCP) protocols. (Sync-link gateways should be considered an additional Symbolics feature and not a full-service gateway.)

7.1. Hardware Requirements for Sync-Link Gateways

To use the sync-link gateways, the two sites must be connected by a dedicated synchronous link. In addition, the sites at each end of the link must have the following:

- Symbolics gate-array machine — Models 3610, 3620, 3630, 3650, or 3653 — to be used as a gateway. The processor load of being a gateway is light, approximately 5 percent, so this can be either a user or server machine.

- For gateway service at 9600 baud: a CSU/DSU modem connected to the **bulkhead** RS-232 port of the Symbolics machine by a male-to-female RS-232 extension cable.

- For gateway service at up to 56 kilobaud: a CSU/DSU modem plus an RS-232-to-V.35 interface converter such as the Black Box Model GA-IC221, or equivalent.

The alternative to the dedicated line — a switched dial line — is feasible, but not recommended.

The full hardware setup should be prepared before you enable the software.

7.2. Configuring Sync-Link Gateways

The sync-link gateway works by connecting two subnets through a third "subnet" that is the synchronous link. For example, assume that you have one site in Akron, one in Toledo and a trans-Ohio synchronous link. In fact, the two sites can be across the country or across the room. Only the two gateway systems are on the third subnet, but both must have addresses on that subnet. **Machines on all three of these subnets must have different Chaos and Internet addresses.**

Sync-link gateways can be configured such that there is one global network with one namespace or one network with two or more namespaces.

In general, if you have a site with several machines and are setting up a new satellite site with only a few machines, a single namespace is the better choice. On the other hand, if you have two long-standing sites with several machines, you

will probably find it preferable to keep the namespaces separate. For more information: See the section "Bootstrapping Sync-link Gateways", page 81. Gateways can be configured for Chaos-only networks, or for networks with Chaos and Internet. Configuration of these two kinds of gateways is closely related, but discussed separately.

7.2.1. Configuring Chaos-only Sync-Link Gateways

If the gateway is to use the Chaos protocol only, use the Namespace Editor to edit the host objects of both gateway machines as follows.

First, add a Chaos address for the gateway machine on the link. **This is different from the host's Chaos address on the local subnet.** That is, the gateway machine has addresses both on the local subnet and the synchronous link, which is also a subnet.

```
Address:  CHAOS 401
```

Now add a `Peripheral:` attribute for a Sync-Interface.

```
Peripheral: None Graphics-Tablet Kanji-Tablet Modem Pad
Sdlc-Interface Serial-Pseudonet Sync-Interface Other
  Unit: 1
  Baud: 300  600  1200  1800  2000  2400  3600  4800  7200  9600  19200  56000
  Chaos: 401
  Internet: an Internet address of the form A.B.C.D
  Clock-master: Yes  No
  Clock-constant: a decimal integer
  Rts-cts-protocol: Yes  No
```

The `Chaos:` attribute must have the same Chaos address number as the `Chaos Address:` attribute you just added.

The `Internet:` attribute is not used.

The `Clock-master:` and `Clock-constant:` attributes are not used. Timing is supplied by the dedicated synchronous link or the modem.

The `Rts-cts-protocol:` attribute should be **No** unless you are using a dial-up line, which is not recommended.

7.2.2. Configuring Sync-Link Gateways with Both Chaos and Internet

If the sync-link gateway is to use both the Chaos and Internet protocols, use the Namespace Editor to edit the host objects of both gateway machines as follows. You must have the Symbolics IP/TCP product to use the Internet protocol.

First, add a Chaos address and an Internet address for the gateway machine on the link. **These addresses are different from the host's Chaos and Internet addresses on the local subnet.** That is, the gateway machine has addresses both on the local subnet and the synchronous link, which is also a subnet.

```
Address:  CHAOS 401
Address:  INTERNET 128.81.1.1
```

Now add a `Peripheral:` attribute for a Sync-Interface.

```
Peripheral: None Graphics-Tablet Kanji-Tablet Modem Pad
Sdlc-Interface Serial-Pseudonet Sync-Interface Other
  Unit: 1
  Baud: 300  600  1200  1800  2000  2400  3600  4800  7200  9600  19200  56000
  Chaos: 401
  Internet: 128.81.1.1
  Clock-master: Yes  No
  Clock-constant: a decimal integer
  Rts-cts-protocol: Yes  No
```

The `Chaos:` attribute must have the same Chaos address number as the Chaos `Address:` attribute you just added.

The `Internet:` attribute must have the same Internet address number as the Internet `Address:` attribute you just added.

The `Clock-master:` and `Clock-constant:` attributes are not used. Timing is supplied by the dedicated synchronous link or the modem.

The `Rts-cts-protocol:` attribute should be **No** unless you are using a dial-up line, which is not recommended.

Now add a `Service:` attribute with the following service, medium, and protocol triple.

```
Service:  GATEWAY IP INTERNET-GATEWAY
```

Now add a `User Property:` attribute.

```
User Property: DEFAULT-INTERNET-GATEWAY hostname
```

In this case, *hostname* is the name of the host that serves as the Internet gateway to other networks. If you have no such system at your site, you do not have to supply this attribute.

7.3. Bootstrapping Sync-link Gateways

Before creating your sync-link gateways, you have to decide whether you want the two sites to share a single namespace or to have separate namespaces. In general, if you have a site with several machines and are setting up a new satellite site with only a few machines, a single namespace is the better choice. On the other hand, if you have two long-standing sites with several machines, you will probably find it preferable to keep the namespaces separate.

If you intend to have separate namespaces, all hosts you wish to have in communication must be running Genera 7.2 or a later release. If you intend to have a single namespace, only the gateway systems must be running Genera 7.2 or a later release.

Access is equal with either approach.

7.3.1. Bootstrapping Sync-Link Gateways with a Single Namespace

In general, if you have a site with several machines and are setting up a new satellite site with only a few machines, a single namespace is the better choice.

Here is the procedure to follow to set up gateways with a single namespace.

Both gateway machines must be running Genera 7.2 or a later release.

1. Create a host object for a gateway machine at the local site.See the section "Configuring Sync-Link Gateways", page 79.

2. Find a current world with namespace information in it. This is a world that has had Define Site or Set Site run on it, built after the gateway was configured in the namespace.

3. Load that world on the local gateway machine and boot it.

4. Take the same world to the new site and load it on the gateway machine there.

5. Boot the world at the new site.

6. Test the link using **zl:hostat**, specifying the octal Chaos address, not the host name. Use **tcp:send-icmp-echo**, specifying the Internet address in the format Internet|A.B.C.D, to test an Internet link.

7. Now you can make new worlds at the new site.

For an alternate technique: See the section "Bootstrapping Sync-Link Gateways with Separate Namespaces", page 82.

7.3.2. Bootstrapping Sync-Link Gateways with Separate Namespaces

If you have two long-standing sites with several machines, you will probably find it preferable to keep the namespaces separate.

Here are the constraints for this configuration:

- Both gateway systems must be running Genera 7.2 or a later release.

- All systems that you wish to be able to communicate across the gateways must also be running Genera 7.2 or a later release. Machines running earlier releases will be able to communicate within the local namespace, but when they are booted users will see notifications that the Global Network Name attribute is not recognized.

- Neither gateway server can be a primary namespace server in this configuration.

Here is the procedure to follow to set up sync-link gateways with separate namespaces.

1. Edit the host objects for the gateway systems at each site. See the section "Configuring Sync-Link Gateways", page 79. Reboot both systems or issue the Reset Network command on both systems.

2. Test the link using **zl:hostat**, specifying the octal Chaos address, not the host name. Use **tcp:send-icmp-echo**, specifying the Internet address in the format Internet|A.B.C.D, to test an Internet link.

3. Edit the Chaos and Internet network namespace objects at *both* sites and add a common Global Network Name to each object.See the section "The Global Network Name Network Attribute" in *Site Operations*.

4. Reboot the namespace server for one site.

5. Use **neti:find-site** at that site, naming the second site. Now the first namespace knows about the other.

6. Save the world on the namespace server. Never go back to a world built before this time.

7. Reboot the namespace server at the other site.

8. Use **neti:find-site** at that site, naming the first site. Now both namespaces are aware of each other.

9. Save the world at the second site.

10. Optionally, add each site name to the other site's search list. Whether you do this depends on the amount of traffic you expect between the sites. If you do not do this, users will have to specify the other site's name when communicating with it.

11. Optionally, make each gateway machine a secondary namespace server for the other. Remember that the gateway machine cannot be a primary namespace server if there are separate namespaces. There are two ways of making the gateway machines secondary namespace servers:

    - Use the `Secondary Name Server:` namespace attribute. At the first site, name the second site; at the second site, name the first site. This will cause any namespace search to search of all namespaces known to each sync-link gateway namespace.See the section "The Secondary Name Server Namespace Attribute" in *Site Operations*.

    - Use the `Default Secondary Name Server:` host attribute for the two gateway machines. At the first site, name the second site; at the second site, name

the first site. This will cause any namespace search to search only the two namespaces in the global network.See the section "The Default Secondary Name Server Host Attribute" in *Site Operations*.

For an alternate technique: See the section "Bootstrapping Sync-Link Gateways with a Single Namespace", page 82.

8. 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.

8.1. Namespace System Lisp Data Types

The various database data types are implemented on the Symbolics computer 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.

8.2. 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?" This variable can be queried for the name of the site as follows:

```
(send net:*local-site* :name) => :SCRC
```

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.

8.3. 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	Signals a **neti:object-not-found-in-search-list** error. This is the default.
nil	Returns **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>
   T

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

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, it 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 **zl:equal** returns **t**.
- 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"))))
```

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
```

net: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.

cl-neti::link-namespaces *site-to-find* &key *:merge-networks* *Function*

Enables a primary namespace server to find other namespaces not in the search rules, and to forcibly merge the networks between the two namespaces. Before attempting to use this function, you should consider exactly what your goal in terms of namespace administration is. If you have two namespaces you want to merge, it may be preferable to select one of them to be the final namespace and move the objects from the other into that one. Alternatively, if it is important to have two separate namespaces for administrative reasons but you want connectivity, you should consider using the Domain system to have the two sites know about each other. See the

section "The Domain System and the Namespace System", page 154. If you are unsure about these alternatives, please contack Symbolics Customer Support. **cl-neti::link-namespaces** replaces the namespace linking functionality of **neti:find-site**.

8.4. Messages to Namespace Names and Objects

8.4.1. 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.

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

8.4.2. Messages to net: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*

Looks up the object's *indicator* property. If it finds such a property, it returns the value; otherwise it returns **nil**.

:name *Message*

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

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

:primary-name *Message*

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

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

:names *Message*

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

```
(send net:*local-host* :names) => (#<NAME HARVARD|JUNCO 2346253>
                                   #<NAME HARVARD|J 2346267>
                                   #<NAME HARVARD|JUNKO 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"
```

8.5. Namespace Server Access Paths

For a definition of service access paths: See the section "Service Access Path", page 93.

Once the network system has computed a service access path for the **:namespace** service for a given host, it does not recompute that path again unless you use **neti:recompute-namespace-server-access-paths**, or **neti:recompute-all-namespace-server-access-paths**, or cold boot.

neti:show-namespace-server-access-paths &optional *namespace* *Function*

Displays the currently cached service access paths for the given *namespace*, along with the desirability of each path. If *namespace* is not given, only those service access paths for the local namespace are displayed.

neti:recompute-all-namespace-server-access-paths *Function*

Forces the generic network system to compute fresh service access paths for all namespaces instead of depending on paths computed earlier. It is necessary to use this function after altering the **host-protocol-desirability** site attribute to make the change take effect.

neti:recompute-namespace-server-access-paths &optional *namespace* *Function*

Forces the generic network system to compute fresh service access paths for the given *namespace*, instead of depending on paths computed earlier. If *namespace* is not given, this function operates on service access paths for the local namespace.

8.6. 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.

9. Interfacing to the Generic Network System

This section describes how to write programs that interface to the Generic Network System, including how to invoke network services in a program and how to implement new services.

It is also possible to write new mediums and even new types of networks. However, implementing new code at the medium and network level is considerably more complex than at the service level. For more information, see the section "Implementation of the Generic Network System", page 117.

9.1. How a Network Service is Performed

This section describes the course of events that takes place when a generic service is requested and performed:

1. A program on the user side makes a request for a generic network service.

   Usually the request occurs via **net:invoke-service-on-host** (used when the service must be performed by a particular host, such as for access to a file), or **net:invoke-multiple-services** (used when it is unimportant which network host provides the service, such as returning the time of day).

2. The user side tries to find a path to the service.

   When **net:invoke-service-on-host** is used, the generic network system on the user side tries to find the best path to the given service on the host. When **net:invoke-multiple-services** is used, the generic network system seeks multiple paths to the service. In either case, the application program that requested the generic service is not involved in finding the path; this is accomplished by the generic network system.

   The generic network system uses one of several functions to find a path. It uses **net:find-paths-to-service-on-host** when the service must be performed by a particular host, and **net:find-paths-to-service** when it is unimportant which host provides the service. These two functions return a *service access path*, a structure representing a path to a service on a host, including a description of the protocol and medium to be used. For more information, see the section "Service Access Path", page 93.

   The user side uses the namespace database to locate paths to services and hosts. For more information, see the section "Finding a Path to a Service on a Remote Host", page 109.

   If the generic network system cannot find a path to the service, the service cannot be performed. An error is signalled.

3. The user side gets the contact identifier for the service.

   The service access path describes the protocol and medium to be used; the next step is to find the contact identifier for that protocol. Each medium has a function that associates a contact identifier with a protocol.

Medium	*Function that Defines a Contact Identifier*
CHAOS	**chaos:add-contact-name-for-protocol**
TCP	**tcp:add-tcp-port-for-protocol**
UDP	**tcp:add-udp-port-for-protocol**
DNA	**dna:add-dna-contact-id-for-protocol**

   If the contact name is defined, the user side makes a request for connection to the server host (or hosts) on that contact identifier. If no contact name for this protocol is defined on the user host, an error is signalled.

4. The user side tries to make initial contact with the server side using the contact identifier.

   The server operating system examines the contact identifier and creates a server process. The operating system can either reject the request from the user side, or complete the connection. When the server side is a Symbolics computer, the same form that defines the contact identifier also identifies where the code that performs the service is defined. The server process finds that code (in a **net:define-server** form).

   If no server for this contact identifier is defined on the server host, an error is signalled (on the user host).

5. The user and server side exchange data.

   When the service is implemented with the generic **:byte-stream** or **:byte-stream-with-mark** medium, the user program often opens a stream via one of the possible mechanisms. The stream is set up to receive whatever information comes from the server side. When the **:datagram** medium is used, no stream is opened; instead, the server fills in an array with data and sends it to the user side.

   All actions of the user program are defined in the **net:define-protocol** form. All actions of the server program are defined in the **net:define-server** form.

6. The user side finishes its job.

   The user program typically processes the data in some way. If the **:byte-stream** or **:byte-stream-with-mark** medium is used, the user program closes the stream.

9.2. Invoking Network Services

This section describes the functions, variables, and data structures related to invoking network services. The primary data structures of interest are *service access paths* and *file access paths*:

See the section "Service Access Path", page 93.
See the section "File Access Path", page 94.

The functions and variables for invoking network services are:

net:invoke-service-on-host
: Provides the simplest way to invoke a network service. Appropriate when it is important which host should perform the service, such as for **:login** or **:file** service.

neti:*invoke-service-automatic-retry*
: Controls whether **net:invoke-service-on-host** automatically tries all paths.

net:invoke-multiple-services
: Provides a way to follow multiple paths to a service at once. Useful when it is unimportant which host provides the service; for example, for **:time** service.

net:find-paths-to-service
: Returns a list of service access paths for a given service on any hosts to which a path exists. Often used to compute service access paths for **net:invoke-multiple-services**.

For information on the lower-level functions that implement service lookup and invocation, see the section "Implementation of the Service Lookup Mechanism", page 138.

9.2.1. Service Access Path

Application programs request services using either **net:invoke-service-on-host** or **net:invoke-multiple-services**. The generic network system then has two steps to accomplish: find a path to the service, and invoke it.

A *service access path* is a structure that represents a path to a service on a host. It describes the name of the service, any arguments to the service, the server host, the protocol, the medium, and the desirability of the path.

Several functions used by the generic network system return one or more service access paths, including:

net:find-paths-to-service
net:find-path-to-service-on-host
net:find-paths-to-service-on-host
net:find-path-to-protocol-on-host
net:find-paths-to-protocol-on-host

For example:

```
(net:find-path-to-service-on-host :send (net:parse-host 'card))
-->#<SERVICE-ACCESS-PATH SEND (CONVERSE) -- CARDINAL on CHAOS 61631064>

(describe *)
-->#<SERVICE-ACCESS-PATH SEND (CONVERSE) -- CARDINAL on CHAOS 100156265>
     is a NETI:SERVICE-ACCESS-PATH
       NETI:SERVICE:         :SEND
       ZL:ARGS:              NIL
       NET:HOST:             #<LISPM-HOST CARDINAL 6406456>
       NETI:PROTOCOL:        #<PROTOCOL CONVERSE 245141204>
       NETI:MEDIUM:          #<MEDIUM-DESCRIPTION on CHAOS 100156261>
       NETI:DESIRABILITY:    0.9
       NETI:STREAM:          NIL
   #<SERVICE-ACCESS-PATH SEND (CONVERSE) -- CARDINAL on CHAOS 100156265>
   is implemented as an ART-Q type array.
   It uses %ARRAY-DISPATCH-WORD; it is 8 elements long.
```

Several functions used by the generic network system require one or more service access paths as an argument, including:

net:invoke-service-access-path
neti:most-desirable-service-access-path
net:start-service-access-path-future
net:service-access-path-future-connected-p
net:continue-service-access-path-future
net:abort-service-access-path-future

For information on how the generic network system finds one or more service access paths, see the section "Finding a Path to a Service on a Remote Host", page 109.

9.2.2. File Access Path

A *file access path* is an internal data structure that represents a path from one host (on which an application program requested **:file** service) to a file. That file can be stored on a remote host, or on the local FEP file system. If the file access path describes a path to a file on a remote host, part of the data structure is a service access path. For more information, see the section "Service Access Path", page 93.

A file access path is created by the generic network system when an application program performs an operation on a file. For example, when you give the Delete File command, the generic network system creates a file access path to the targeted file, and then invokes the "DELETE" operation on it.

The most important operation done on file access paths is the "OPEN" operation, which returns a stream. The type of stream depends on the network protocol being used and the arguments given to "OPEN".

In summary, when an application program performs file operations, a file access path provides a link to the file. Some file operations are performed directly on file access paths (such as "DELETE", "RENAME", and so on). When significant input or output is necessary, the program sends an "OPEN" command to the file access path, and receives a stream in return. The program then sends input or output commands to the stream, finally closing the stream.

When you select the Peek program, and click on [File Systems], the display shows the active and inactive file access paths. A file access path is represented as follows:

```
Host STONY-BROOK
  Access path to S using NFILE
```

You can click Left on "Access path to S using NFILE" for a menu of operations to perform on the access path, which typically includes:

Reset
Describe
Inspect

9.2.3. Functions for Invoking Network Services

The following functions (and variable) provide an interface to the part of the generic network system that finds paths to services and invokes them.

The internal functions that implement this mechanism are described elsewhere: See the section "Implementation of the Service Lookup Mechanism", page 138.

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

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

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

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**.

net:invoke-multiple-services *(service-access-paths timeout* &optional *whostate service-variable) (host-variable* &rest *results-variables)* &body *clauses* *Function*

A useful function for following multiple paths to a service at once. It starts up service futures for multiple hosts, and invokes the service on each host when the connection is complete. The argument *service-access-paths* includes the information on the services requested of the hosts.

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.

Note that unlike **net:invoke-service-on-host**, this function is not given *service-args*. **net:invoke-multiple-services** is intended for simple services that do not take arguments. If you need to invoke services that do take arguments on multiple hosts, you can use some of the internal functions in the generic networks system, such as **net:start-service-access-path-future**. See the section "Implementation of the Service Lookup Mechanism", page 138.

service-access-paths A form that will return a list of service access paths. Often this is 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.

results-variables 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))))
```

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. For example:

```
(net:find-paths-to-service :time)
```

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

Returns a list of service access paths for the particular service with access paths using broadcast. For example:

```
(net:find-paths-to-service-using-broadcast :time)
```

9.3. Defining a New Network Service

You can define a new network service built on the foundation of the generic network system, taking advantage of the layers of network software already in place on Symbolics computers. The new service should use a medium that is already defined.

The steps for defining a new network service are:

- Defining the the server host with **net:define-server**.
- Defining the client host with **net:define-protocol**.
- Editing the namespace database on the client host.
- Adding contact identifiers for the new service on both client and server hosts.
- Defining the client host for the **:local** medium on the client host (optional).

Note that when you build a server, if you are using a protocol (such as Eval, Send, Converse, NFILE, and so on) not requiring a specific medium, you can use a generic medium such as **:byte-stream**.

The functions for defining new client and server sides of protocols are described in detail at the end of this section: See the section "Functions for Defining Users and Servers", page 102.

Defining the Server Host

Use **net:define-server** for defining the server side of the protocol. You can build a server that handles all requests sent by servers using any **:byte-stream** medium. The generic **:byte-stream** medium works for all network media implementing **:byte-stream** network connections. Networks implementing the byte-stream medium include Chaos, TCP, DNA, Serial-pseudonet, SNA, XNS, and many more. You do not have to worry about the specifics of the underlying network as long as the server you build uses a **:byte-stream** medium.

Building a server that uses the byte-stream mediums enables you to avoid working with the low-level code for other protocols such as Chaos, TCP, and DNA, just to name a few. For example, building a **:byte-stream** server enables you to avoid working with the following for a Chaos server:

```
(net:define-server :example-1 (:medium :chaos ... )
   (server-function-1 network-connection-stream))
```

Additionally, building a **:byte-stream** server enables you to avoid working with the following for a TCP server:

```
(net:define-server :example-1 (:medium :tcp ... )
   (server-function-1 network-connection-stream))
```

Additionally, building a **:byte-stream** server enables you to avoid working with the following for a DNA server:

```
(net:define-server :example-1 (:medium :dna  ... )
   (server-function-1 network-connection-stream))
```

Defining the Client Host

After you define the server host protocol in the network stream, you have to define the client host protocol. Use **net:define-protocol** for defining the client side.

Editing the Namespace Database

When the network system determines whether the client host and the server host have a common path, the client host consults the **net:define-protocol** forms to determine the protocols supported for the desired service. The network system consults the namespace host object for the server host to determine which protocols and mediums the server host supports for the desired service. The network system then matches the protocols and mediums for the desired service on both hosts and chooses the most desirable path. Therefore, the namespace host object for the server host has to contain names for service, medium, and protocol equivalent to those in the **net:define-protocol** form evaluated on the client host.

```
(net:define-protocol :protocol (:service :medium) ...)
```

Typically, the **net:define-protocol** form contains a generic medium (such as **:byte-stream**), but the namespace host object indicates a specific medium, such as **:chaos** or **:tcp**. A protocol server defined to use the generic **:byte-stream** medium can use the specific medium **:chaos**, **:tcp**, or **:dna**.

Adding Contact Identifiers

The client host initially makes contact with the server host using a contact identifier specifying the service requested. Make sure the contact identifier is known to both hosts, and that it is the same on both hosts. The contact identifier serves two purposes:

1. It enables both hosts to communicate by informing the server host to listen on a specific contact identifier, and by informing the client host to request a service on that contact identifier.

2. It links the contact identifier with the actual code performing the service (on the server host) and the code requesting the service (on the client host).

The functions for adding contact identifiers are:

Medium	*Function for Defining Contact Identifier*
CHAOS	**chaos:add-contact-name-for-protocol**
TCP	**tcp:add-tcp-port-for-protocol**
UDP	**tcp:add-udp-port-for-protocol**
DNA	**dna:add-dna-contact-id-for-protocol**

Defining a Client Host for the :local Medium

You can define a user side for the **:local** medium, which is an optimization for the case when the user and server host are the same machine. There is no need to edit the namespace database for a local service. Note that defining a user side for the **:local** medium is an optional step.

9.3.1. Example of Defining a Server Using Network Server Code

1. You can specify the following code for handling all requests sent by client hosts using any **:byte-stream** medium, by supplying **:byte-stream** as the medium.

```
(net:define-server :example-1 (:medium :byte-stream
                               :stream network-connection-stream)
  (server-function-1 network-connection-stream))
```

   This code automatically adds an entry titled :example-1 to the list of known protocols. When you make a request using any **:byte-stream** medium for protocol :example-1, the connection completes and the local variable network-connection-stream is bound to the network connection.

2. Most network streams are buffered streams. If you write a line of code such as (**print** foo *stream*), the data is not sent; it is placed in a buffer and waits for further instruction. In order to send the data, you must place a **zl-user:force-output** operation in your program. For example, the following program adds two numbers and sends the result over the network:

```
(defun server-function-1 (stream)
  (let* ((num1 (read stream))
         (num2 (read stream)))
    (print (+ num1 num2) stream)
    (send stream ':force-output)))
```

3. You have to make sure that both hosts are closed at the proper time. If you do not specify a message that closes the stream, one or both of your hosts can remain open indefinitely. In order to avoid this situation, you can perform a closing operation within your program. Consider the following example:

```
(net:define-server :example-1 (:medium :byte-stream :stream
        network-connection-stream)
   (server-function-1 network-connection-stream)
   (send network-connection-stream ':close ':abort))
```

4. In order to make a server functional, you have to tell the network media when to use a given protocol.

   - For Chaos, you must associate a contact name (which is a string) with every protocol that you use.

     For example, you can inform the network medium of the Chaos protocol as follows:

     ```
     (chaos:add-contact-name-for-protocol :example-1)
     ```

   - For TCP or UDP, you must assign a port number to the protocol.

     For example, you can inform the network medium of the TCP protocol as follows:

     (tcp:add-tcp-port-for-protocol :example-1 *portnumber*)

     For more information concerning currently used port numbers, see the RFC for assigned numbers.

   - For a UDP server, you can inform the network medium of the UDP protocol as follows:

     (tcp:add-udp-port-for-protocol :example-1 *portnumber*)

     For more information concerning currently used port numbers, see the RFC for assigned numbers.

   - For DNA, you associate a contact name (a string) with the protocol.

     For example, you can inform the network medium of the DNA protocol as follows:

     (dna:add-dna-contact-id-for-protocol :example-1 *string*)

5. In order to define the protocol on the client host, you can use **net:define-protocol** as follows:

```
(net:define-protocol :example-1 (:example-1 :chaos)
  (:invoke-with-stream-and-close (stream num-1 num-2)
              (client-function-1 stream num-1 num-2)))
```

```
(defun client-function-1 (stream num1 num2)
  (print num1 stream)
  (print num2 stream)
  (send stream ':force-output)
  (read stream))
```

Note that you do not have to forcibly close any connections (the server closes the connections). The network system does a default close.

6. In order to execute a server on the server host, you inform the client host. Consider the following example:

   (net:invoke-service-on-host *service host args...*)

   You can also define a function enabling you to automate the procedure for executing a server on the server host. Consider the following:

```
(defun test-ex-1 (host number1 number2)
  (setq host (net:parse-host host))     ; make sure it's real
  (net:invoke-service-on-host :example-1 host number1 number2))
```

   You can now execute (test-ex-1 *host* 1 2), which returns 3.

For information on building servers using RPC code: See the section "Example of Creating a Simple UNIX Application for the UX" in *Open Genera User's Guide*.

9.3.2. Summary of Functions for Defining Users and Servers

net:define-protocol Defines the user side of a protocol.

net:get-connection-for-service
Can be used inside of an **:invoke** clause of **net:define-protocol** to get a network stream to the service on the correct medium.

net:define-server Defines the server side of a protocol.

chaos:add-contact-name-for-protocol
Associates a Chaosnet contact name with the protocol name.

neti:with-server-error-disposition
Creates an environment for handling errors within a server. Used in conjunction with **net:define-server**.

neti:change-server-error-disposition
Changes the error disposition for a server. Used in conjunction with **net:define-server**.

9.3.3. Functions for Defining Users and Servers

net:define-protocol *protocol-name (service-name base-medium)* *Special Form*
&body *options*

Defines the protocol *protocol-name*, a keyword symbol, which provides the generic network service *service-name*. *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 a generic medium such as **:datagram** or **:byte-stream**. It can also be **:local**, meaning that the protocol is not implemented in the network at all, but via some functions running on the local machine.

Each *option* is a list whose first element is a keyword. Exactly one of the following options is required to specify the type of protocol:

(**:invoke** *function*) When the service is invoked, *function* is called with the service access path as an argument. (The service access path is returned by the network system when it finds a path to the remote host; this happens automatically.)

(**:invoke** *lambda-list* &body *body*)
Defines a function to be called when a service is invoked. As with (**:invoke** *function*), this function is called with the service access path as its argument.

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

(**:invoke-with-stream** *lambda-list* &body *body*)
Defines a function which is called when the service is invoked. As with (**:invoke-with-stream** *function*), the first argument passed to this function is the stream, and the remaining arguments are the arguments to the service invocation. The code in *body* is responsible for closing the stream when it is done using the stream.

*(***:invoke-with-stream-and-close** *function)*
Similar to **:invoke-with-stream**, except that when *function* returns, the network system closes the stream, so the function need not do that.

(**:invoke-with-stream-and-close** *lambda-list* &body *body*)
Analogous to (**:invoke-with-stream** *lamdba-list* &body *body*) above, except that when *body* returns, the network system closes the stream, so *body* need not do that.

The following *options* are optional:

(**:desirability** *number*)
number is a number between 0.0 and 1.0 that describes how well this protocol provides the service. The default is 1.0.

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

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 104. 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*.

Higher-level protocols such as LOGIN and FILE provide their own mechanisms for informing the service system of the implementation of new protocols. These are macros that expand into a **net:define-protocol** form with suitable options. They are not documented.

The following example 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 is defined by the service itself. For some services, such as time, this is done via the returned value. For others, an error is signalled. This error can 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 formatted incorrectly (for example, does not contain exactly four data bytes).

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

The following example defines a simple remote EVAL protocol over TCP and Chaos. The purpose of the protocol is to ship a form to another host, evaluate the form, and ship the results back.

Note that this protocol is defined to use ASCII characters. This is so non-Symbolics hosts may define servers for this protocol. If ASCII compatibility is not an issue, removing **:ascii-translation t** from the stream creation options would allow the full Symbolics character set to be used.

Also note that this example uses a generic **:byte-stream** medium. This option gives both TCP and Chaos versions with no extra effort. The system handles the details.

```
(net:define-protocol :simple-eval (:simple-eval :byte-stream)
  ;; The list (stream :ascii-translation t :characters t)
  ;; tells the network system to bind STREAM to a
  ;; bidirectional network stream created by
  ;; NETI:GET-CONNECTION-FOR-SERVICE with options
  ;; :ASCII-TRANSLATION T, etc.
  ;; If you trust the defaults for
  ;; NETI:GET-CONNECTION-FOR-SERVICE, you can replace
  ;; this entire list with the symbol STREAM.
  ;;
  ;; The rest of the arglist represents the arguments
  ;; passed to INVOKE-SERVICE-ON-HOST.  In this case,
  ;; there is only one, the form to be evaluated.

 (:invoke-with-stream-and-close
    ((stream :ascii-translation t :characters t)
     form)
    ;; Print form and read result in known environment
    (with-standard-io-environment
      (print form stream)
      (force-output stream)
      (apply 'values (read stream)))))
```

Below is the local protocol. This is just an optimization in case this protocol is invoked with the intention of being executed on the local machine.

```
(net:define-protocol :local-simple-eval (:simple-eval :local)
  (:invoke (ignore form)
    ;; Read form and print result in known environment
    (with-standard-io-environment
      (eval form))))
```

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

Can be used inside of an **:invoke** clause of **net:define-protocol** 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 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.

net:define-server *protocol-name options* &body *body* *Function*

Define the top-level function of a network server. When a host receives a request for connection for this service, the generic network system creates a process, and the *body* given here is run in that process.

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. 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**, **:byte-stream-with-mark** 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 user host.

:error-disposition A keyword that determines what should happen if an unhandled error condition is signalled in the server. Valid error dispositions are:

nil or **:notify**
A notification is given on the server machine 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 to 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 host.

:network The value of this keyword is the name of a variable that is bound to the network object through which the 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 default. It causes the server request to be rejected if the host requesting 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 default. 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** and **:byte-stream-with-mark** medium types:

: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:

: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.

:token-list
A token list stream is constructed on the supplied medium, which must be **:byte-stream-with-mark**. For more information, see the section "Token List Transport Layer", page 165.

:translation
Enables you to specify the character set that the protocol uses. Possible values are **nil**, **:ascii**, or **:unix**. Note that specifying **nil** causes the protocol to use the Symbolics character set. For more information on character set translation, see the section "NFILE Character Set Translation", page 181.

: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 The value of this keyword is a list of variable names such as **: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 is 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 stores 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 use by any network protocols that take advantage of special features of Chaosnet, and would be inconvenient to implement over a generic medium. The following keyword is recognized for the **:chaos** medium type:

:conn 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* *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* *Function*

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 **net: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**.

9.4. Finding a Path to a Service on a Remote Host

This section describes how the Symbolics Generic Network system finds paths to a service. In this section, the user host is a Symbolics computer. When a service is requested, it is possible that the remote host has more than one way of providing the desired service; it is also possible that the remote host has no way of providing the service. The user host is responsible for determining which paths (if any) are possible, and choosing the most efficient path.

The user host must find the answers to the following questions:

- What kinds of connections is it capable of making?

  This question has two parts. First, which mediums and protocols does this host support for the desired service? Symbolics hosts store that information in **net:define-medium** and **net:define-protocol** forms. Second, which network connections are available to this host? The networks that a Symbolics host supports are listed in the **address** attributes of its host object.

- What kinds of connections is the server host capable of making?

  To determine which mediums and protocols the server host supports for the desired service, the user host consults the service attributes of the server's host object. To determine which networks are supported by the server host, the user host consults the **address** attributes of the server's host object. (When a service is requested locally, there is no need to consult the namespace database.)

When the user host has gathered all the required information, it generates a list of possible paths, and chooses the best path.

See the section "Implementation of the Service Lookup Mechanism", page 138.

9.4.1. Finding a Path to a Local Service

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 servers can be provided for the time-of-day service. Symbolics computers support a medium called **:local** for this purpose. When the **:local** medium is used, the user and server sides communicate by Lisp function calls, passing Lisp objects directly, rather than by sending bytes through a network.

When a service is requested that can be satisfied locally, there is no need to consult the namespace. Thus, there is no need to have a service attribute with the **:local** medium in its host object.

9.4.2. Determining What Kinds of Connections a Symbolics Computer Can Make

To answer the question "Which protocols and mediums does the local host support for the desired service?" the user host looks up all the **net:define-protocol** forms that define a user side for the desired service. If the desired service is **:file**, the host might find that it supports **:file** service as follows:

- With **:nfile** protocol and the **:byte-stream-with-mark** medium.
- With **:qfile** protocol and the **:chaos** medium.

To answer the question "Which networks does this host support?", the user host looks at the **address** attributes in its own host object. For example:

```
Address: Pair: CHAOS 413
Address: Pair: INTERNET 192.10.41.135
```

This host is on two networks: one is named CHAOS and the other is named INTERNET.

9.4.3. Determining What Kinds of Connections a Remote Host Can Make

It is the user host that must determine what kinds of connections the server host can make. In all networking environments, the user host has some mechanism for figuring out what services, protocols, and mediums are supported by the other hosts on the network. Symbolics computers use the namespace database for this purpose.

Specifically, the Symbolics computer consults the host object for the server host. To answer the question "Which mediums and protocols does the server host support for the desired service?", the user host looks at the service attributes. For example if the desired service is **:file**, these service attributes apply:

```
Service: Set: FILE CHAOS NFILE
Service: Set: FILE TCP NFILE
Service: Set: FILE CHAOS QFILE
```

This host can provide **:file** service using the **:nfile** protocol over the **:chaos** medium or the **:tcp** medium. It can also provide **:file** service using the **:qfile** protocol over the **:chaos** medium.

To answer the question "Which networks does the server host support?", the user host looks at the **address** attributes of the server's host object.

To see how the **:address** attributes are interpreted: See the section "Determining What Kinds of Connections a Symbolics Computer Can Make", page 110.

9.4.4. Finding the Possible Paths to a Host

To find paths to a remote host, the user host needs detailed information on the mediums it supports. The definition of a medium (in the **net:define-medium** form) describes in detail what criteria must be satisfied for a connection to be possible.

The definition of a medium includes a set of possible *implementations* of the medium. Each implementation describes a way to form a network connection using that medium. See the function **net:define-medium**, page 133.

Each implementation contains one or more *steps*. A one-step implementation is a way to connect directly to the server host. A two-step implementation is a way to connect first to a *gateway* (a host on more than one network); the gateway then connects to the server host. (A three-step implementation is a way to go through two levels of gateway. None of the defined mediums 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 in the **net:define-medium** special form. 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*)
: A connection is possible if the user host and the server host are both on the same network of type *network-type*. The connection can be formed directly over that network. For networks of type CHAOS, DIAL, or INTERNET, the "same network" means that the name of the network is the same (in the **address** attribute of the host object) for both hosts. For networks of type DNA, the area number must also be the same for both hosts.

(**:medium** *medium*)
: A connection is possible if the two hosts can connect with the specified *medium*.

(**:service** *service*)
: A connection is possible if a a connection can be formed to a server providing *service*, and that server can complete the remaining steps of the path.

For example, the following form defines the **:chaos** medium:

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

The **:chaos** medium includes only one implementation, which is a one-step implementation. To establish a **:chaos** or **:chaos-simple** connection to a target host, both hosts must be on the same **:chaos** network. (Note that the keyword **:chaos** is being used in two independent ways here: as a medium, and as a network type.)

For the purposes of this example, the following form defines the medium called **:tcp** and provides two implementations:

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

The first implementation is a one-step implementation; it says that you can establish a **:tcp** connection with a host if you are on the same **:internet** as it. The second implementation 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 second step is a **:medium** step. This allows many levels of gateway to be used.

This becomes clearer with an example: See the section "Example of Finding a Path to a Host", page 112.

9.4.5. Example of Finding a Path to a Host

This section provides an example to show how the user host finds a path to a desired service on the server host.

In this example, the host named Pokey requests **:file** service from the host named Gumby. Both Pokey and Gumby are Symbolics computers.

The request for **:file** service happened when the user of Pokey gave the Edit File command, and entered the pathname of a file stored on Gumby. Thus, Pokey is the user side and Gumby is the server side of this transaction.

Pokey needs to answer the question "Which protocols and mediums are supported locally for the desired service?" It checks the **net:define-protocol** forms, and finds that it supports three different user protocols for **:file** service:

- **:qfile** protocol over the **:chaos** medium
- **:nfile** protocol over the **:byte-stream-with-mark** medium
- **:dap** protocol over the **:dna** medium

Pokey supports the **:nfile** protocol over the generic **:byte-stream-with-mark** medium, which is built on **:byte-stream**, another generic medium. With generic mediums, it is necessary to find the set of specific mediums that support it; a generic medium is not sufficient in itself to make a connection. Each definition of a specific medium that implements **:byte-stream** (and hence, **:byte-stream-with-mark**) includes information on how Pokey can make a connection using that medium.

Pokey finds three **net:define-medium** forms that provide implementations for the **:byte-stream** medium: **:tcp**, **:chaos**, and **:dna**. Thus Pokey has determined that it supports **:nfile** over the **:tcp**, **:chaos**, and **:dna** mediums. Pokey thus supports the following user protocols and specific mediums:

- **:qfile** protocol over the **:chaos** medium
- **:nfile** protocol over the **:chaos** medium
- **:nfile** protocol over the **:tcp** medium
- **:dap** protocol over the **:dna** medium

Pokey must now answer the question "Which protocols and mediums are supported by the remote host for the desired service?" It checks the host object for Gumby, and sees the following two attributes:

```
Service: Set: FILE CHAOS QFILE
Service: Set: FILE CHAOS NFILE
Service: Set: FILE TCP NFILE
```

This indicates that Gumby supports the following server protocols for **:file** service:

- **:qfile** protocol over the **:chaos** medium
- **:nfile** protocol over the **:chaos** medium
- **:nfile** protocol over the **:tcp** medium

Pokey eliminates the **:dna** medium as a possibility because Gumby does not support a server side for **:dap** protocol over the **:dna** medium.

At this point there are two possibilities that Pokey must investigate: using the **:chaos** medium or the **:tcp** medium.

Pokey investigates the first possibility: using the **:chaos** medium. The definition of the **:chaos** medium contains a single implementation, which is:

```
(:network :chaos)
```

This implementation means that to establish a connection to a remote host using the **:chaos** medium, both hosts must be on the same Chaos network. Pokey must now determine whether Pokey and Gumby are on the same **:chaos** network. Pokey checks its own host object for the **address** attributes, and finds:

```
Address: Pair: CHAOS 1043
Address: Pair: INTERNET 192.10.41.135
Address: Pair: DNA 3.7
```

Pokey then looks at the host object for Gumby, and finds the following **address** attribute:

```
Address: Pair: PRIVATE-CHAOS 424
Address: Pair: INTERNET 139.5.17.135
```

Both Pokey and Gumby are on networks of type CHAOS. (To find out the type of a network, look in the network object for its **type** attribute.) Pokey is on a network called CHAOS, and Gumby is on a network called PRIVATE-CHAOS. Since the networks have different names, they are different Chaos networks. Thus Pokey eliminates the possibility of using the **:chaos** medium to connect to Gumby.

Pokey now considers the **:tcp** medium. The definition of the **:tcp** medium contains two implementations:

```
((:network :internet))
((:service :tcp-gateway) (:medium :tcp)))
```

The first implementation of the **:tcp** medium indicates that you can establish a **:tcp** connection with a host if you are on the same **:internet** as it. Pokey looks at the **address** attributes again to decide whether Pokey and Gumby are on the same Internet. They are both on the network named INTERNET, so they are on the same Internet network. Pokey has succeeded in finding the first possible path: using the **:tcp** medium to make a one-step connection to Gumby, over the same Internet network.

The second implementation of the **:tcp** medium 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.

Pokey searches the namespace database for hosts that provide **:tcp-gateway** service. This time Pokey is not asking for that service on a specific host, but on any host.

Pokey finds a host named Collie, whose host object contains the following service attribute:

```
Service: Set: TCP-GATEWAY CHAOS TCP-GATEWAY
```

(Note that Symbolics computers do not support **:tcp-gateway** service; Collie is a different kind of host.)

To be able to connect to Collie and request the **:tcp-gateway** service, Pokey must use the **:chaos** medium, and Pokey and Collie must be on the same Chaosnet. Collie's host object contains the following **address** attributes:

```
Address: Pair: CHAOS 1055
Address: Pair: INTERNET 192.10.41.266
```

Both Pokey and Collie are on the network named CHAOS. Pokey can request the **:tcp-gateway** service using the **:chaos** medium.

Pokey now investigates whether Collie can connect to Gumby using **:tcp**. Both Collie and Gumby are on the same Internet network, so this too is possible.

Pokey has succeeded in finding a second path to **:file** service on Gumby. Pokey can connect to Collie using **:chaos** medium. In turn, Collie can connect to Gumby using the **:tcp** medium.

It is up to Pokey to choose the more efficient of the two possible paths. Pokey chooses the one-step path (using **:tcp** to connect directly) rather than the more time-consuming two-step path (using **:chaos** to connect to Collie, and then **:tcp** to connect to Gumby).

For information on how application programs can interface to this mechanism: See the section "Invoking Network Services", page 93.

For more details on the implementation of the mechanism described here: See the section "Implementation of the Service Lookup Mechanism", page 138.

9.4.6. Desirability of Network Protocols

When you request a network service the Symbolics generic network system finds the possible paths to that service. When more than one path to the service exists, the generic network system tries to choose the most efficient path. The network system computes a number representing the *desirability* of each path.

Desirability is a floating-point number between 0 and 1. When computing desirability, the network system takes into account three factors: the desirability of the protocols (as indicated in the **net:define-protocol** forms), the **host-protocol-desirability** attribute of site objects in the namespace, and per-network dynamic information.

The relative desirability factors of the various Symbolics network protocols are as follows:

- IP/TCP protocols have the highest desirability.
- Chaos protocols are less desirable than IP/TCP.
- DNA protocols are less desirable than Chaos.

You cannot change the desirability of the protocol, or the dynamic information. But you can alter the desirability factors at your site by entering a value for the **host-protocol-desirability** site attribute in the namespace database. See the section "The Host Protocol Desirability Site Attribute" in *Site Operations*.

10. Implementation of the Generic Network System

This section describes the internals of the network system, including the implementation of packets (the basic unit of communication), interfaces (software to move packets from one machine to another), networks, mediums, the service lookup mechanism, and servers. Before reading this section, you should be familiar with the standard issues involved with implementing networks.

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 described elsewhere: See the section "Interfacing to the Generic Network System", page 91.

Note that the term "protocol" is used in this section to mean something different than it does at the higher network levels. In this section, protocols are at a lower level than mediums.

10.1. Packets

Packets are the basic unit of communication between network hosts. The Symbolics computer implements a packet as an array of fixnums, typically **sys:art-8b** or **sys:art-16b**. A Chaosnet packet is a **sys:art-16b** array, but a TCP packet might be a **sys:art-8b** array.

sys: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 a **sys:art-string** "packet" out of the data portion of the Chaos packet.

10.1.1. 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; users request packets from that pool, and later return packets to it.

This section describes the implementation of packets and provides some of the design considerations.

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 pool is wired, and thus acceptable to the microcode; the other pool is available to users and networks, and 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 and are wired before transmission.

- Have one pool of packets that are always wired.

- Have two pools of packets: one pool is wired and acceptable to the microcode; the second pool is composed of packets that are created and wired when needed. 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 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 measures had to be taken to ensure that applications and protocol implementations deallocated all packets.

The Release 6.0 network implementation used the fourth possibility; it is still in use now. 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 should 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 an error occurs during *do some things* and the function is exited, the **unwind-protect** frees 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. *Do some more things* is not allowed to use the packet (because it is supposed to have been freed) and the **unwind-protect** clause does not free the packet, both because the variable **packet** was set to **nil**.

10.1.2. Functions Related to 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**, **neti: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 127.

neti: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 **neti:allocate-packet-buffer**, coerces the packet to its own format and fills in default fields. See the section "Example of Programming with Packets", page 122.

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 119.

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. **neti: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 **neti:deallocate-packet-buffer**) or which can adjust meters and do other internal bookkeeping.

neti:raw-packet-buffer-size *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 can 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.

10.1.3. Subpackets and Coercing Packets

The packet that **neti:allocate-packet-buffer** returns is a **sys:art-8b** array of some length that is dependent on the architecture of the machine: See the variable **neti:raw-packet-buffer-size**, page 119. Raw **sys:art-8b** arrays are insufficient for some network purposes. For example:

- Chaosnet views the packet as 16-bit words, so it prefers an **sys:art-16b** array. Chaosnet also views the data portion of a 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 often 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**. However, 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**. 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 **neti:get-sub-packet**. See the section "Example of Programming with Packets", page 122.

The *leader-length* argument to **neti:get-sub-packet** is not required. If it is not supplied, the system supplies its own. Subpackets always have a fill-pointer that is available for general use. The *named-structure-symbol* argument to **neti:get-sub-packet** is also not required.

neti:get-sub-packet creates new displaced arrays only if it is necessary. When it is necessary to create a new subpacket with specific attributes, **neti:get-sub-packet** caches the information in the packet buffer. The next time the same attributes are requested, **neti:get-sub-packet** returns the cached subpacket instead of creating a new one.

Note: When using **sys: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 used by PDP-10s, PDP-20s and 68000s. Chaosnet is a little-ender protocol, but the DoD Internet Protocol (IP) and the DoD Transmission Control Protocol (TCP) are big-ender protocols. Thus, 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 subpacket. 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.

10.1.4. Example of Programming with Packets

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 a **sys:art-16b** array. We view the data, however, as a string (an array of type **sys: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 packet's destination
  source-address        ;protocol address of 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))
```

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 can be deallocated when it is on a transmit list (**neti:deallocate-packet-buffer** is careful), but packets cannot 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 cannot 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 or allocated packet buffers, not just free packet buffers. For example:

```
(neti:map-packet-buffers #'print)
```

prints 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. **neti: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:

- **neti: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.

10.2. Network Interfaces

In this discussion, interface 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.

10.2.1. 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 and networks do not automatically start sending packets back and forth; they are explicitly informed about each other. Specifically, for each interface in **neti:*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.

10.2.2. 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", page 129.

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.

10.2.3. Miscellaneous Notes on 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 **neti: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 additional byte offset, depending on the transmit needs of the interface.

10.3. Implementation of Networks

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.

10.3.1. Defining a Network

Networks are represented as flavor instances. Networks that are in the namespace database are based on the **net: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 **net: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 **net: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 **net: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 (:type magic-network) () ':magic)
```

10.3.2. Implementation of Network Addresses

People usually refer to hosts by textual names. Applications usually convert the name into a host object by calling **net: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 hu-

man 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 **sys:art-8b** array. The method of the **net: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 **net: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 (:parse-address chaos-network) (string)
  (parse-number string 0 nil 8 t))

(defmethod (:unparse-address chaos-network) (address)
  (format nil "~O" address))
```

10.3.3. Invoking Mediums

Once a path to the service is chosen, the service lookup mechanism 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.

See the function **net:define-medium**, page 133.

10.3.4. Packet Reception

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.

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 of Programming with Packets", page 122.

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**.

10.3.5. Packet Transmission

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 126.

10.3.6. Initializing, Resetting, and Enabling Networks

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 **neti:*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, it will not be automatically put on **net:*local-networks***. To interact with global network operations, the network should add itself to **net:*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*

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 **net:*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.

10.3.7. Byte Stream Conventions

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.

10.3.8. 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 a **sys: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 (:address-resolution-parameters chaos-network) ()
  (values #x+0804 2 ':little 0))
```

10.3.9. Interaction with Peek Network Mode

The Peek program can maintain visual information about networks and interfaces.

Networks that are not based on the **net: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 **net: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 **net:network** base flavor returns **nil**.

10.4. Implementation of Network Mediums

Network mediums are defined with the special form **net:define-medium**:

net:define-medium *medium types* &body *implementations* *Function*

Defines a medium named *medium*, which supports *types*, which is either a list of mediums, or an empty list. When defining a generic medium, *types* is often an empty list. For example, the following forms define the generic mediums **:byte-stream** and **:datagram**:

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

When defining a specific medium that supports one or more generic mediums, *types* contains the names of the generic mediums supported. For example, this form defines the **:chaos** medium, which is a specific medium that supports two generic mediums, **:byte-stream** and **:byte-stream-with-mark**:

```
(define-medium :chaos (:byte-stream :byte-stream-with-mark)
  implementations...)
```

An element of the body can either be an implementation or a list of the following form:

(*implementation lambda-list . body*)

This 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.

Each implementation describes a way to form a network connection using this medium. Each implementation contains one or more *steps*. A one-step implementation is a way to connect directly to the server host. A two-step implementation is a way to connect first to a *gateway* (a host on more than one network); the gateway then connects to the server host. (A three-step implementation is a way to go through two levels of gateway. None of the defined mediums 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 in the **net:define-medium** special form. 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*)
: A connection is possible if the user host and the server host are both on the same network of type *network-type*. The connection can be formed directly over that network. For networks of type CHAOS or INTERNET, the "same network" means that the name of the network is the same (in the **address** attribute of the host object) for both hosts. For networks of type DNA, the area number must also be the same for both hosts.

(**:medium** *medium*)
: A connection is possible if the two hosts can connect with the specified *medium*. See below for additional notes on the syntax of a **:medium** step.

(**:service** *service*) A connection is possible if a connection can be formed to a server providing *service*, and that server can complete the remaining steps of the path.

The syntax of an encapsulating **:medium** step is:

```
(((:medium underlying-medium))
     (service-access-path-arg
      underlying-connection/connection-args
      {connection-args}) . body)
```

service-access-path-arg is a variable that is bound to the service access path.

underlying-connection/connection-args may be a symbol or a list. If it is a symbol, it is bound to the underlying connection obtained via *underlying-medium*. If *underlying-medium* is a stream medium, this is a stream.

If *underlying-connection/connection-args* is a list, it is of the form:

(*underlying-connection* {*underlying-connection-args*})

underlying-connection is as above. {*underlying-connection-args*} are passed to the stream as connection arguments. Note that they must be compile-time constants.

Here is an example of this syntax:

```
(define-medium :byte-stream-with-mark ()
  (((:medium :byte-stream))
    (ignore (raw-stream :characters nil) &rest connection-args)
   (make-instance (if (get (locf connection-args) :token-list)
                      'buffered-token-stream
                      'buffered-stream-with-mark)
                  :raw-stream raw-stream)))
```

Normally, a medium with a **:medium** step receives the following arglist:

(*service-access-path stream* &rest *args*)

However, you can include the form **(declare (neti:call-with-medium t))** in the body of the medium step, which makes the arglist:

(*service-path medium* &rest *args*)

This allows the medium function to obtain its own connection. For more information,

see the section "Examples of Defined Mediums", page 136.

:byte-stream-with-mark Medium

The following form defines the generic medium **:byte-stream-with-mark**:

```
(define-medium :byte-stream-with-mark ()
  (((:medium :byte-stream)) (ignore (raw-stream :characters nil)
                                    &rest connection-args)
   (make-instance (if (get (locf connection-args) :token-list)
                      'buffered-token-stream
                      'buffered-stream-with-mark)
                  :raw-stream raw-stream)))
```

:chaos Medium

The following form defines the **:chaos** medium:

```
(define-medium :chaos (:byte-stream :byte-stream-with-mark)
  (((:network :chaos)) (service-access-path &allow-other-keys &key
                                            byte-size (characters t)
                                            &rest args)
   ;;++ futures
   (setf args (si:rem-keywords args '(:byte-size)))
   (lexpr-funcall #'open-stream
                  (neti:service-access-path-host service-access-path)
                  (get-chaos-contact-name-for-protocol service-access-path)
                  :byte-size (and (not characters) (or byte-size 8))
                  args)
   ))
```

:chaos is a specific medium that supports two generic mediums: **:byte-stream** and **:byte-stream-with-mark**.

The **:chaos** medium includes only one implementation, which is a one-step implementation. To establish a **:chaos** connection to a target host, both hosts must be on the same **:chaos** network. (Note that the keyword **:chaos** is being used in two independent ways here: as a medium, and as a network type.)

:chaos-simple Medium

The following form defines the **:chaos-simple** medium:

```
(define-medium :chaos-simple (:datagram)
  (((:network :chaos)) (service-access-path &rest connection-args)
   (let ((host (neti:service-access-path-host service-access-path))
         (contact-name (get-chaos-contact-name-for-protocol
                         service-access-path)))
     (if (eq host ':broadcast)
         (lexpr-funcall #'open-broadcast-simple-stream contact-name
                                                       connection-args)
         (lexpr-funcall #'open-simple-stream host contact-name
                                                 connection-args)))))
```

:tcp Medium

The following form defines the medium called **:tcp**:

```
(define-medium :tcp (:byte-stream)
  (((:network :internet)) (service-access-path &rest connection-args)
   (multiple-value-bind (host network ignore)
       (neti:decode-service-access-path-for-medium service-access-path)
     (ignore network)
     (let* ((protocol-name (neti:protocol-name
                            (neti:service-access-path-protocol
                              service-access-path)))
            (port-number (tcp:protocol-name-tcp-port protocol-name t)))
       (cl:apply #'tcp:open-tcp-stream host port-number
                                       nil
                                       connection-args))))
  ((:service :tcp-gateway) (:medium :tcp))
  ((:service :byte-stream-gateway) (:medium :tcp)))
```

:tcp is a specific medium that supports one generic medium: **:byte-stream**.

This form defines three implementations of the **:tcp** medium. The one-step implementation of the **:tcp** medium is:

```
(:network :internet)
```

This implementation says you can establish a **:tcp** connection with a host if you are on the same **:internet** as it.

The two-step implementations are:

```
((:service :tcp-gateway) (:medium :tcp))
((:service :byte-stream-gateway) (:medium :tcp))
```

These implementations say that you can establish a **:tcp** connection by finding a path to any gateway host that provides either the **:tcp-gateway** or the **:byte-stream-gateway** service, and that can, itself, form a **:tcp** connection to the target host. Note that the second step is a **:medium** step. This allows many levels of gateway to be used.

:pseudonet Medium

The **:pseudonet** medium always uses a gateway to access a network of type **:gateway-pseudonet**. This is used for accessing hosts that are not really on a network but are connected to some other host via something weaker, such as serial lines.

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

10.5. Implementation of the Service Lookup Mechanism

This section describes the internal functions and variables that are used by the generic network system when the Symbolics computer is requesting a service from another host. Thus in this section the Symbolics computer is the user side. For information on activities performed when the Symbolics computer is the server side: See the section "Starting Network Servers", page 142.

10.5.1. Summary of Functions for Service Lookup and Invocation

The user interface for looking up and invoking services is described elsewhere: See the section "Invoking Network Services", page 93.

Finding Paths to Services and Protocols

A *service access path* is a structure that represents a path to a service on a host. It describes the name of the service, any arguments to the service, the server host, the protocol, the medium, and the desirability. See the section "Service Access Path", page 93.

Note that the functions that find paths are not given *service-args*, because the mechanism that finds service access paths does not implement a very fine weeding-out process. The namespace database knows whether network protocols and hosts implement a service, but does not contain information on whether that service can be performed under some restricted set of arguments. Thus *service-args* are given only to the functions that invoke services.

net:find-paths-to-service-on-host
: Returns a list of all possible service access paths for a particular service on a given host.

net:find-path-to-service-on-host
: Returns a single service access path for a particular service on a given host, or signals an error if none can be found.

net:find-paths-to-protocol-on-host
: Similar to **net:find-paths-to-service-on-host**, except that the protocol itself is specified.

net:find-path-to-protocol-on-host
: Similar to **net:find-path-to-service-on-host**, except that the protocol itself is specified.

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

neti:most-desirable-service-access-path
: Takes a list of service access paths sorted by desirability and randomly chooses one from the equally desirable subset at the front. This distributes the server load evenly in the long run.

Service Futures

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
Initiates a request for service on a given service access path.

net:service-access-path-future-connected-p
Takes a service path previously given to **net:start-service-access-path-future** and returns **t** if the connection is now complete.

net:continue-service-access-path-future
Takes a service access path that is connected, and returns the values that invoking the service would. If the connection was not completed successfully, an error is signalled.

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

10.5.2. Functions for Service Lookup and Invocation

The functions and variables that provide a user interface for invoking network services include:

net:invoke-service-on-host
neti:*invoke-service-automatic-retry*
net:invoke-multiple-services
net:find-paths-to-service

They are described elsewhere: See the section "Functions for Invoking Network Services", page 95.

net:find-paths-to-service-on-host *service host* &optional *only-need-best must-have-one* *Function*

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

```
(net:find-paths-to-service-on-host :time (net:parse-host "bronx"))
```

If *only-need-best* is supplied and non-**nil**, this indicates that we are going to use the best path only, which saves time searching for many longer paths.

If *must-have-one* is supplied and non-**nil**, this function signals an error if no paths are found. Otherwise **nil** is returned.

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

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

```
(net:find-path-to-service-on-host :time (net:parse-host "bronx"))
```

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.

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.

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.

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 implemented locally, 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 looking for only 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.

10.5.3. Messages Related to Service Lookup

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. See the section "Desirability of Network Protocols", page 115.

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 **net: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 **net: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 **net:network** base flavor returns **nil**.

10.6. Starting Network Servers

This section describes the actions taken by a Symbolics computer when it is the server side of a connection, responding to a request for a network service from another host. For information on activities performed when the Symbolics computer is the user side, see the section "Implementation of the Service Lookup Mechanism", page 138.

10.6.1. 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.

10.6.2. Calling the Server Function

The function **neti:funcall-server-internal-function** is called to set up for calling the server 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**.

10.6.2.1. Commonly Used Arguments to Servers

This section describes several commonly used arguments to servers. You can use **neti:server-argument-descriptions** to find out what arguments a server takes.

:reject-unless-trusted
: If this property is non-**nil** and the host requesting the service is not trusted, the request for the service should be refused.

:trusted-p
: If this is one of the arguments to the server, **:trusted-p** and a determination of the requesting host's trustedness should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

:host
: If this is one of the arguments to the server, **:host** and the host object for the foreign host should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

:network
: If this is one of the arguments to the server, **:network** and the network invoking the server should be one of the keyword-value pairs given to **neti:funcall-server-internal-function**.

10.6.2.2. Commonly Used Arguments to Mediums

The major dispatch is based on which medium the server uses. Networks can support several generic mediums: **:byte-stream**, **:byte-stream-with-mark**, and **:datagram**. A network can also implement network-specific mediums and network-specific servers that use them.

If the server uses the **:byte-stream** or **:byte-stream-with-mark** 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 a **sys:art-8b** or **sys: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, three additional keyword-value pairs are supplied.

- **:request-array** is a **sys:art-8b** or **sys: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.

10.6.3. Functions Related to Starting Servers

The following functions and variables are used by Symbolics computers that are responding to a request from another host. The Symbolics computer is the server side of the connection.

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 containing any information the server might need to know. **neti: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.

11. Network, Medium, and Protocol Descriptions

This chapter describes four types of networks: Chaosnet, Dialnet, Internet, and DNA. All Symbolics computers are equipped to support Chaosnet. All Symbolics computers have the software to support Dialnet; however, a modem is also needed to use Dialnet.

Sites that purchase the optional IP/TCP software package can support Internet networks. Similarly, sites that purchase the optional DNA software package can support DNA networks. A DNA network is one in which hosts communicate using DECnet protocols.

The Internet and DECnet protocols are fully documented by other sources. See the section "References to IP/TCP Protocol Specifications", page 156.

See the section "References to DECnet Protocol Specifications", page 158.

In addition to describing the four types of networks, this chapter contains protocol specifications for the BYTE-STREAM-WITH-MARK network medium, the token list transport layer, the NFILE file protocol, and two namespace protocols. All Symbolics computers support these protocols.

11.1. Internet Networks

11.1.1. Introduction to Internet Networks

In Symbolics terminology, *Internet* is a type of network. If a site supports Internet:

- The site's namespace database has a network object of type Internet.
- One or more hosts have Internet addresses; the addresses are stored in the **address** attribute of the host objects. See the section "How to Obtain an Internet Address", page 34. See the section "Format of Internet Addresses", page 27.
- Hosts can communicate with other hosts on the Internet using standard IP/TCP protocols; the known protocols are stored in the **service** attributes of the host object.

The optional IP/TCP software package enables Symbolics computers to communicate with IP/TCP protocols. These protocols are listed elsewhere: See the section "TCP and UDP Protocols Supported by Symbolics Computers as Users", page 46. See the section "TCP and UDP Protocols Supported by Symbolics Computers as Servers", page 47.

Two kinds of sites could take advantage of the IP/TCP software package:

- A site that has other computers that can communicate with IP/TCP protocols, but cannot communicate with Chaosnet; the IP/TCP software package would enable the Symbolics computers at the site to communicate with the other hosts.

- A site that has hosts connected to the ARPA Internet; the IP/TCP software package would enable the Symbolics computers at the site to have ARPA Internet access as well.

Extensive documentation on IP/TCP protocols and other aspects of Internet is made available by the ARPA Network Information Center. For more information: See the section "References to IP/TCP Protocol Specifications", page 156.

The document *Symbolics IP/TCP Software Package* describes the installation and site configuration procedure.

11.1.2. Internet Domain Names

11.1.2.1. Introduction to Internet Domain Names

The Internet Domain Names system is a collection of specifications and procedures which implement the DOMAIN protocol, which is commonly used on the ARPA Internet. The DOMAIN protocol deals extensively with naming. It was created to address several problems.

One major problem that the Domain system addresses is the management of the ever-growing number of hosts on the Internet. When there were only a few hundred hosts, it was reasonable to keep a master file of hosts in a central location to be copied across the network periodically. As more and more hosts were registered, the Internet administrators found that they wanted to separate the hosts into smaller administrative units. Information about these hosts would then be maintained locally. As a result, the Domain system places these hosts in a tree-structured administrative system.

The second major problem that the Domain system attempts to address is the difficulty encountered when sending mail between different networks. Each network has a different naming scheme. These different naming schemes have hindered the interconnection of various networks. The Domain system attempts to allow connections between networks as diverse as Internet, CSnet, BITnet, UUCP, Symbolics Dialnet, and others.

For instance, in the past, mail addresses looked like:

- *user%host*.CSnet@CSnet-Relay.ARPA
- *random-host*!*uninteresting-host*!*host*!*user*@UCBVAX.ARPA
- adi/*user%host*.BITnet@WISCVM.ARPA

When addresses are automatically generated by various mailers, the results can be combined to make long and complex addresses.

If all the hosts involved are using the Domain system, all these mail addresses may be viewed as:

- *user@domain*

Symbolics implements the Domain specification described in several Requests for Comments (RFCs) available from the Network Information Center, SRI International. Symbolics implements the Domain specification on both TCP and Chaosnet. See the section "References to IP/TCP Protocol Specifications", page 156.

11.1.2.2. How the Domain System is Structured

Domains are administrative entities. There are no geographical, topological, or technological constraints on a domain. The hosts in a domain need not have common hardware or software, nor even common protocols. Most of the requirements and limitations on domains are designed to ensure responsible administration.

The Domain system is a tree-structured global namespace that has a few top-level domains. The top-level domains are themselves subdivided into domains. These domains can be further subdivided into yet more domains, and so on.

The administration of a domain requires controlling the assignment of names within that domain and providing access to the names, addresses, and list of valid services to users both inside and outside the domain.

The top-level domains are:

- **GOV** Government
- **EDU** Education
- **COM** Commercial
- **MIL** Military
- **ORG** Organization (an "other" category)
- **NET** Network administrative entities

Temporarily, the top-level domains also include:

- **ARPA** The current ARPA-Internet hosts

Additionally, the English two-letter codes identifying a country according to the International Standards Organization (ISO) Standard for *Codes for the Representation of Names of Countries* (ISO 3166, International Standards Organization, May 1981) can be used as top-level domains.

Sufficiently large companies can qualify for their own top-level domain. As of this writing, no company has attempted to qualify for a top-level domain.

11.1.2.3. How Domain Names Are Structured

The structure of a domain name is formally prescribed. Domains names are printed with each level of the domain name separated by a period. The Domain system knows nothing about hosts or sites; it deals only with names. The order of appearance in a domain name goes from the most specific to the most encompassing. For example, one of the Symbolics domain names is:

```
SCRC.Symbolics.COM
```

Based on the structure of the name, we can surmise that `SCRC` is the particular domain within Symbolics, and indeed it corresponds to a site in the Symbolics namespace. Continuing, we deduce that Symbolics is the name of the company that has a domain name of `Symbolics.COM`, and that `COM` is the top-level domain for commercial organizations. It is equally possible that `SCRC` is the name of a host. There is no way to tell *from the name* what `SCRC` is.

A host named "Rocky" in the Aerospace department at Whatsamatta University might have the domain name of:

```
Rocky.Aero.Whatsamatta.EDU
```

Looking from the most general to the most specific, this host is a part of the `EDU` domain. More specifically, it is a part of the domain `Whatsamatta.EDU`. Within the domain `Whatsamatta.EDU`, there is another domain—`Aero.Whatsamatta.EDU`. At this point, there is nothing to tell us if `Rocky` is a domain or a host in a domain. We know that `Rocky` is a host, because it is stated above. But you should always be aware of this potential ambiguity when reading domain names.

A domain name is just a name. The naming convention requires that some authoritative entity agree that it will be responsible for providing information about some domain and will guarantee that the information provided will follow the domain conventions. There is nothing implicitly better, worse, different, or otherwise unusual about the number of segments in a domain name.

As a consequence of the above convention, periods are effectively reserved characters. The domain `Whatsamatta.EDU` should not be referred to as `Whats.a.matta.EDU`. The latter is in an entirely different domain. A name must contain either a character, a numeral, a dash, an underscore, or some combination of these elements. Domain implementations are currently required to be case-insensitive.

11.1.2.4. How Genera Uses Internet Domain Names

This section describes how Genera implements Internet Domain Names, and how they are related to the Namespace system. For related information, see the section "The Domain System and the Namespace System", page 154.

How do Symbolics computers find network-related information?

In the Symbolics networking environment hosts must be able to obtain certain types of information about hosts and users of the network. The Namespace system stores that information in its database, and provides it to hosts that request information. A typical site has one designated namespace server.

Along with the namespace database, Symbolics computers support the Internet Domain Names style of requesting and obtaining network-related information.

Symbolics computers look for naming information as follows:

- They first seek it in the namespace database.
- If the information is not found, and the request involves an Internet Domain Name, they seek the information from hosts on the network called Name servers.

What kind of sites benefit from Internet Domain Names?

This facility is useful for:

- Sites with one or more hosts that use IP/TCP and are connected to the ARPA Internet, or any Internet that uses Domain Names.
- Sites that use Dialnet.
- Any site that uses the Internet Domain Names style of addressing.

When is the Internet Domain Names facility used?

The Internet Domain Names facility is integrated with the generic network system's procedure for finding a path to a host. When a network service is requested from a remote host, the generic network system must find a path to that host. For example, when you send an electronic mail message, the "To" field can contain an Internet Domain Names style of name, such as:

```
To: Customer-Reports@STONY-BROOK.SCRC.SYMBOLICS.COM
```

The generic network system must find the network address of the host named `STONY-BROOK.SCRC.SYMBOLICS.COM` in order to send the message.

How does Genera find a host's network address?

The part of Genera that does this is called the *name resolver*. Specifically, it is code that is part of **net:parse-host**, which is used often by the generic network system. The name resolver first consults the namespace database for this kind of information. If the information is not found, and the name is an Internet Domain Names style of name (with periods separating the components), the name resolver uses the Internet Domain Names facility. These steps are described below.

How does the name resolver search the namespace for an Internet Domain style name?

In this case, the name resolver looks for a namespace whose **internet-domain-name** attribute is `SCRC.SYMBOLICS.COM`, and then looks for a host named `STONY-BROOK` in that namespace. If no such namespace is found, or no host is found in such as namespace, the resolver begins to seek the information from Name servers. A name must contain at least one period to be a candidate for this kind of resolution.

How does the name resolver seek the Internet Domain Names information?

The name resolver determines whether it has the requested information stored in a local cache; this would happen if it had already processed a similar request. This step saves the resolver from making an unnecessary search for information. If the information is not found in the local cache, the resolver seeks the information from another host on the network. The resolver makes a request of the central name resolver, if any host at the site provides the **:domain** service. If not, it makes a request of one of several designated hosts on the network known as Name servers.

How does the name resolver know if the site has a central name resolver?

When a host is booted, or the Reset Network command is given, the host looks in the namespace database in the current site for any hosts that provide **:domain** service. If so, the resolver always makes requests of the central name resolver instead of making requests directly of the Name servers. If no host provides **:domain** service, the host makes direct requests of Name servers. The host consults the **root-domain-server-address** attribute of the site object to find out the addresses of the servers for the top-level ("root") domain.

What namespace objects does the name resolver create?

Because so much of the network software depends on objects being present in the namespace, the name resolver was implemented to create a host object for hosts that were not already stored in the namespace, but were located via some Name server. For the host named `STONY-BROOK.SCRC.SYMBOLICS.COM`, a namespace called `DOMAIN` is created, if not already present. A host object named `STONY-BROOK.SCRC.SYMBOLICS.COM` is created in the `DOMAIN` namespace, if it is not already present.

11.1.2.5. Symbolics Computers as Central Name Resolvers

Name servers are hosts that provide a service to all hosts on the Internet. A *central name resolver* is a host that provides a service to all hosts at a site; that service is described here.

Some sites gain advantages when they designate a single host to perform most of the name resolution for the entire site. Each host at the site contains the name resolver software, but in this configuration that code does not make requests to Name servers on the network. Instead, it makes a request of the central name resolver host. Note that you can configure your site to have multiple hosts designated as central name resolvers.

A central name resolver receives requests from hosts at the site, and processes them by requesting the desired information from Name servers. When information is returned, the central name resolver shares it with the user host, and also stores it in a local cache. Thus, if a second host at the site requests the same information, the central name resolver can return it quickly, without resorting to another network request.

To designate a host as a central name resolver, you should add the following service attribute to its host object:

```
Service: Set: DOMAIN CHAOS DOMAIN
```

If the resolver supports IP/TCP protocols, you should also add the following:

```
Service: Set: DOMAIN TCP DOMAIN
```

11.1.2.6. Symbolics Computers as Name Servers

The name resolver lets a Symbolics computer go out to the network to request information from Name servers. In addition, Symbolics computers can be Name servers themselves.

When a Symbolics computer is designated as a Name server, it has a responsibility to provide information to other hosts on the network regarding hosts, users, and other network objects within its domain. When it is booted, it loads a file that defines its domain and some other configuration data. Much of the information that the Name server needs resides in the namespace database. The Genera implementation takes advantage of that, and does not require that the Name server duplicate information already stored in the namespace. When the Name server needs information not present in the namespace, it can be stored elsewhere. The file SYS:SITE;LAUNCH-DOMAIN-SERVER.TEXT contains the pathnames of any additional data files.

A computer that is designated to be a Name server *for the ARPA Internet* must support IP/TCP, because it must be capable of communicating with other hosts on the Internet using IP/TCP.

Note that a Symbolics computer can be a Name server even if it is not connected to the ARPA Internet and does not support IP/TCP. For example, a site that supports only Chaosnet protocols could still use Internet Domain Names to name users and hosts. All that is required is that each host on the network is capable of requesting Name resolution and that the designated Name server is capable of storing and providing the information necessary to resolve Internet Domain Names.

It is not necessary that a Symbolics computer acting as a Name server have the **:domain** service attribute in its host object.

11.1.2.7. Internet Domain Name Namespace Attribute

During installation you specify 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. All hosts that are named within that namespace then inherit the Internet Domain Name that is entered in the namespace object.

For example, the SCRC namespace object might have this attribute:

```
Internet Domain Name:  SCRC.Symbolics.COM
```

SCRC|JUNCO is a host named Junco in the SCRC namespace. Junco inherits the Internet Domain Name of its namespace, so its Internet Domain Name is:

```
Junco.SCRC.Symbolics.COM
```

In some cases a host in that namespace is not in the same Internet domain. An individual host can override the Internet domain of its namespace by entering a value in the **Internet Domain Name** attribute of its host object. In this example the host SCRC|GRACKLE has the Internet Domain Name Grackle.MIT.EDU.

```
Internet Domain Name:  Grackle.MIT.EDU
```

The **Internet Domain Name** attribute of the host object is used solely to override the attribute of the namespace object.

11.1.2.8. The Domain System and the Namespace System

In many ways, the Domain system and the Namespace system attempt to solve the same problems. Both the Namespace system and Domain System attempt to deal with the issue of naming. Both systems deal with a collection of names that refer to a grouping of machines. In the case of the Namespace system, this collection is called the namespace. In the case of the Domain system, we shall refer to this as a domain or a subtree. However, it is important not to draw too close an analogy between the two.

It might appear that both systems map to "administrative entities". Actually, the Domain system returns attributes that are connected to names. The Namespace system goes beyond the Domain system in describing the hosts, users, printers, and networks within an entity known as a Site. There is nothing in the Domain system that is equivalent to a Site. Humans make the connection between a name and an administrative entity like a site; the Domain system software deals *only* with names.

The major difference between the Symbolics Namespace system and the Domain system is that the space containing the set of all Namespace names is flat, whereas the Domain system is organized as a hierarchy. From one perspective, this hierarchy can be viewed as a tree-structured administrative hierarchy.

Any site which has Symbolics computers must use the Namespace system. Communication with other Symbolics machines within a site can occur without use of the Domain system if the Symbolics machines are in the Namespace system. Any site which wants to communicate with other sites must use the Domain system. If there are non-Symbolics machines at your site and you want to communicate with them via IP/TCP, you should run the Domain system. If you are running Dialnet and Genera, you must use the Domain system.

The Domain system and the Namespace system appear to have information which overlaps. In point of fact, you cannot describe information via the Domain system that is also represented in the Namespace system. In other words, the Namespace System is *always* asked first, and it *always* wins any argument about the validity of any piece of data. If a query about a host that is mentioned in both the Namespace system and in the Domain system occurs, the information from the Namespace system will be used.

There is only one way of assuring that the information in the Namespace system and the Domain system don't conflict: by making a Symbolics computer the prima-

ry domain resolver for machines that are in the Symbolics namespace at your site. If this is done, the namespace information will be used to complete the domain information. If this is not done, data integrity will be compromised, since you must manually update all host information in both the Namespace system and the Domain system at the same time.

If you have machines that are not part of the Symbolics namespace, you should have a Symbolics machine serve as the piece of the domain tree that corresponds to the Symbolics namespace, and let any other machine deal with other parts of the domain tree. There is no useful way a machine can be a server for only part of a namespace/subtree. Note that nowhere in this discussion have we mentioned "site", only namespace.

You cannot have a partial representation of the hosts in the Namespace system and the remainder in a domain server elsewhere. Partial representation of information in one domain server and the rest in another domain server is also not allowed. Confusion occurs when there is *not* a single authority for a block of names, when one server has one piece of the namespace/subtree and some other server has the rest of the namespace/subtree. *This restriction is not a characteristic of the namespace implementation nor of any domain implementation. Rather, it is a fact common to naming schemes.* If partial information were allowed, it is easy to see that problems would arise as soon as one server's information differed with another's. There *must* be an authoritative server in any naming scheme.

If your organization already has a domain resolver running on another system, you have two options:

- Move the domain resolver to a Symbolics machine.
- Create a new sub-domain containing the Symbolics machines with a Symbolics machine as a domain resolver.

11.1.2.9. Summary of the Internet Domain Names Facility

Name resolver

This code is used to resolve network names, such as turning a host name into the correct network address for that host. The code is part of **net:parse-host**. If a name (such as a host name) contains periods, it is an Internet Domain Names style of name. In these cases, the name resolver checks to see if the namespace database contains the information. If not, the name resolver makes a network query for the needed information. The name resolver queries a central name resolver if any are designated at the site. If not, it queries one of the Name servers directly. The host can look at the **Root-Domain-Server-Address** attribute of the site object to find out the addresses of the top-level Name servers. If that attribute of the site object is empty, and no central name resolvers have been designated for the site, then the name resolver cannot resolve the requested name.

Central name resolver

This is a host that the site depends upon to perform name resolution for all Symbolics computers at the site. A central name resolver is designated by having one or two service attributes for **:domain** service in its host object.

```
Service: Set: DOMAIN CHAOS DOMAIN
Service: Set: DOMAIN TCP DOMAIN
```

To resolve a name, a host first checks the namespace database. If the name is not present in the namespace, the host submits a request to the central name resolver, using the **:domain** protocol. The central name resolver checks its local cache to see if it contains the requested information. If not, it makes a request to a designated Name server. The central name resolver decides which Name server to ask by looking at the **Root-Domain-Server-Address** attribute of the site object. If that attribute of the site object is empty, the central name resolver cannot perform the resolution.

Name server

This is any host that provides information on names and addresses to other hosts, using the DOMAIN protocol. Symbolics computers are capable of being Name servers. The server software is a separately loadable system. Note that a central name resolver serves the hosts within the site, but a Name server also serves hosts outside of the site. Sites can configure one Symbolics computer to be both a central name resolver and a Name server.

11.1.3. References to IP/TCP Protocol Specifications

All documents identified as ARPANET Requests for Comments (RFCs) are available from the ARPA Network Information Center:

ARPA Network Information Center
USC - Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90292
ARPANET: NIC@SRI-NIC

For those with ARPA Internet access, they are also available online as

```
NIC.DDN.MIL:<RFC>RFC###.TXT
```

where ### is the RFC number.

Internet References

Partridge, Craig, *Mail Routing and the Domain System*, RFC 974, January 1986.

Stahl, *Domain Administrators Guide*, RFC1032.

Lottor, *Domain Administrators Operations Guide*, RFC1033.

Mockapetris, P., *Domain Names - Concepts and Facilities*, RFC 1034.

Mockapetris, P., *Domain Names - Implementations and Specifications*, RFC 1035.

Reynolds, J. & Postel, J., *Domain Requirements*, RFC 920, October 1984.

Reynolds, J. & Postel, J., *Official Protocols*, RFC 880, October 1983.

Information Sciences Institute, *Internet Protocol*, RFC 791, September 1981.

Information Sciences Institute, *Internet Control Message Protocol*, RFC 792, September 1981.

Information Sciences Institute, *Transmission Control Protocol*, RFC 793, September 1981.

Postel, J., *User Datagram Protocol*, RFC 768, August 1980.

Postel, J., Reynolds, J., *TELNET Protocol Specification*, RFC 854, May 1983.

Postel, J., *File Transfer Protocol*, RFC 765, June 1980.

Sollins, K. R., *The TFTP Protocol*, RFC 783, June 1981.

Postel, J., *Simple Mail Transfer Protocol*, RFC 821, August 1982.

Harrenstein, K., *NAME/FINGER*, RFC 742, December 1977.

Postel, J., Harrenstein, K., *Time Protocol*, RFC 868, May 1983.

Crispin, M., *SUPDUP Display Protocol*, RFC 734, October 1977.

Harrenstein, K., White, V., Feinler, E., *Hostnames Server*, RFC 811, March 1982.

Reynolds, J., Postel, J., *Assigned Numbers*, RFC 870, October 1983.

11.2. DNA Networks

11.2.1. Introduction to DNA Networks

In Symbolics terminology, *DNA* is a type of network. On a DNA network, hosts communicate using standard DECnet protocols. See the section "References to DECnet Protocol Specifications", page 158.

If a site supports DNA:

- The site's namespace database has a network object of type DNA.
- One or more hosts have DNA addresses. DNA addresses are stored in the **address** attributes of the host objects. See the section "Format of DNA Addresses", page 29.
- Symbolics computers can communicate with other hosts on the DNA network using standard DNA protocols; the known protocols are stored in the **service** attributes of the host object.

The optional Digital Network Architecture (DNA) software package enables the Symbolics computer to access services provided by a VAX/VMS systems using the DNA protocols. These systems can be located either on the local Ethernet or on some other DNA network connected to the local Ethernet via a router node.

The primary goal of the Symbolics DNA software package is to enable a VAX/VMS machine to provide services (such as FILE, LOGIN, and MAIL services) to Symbolics computers using DECnet protocols. Symbolics computers support DNA user programs that communicate with DNA server programs on the VAX/VMS machine.

The supported protocols are listed elsewhere: See the section "DNA Protocols Supported by Symbolics Computers as Users", page 48. See the section "DNA Protocols Supported by Symbolics Computers as Servers", page 48.

Symbolics does not support the use of DNA protocols between two Symbolics computers.

The document *Symbolics DNA Software Package* describes the installation and site configuration procedure.

11.2.2. References to DECnet Protocol Specifications

These documents are available from Digital Equipment Corporation:

Software Documentation
1925 Andover Street TW/E07
Tewksbury, Massachusetts 01876

- *DECnet Digital Network Architecture (Phase IV) General Description*, Order No. AA-N149A-TC
- *DECnet Digital Network Architecture (Phase IV) Ethernet Node Product Architecture Specification*, Order No. AA-X440A-TK
- *DNA Session Control Functional Specification*, Version 1.0.0, Order No. AA-K182A-TK
- *DNA Data Access Protocol (DAP) Functional Specification*, Version 5.6.0, Order No. AA-K177A-TK
- *DNA Routing Layer Functional Specification*, Version 2.0.0, Order No. AA-X435A-TK
- *DNA Network Services Protocol (NSP) Functional Specification*, Version 4.0.0, Order No. AA-X439A-TK
- *Guide to Networking on VAX/VMS*, Order No. AA-Y512A-TE

11.3. 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 characters 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.

11.4. BYTE-STREAM-WITH-MARK Network Medium

11.4.1. Introduction to BYTE-STREAM-WITH-MARK Network Medium

A BYTE-STREAM-WITH-MARK implements a reliable, bidirectional byte stream with one out-of-band (but not out-of-sequence) signal called a *mark*. The design of BYTE-STREAM-WITH-MARK ensures that the mark is always recognizable on the receiving end.

The BYTE-STREAM-WITH-MARK is an *encapsulation* of an underlying stream, which must support the transmission of 8-bit bytes.

The Mark as a Synchronization Signal

Marks are used to resynchronize the stream when something has occurred to interrupt normal operations. For example, an application layer sending data over the BYTE-STREAM-WITH-MARK can abort in the middle of sending that data. Recovery is handled by sending a mark.

BYTE-STREAM-WITH-MARK and NFILE

BYTE-STREAM-WITH-MARK is the network medium used for NFILE. NFILE uses the marks implemented in BYTE-STREAM-WITH-MARK to resynchronize any *unsafe* control connections or data channels. For a description of NFILE's use of marks to resynchronize streams: See the section "NFILE Resynchronization Procedure", page 186.

BYTE-STREAM-WITH-MARK and Underlying Protocols

The BYTE-STREAM-WITH-MARK medium has been implemented to run on TCP and Chaos. Marks are implemented differently on the two protocols. However, the basic design of the BYTE-STREAM-WITH-MARK requires that a mark always be recognizable in the byte stream. Higher-level protocols ensure that transmissions are received intact.

Marks on Chaosnet

A mark is recognized on Chaosnet by a packet bearing the opcode 201 (octal). There are no data in a mark packet, so the data portion of the packet is ignored.

For other (non-Chaos) encapsulated streams that support opcode-bearing packets, the recommended implementation is the reservation of an opcode for the mark.

Marks on TCP: Record Mode

It is crucial for marks to always be unambiguously identified. Therefore, for TCP (and any transport media that do not implement packets natively) a simple record stream is imposed on the medium. The record boundaries serve only to distinguish where a mark can occur.

A record consists of a two-byte byte count, most significant byte first, followed by that many bytes of data. A byte count of zero is recognized as a mark.

Both the sending side and the receiving side must rigorously maintain the integrity of the record boundaries. A writer to the stream must never output a byte count without that number of data bytes following. Similarly, a reader of the stream, after reading a byte count, has effectively contracted to read that many bytes from the encapsulated stream, regardless of whether those bytes are requested by the application layer.

Maintaining Record Integrity

This subsection deals with maintaining record integrity on non-Chaos networks. Since Chaos implements packets natively, no special care is required to maintain record integrity on the Chaos network.

The design discussed here guarantees record integrity; the underlying stream must guarantee data integrity.

The basic design of BYTE-STREAM-WITH-MARK on TCP (and other transport protocols that do not implement packets natively) is to preserve record integrity by putting clearly demarcated, byte-counted records in the natural records of the encapsulated stream. Therefore, when the outer stream requests a buffer's worth of data from the encapsulated stream, it expects to receive a buffer containing one entire, integral, record of that stream, complete with byte count.

Because of diverse network implementations on different operating systems, the software that implements the encapsulated stream might not be able to provide integral record buffers to the BYTE-STREAM-WITH-MARK implementation. For example, the writing stream could have written records that are much longer than available buffers on the receiving system. In this case, a request to read from the encapsulated stream returns some buffer or some amount of data representing less than an entire BYTE-STREAM-WITH-MARK record. The input subroutine of the BYTE-STREAM-WITH-MARK implementation must therefore return a region of this (smaller) buffer, representing less than the full BYTE-STREAM-WITH-MARK record. Nevertheless, the BYTE-STREAM-WITH-MARK must extract the count of the full BYTE-STREAM-WITH-MARK record from the first such buffer of each

BYTE-STREAM-WITH-MARK record, and maintain and update this count as succeeding component buffers are read.

In this case, if the program reading from the BYTE-STREAM-WITH-MARK aborts while reading data, the implementation of BYTE-STREAM-WITH-MARK must continue to read through the remaining buffers of the BYTE-STREAM-WITH-MARK record that has been subdivided in this fashion.

The user side program will have determined that an abort has occurred, and will request the BYTE-STREAM-WITH-MARK to read up to and through the next mark. The BYTE-STREAM-WITH-MARK will have processed a fractional record, and must discard the remaining buffers of the record now being read.

11.4.2. BYTE-STREAM-WITH-MARK Abortable States

BYTE-STREAM-WITH-MARK is designed to provide end-to-end stream consistency in the face of user program aborts. This section describes user program aborts, and how BYTE-STREAM-WITH-MARK handles them.

Definition of User Program Aborts

Aborting the current execution of a program means to halt that execution and to abandon it, never to complete it. The data representing the state of the execution are irrevocably discarded.

User Program Aborts and I/O Streams

Aborting the execution of the code that manipulates I/O streams, in general, poses significant problems. Given that a stream is a static data object, and is intended to be used over and over again, aborting the execution of any routine manipulating a stream can leave it in an inconsistent, unusable state.

Many operating systems solve this problem by manipulating a large subset of streams within the confines of the supervisor or executive program, which is not vulnerable to aborts, short of system failure. Nevertheless, the need still exists to implement streams outside of the boundaries of the supervisor. Furthermore, the Genera environment has no supervisor or executive program, and is thus vulnerable to aborts everywhere.

BYTE-STREAM-WITH-MARK Handling of User Program Aborts

The BYTE-STREAM-WITH-MARK medium is designed to be nearly impervious to the aborting of programs using it. Its design is based on careful analysis of all possible states of the stream, and of the effect of aborts of the programs using the stream in each of these states. This section provides that analysis.

A *transmission* is a collection of user data sent by the application level through the BYTE-STREAM-WITH-MARK whose end is well-defined, once its start has been recognized. For instance, the token list stream, when using BYTE-STREAM-WITH-MARK, sends token lists. When a token list `TOP-LEVEL-LIST-BEGIN` has been sent, the containing transmission is not considered complete until the correspond-

ing TOP-LEVEL-LIST-END is read. See the section "Token List Transport Layer", page 165.

The following cases are possible states of the stream when an abort occurs:

1. Abort occurs when the user program is not manipulating the stream.

   This case presents no problem.

2. Abort occurs after a transmission has been partially sent, at a packet or record boundary.

   This implies that the datum that would indicate the successful complete sending of that transmission has been not yet been sent.

   The BYTE-STREAM-WITH-MARK state is consistent, but the application level state is not. The application level must determine that the execution of the code composing and sending its transmission was, in fact, aborted, and initiate resynchronization via marks.

   The receiving side must be careful not to act upon a transmission (that is, to perform any action or side effect) until the transmission has been successfully received in entirety. This protects the user program from the possibility that an abort can occur after a transmission has been partially sent.

3. Abort occurs during the sending or receiving of a record.

   This is the most vulnerable state of the mechanism. This case does not occur on packet media; it is subsumed by the next case.

   This case is handled by minimizing the extent of this window, and killing the connection when and if the situation is detected. Depending on the operating system involved, you might minimize this window by using interrupt-disabling mechanisms, auxiliary processes or tasks, or some other technique.

   For buffered streams, input and output waiting can be done in consistent states, thus minimizing the amount of time manipulating the actual encapsulated stream. For unbuffered streams, a lot of time can be spent in this window. It is expected that unbuffered streams will be exceedingly uncommon. Nevertheless, the implementation of BYTE-STREAM-WITH-MARK must detect this case.

4. Abort occurs during the sending or receiving of fundamental units of the lowest-level underlying stream (packets, buffers, or bytes).

   This case is usually handled by inhibiting interrupts, or other forms of masking, in the code implementing the encapsulated stream, since no waiting is possible at unexpected times.

11.4.3. Interfacing to the Lisp Machine Byte-Stream-With-Mark

This section describes the messages and underlying protocols of the Genera implementation of BYTE-STREAM-WITH-MARK, with two goals in mind.

This section enables you to:

- Construct applications built on the BYTE-STREAM-WITH-MARK medium.
- Utilize a lower-level medium (other TCP and Chaos, which are both already implemented) as a foundation for the BYTE-STREAM-WITH-MARK medium.

In either case, you accrue the benefits of the design and implementation of BYTE-STREAM-WITH-MARK, most notably the benefit that this medium is nearly impervious to user program aborts.

Any programmer designing an application using BYTE-STREAM-WITH-MARK should also consider using the token list stream, a powerful intermediate-level application that uses BYTE-STREAM-WITH-MARK. See the section "Token List Transport Layer", page 165.

A Genera BYTE-STREAM-WITH-MARK is a bidirectional, buffered, binary (8-bit bytes) stream, supporting all the usual stream messages (**:string-in**, **:string-out**, **:tyi**, **:read-input-buffer**, and so forth). See the section "Streams" in *Symbolics Common Lisp Programming Constructs*.

The raw stream is expected to also be a bidirectional, buffered, binary (8-bit bytes) stream, supporting the messages:

- **:read-input-buffer**
- **:advance-input-buffer**
- **:get-output-buffer**
- **:advance-output-buffer**
- **:force-output**
- **:finish**

The flavor **neti:buffered-stream-with-mark** can be instantiated to create such a stream. This flavor implements the record protocol, which implements marks as zero-length records. The implementation of BYTE-STREAM-WITH-MARK via TCP on the Symbolics machines uses this flavor. The encapsulated stream is accessible via the **:raw-stream** message. For further discussion of the record protocol used by BYTE-STREAM-WITH-MARK: See the section "Introduction to BYTE-STREAM-WITH-MARK Network Medium", page 159.

If a network medium can implement marks natively, as does Chaos on Symbolics Machines, you can directly support the functionality described here, without the record layer and the encapsulating stream, as long as the semantics of the BYTE-STREAM-WITH-MARK are preserved and both sides agree upon the data and mark representations.

:byte-stream-with-mark is a network medium that produces a BYTE-STREAM-WITH-MARK when a connection via it is established on Symbolics Machines. This medium supports the "connection argument" **:token-list**, whose value, when non-NIL, causes a token list stream to be created and returned, encapsulating the BYTE-STREAM-WITH-MARK. The **:stream** connection argument identifies the stream to be encapsulated. See the section "Token List Transport Layer", page 165.

The BYTE-STREAM-WITH-MARK passes the following messages on to its encapsulated stream, intact:

- **:close**
- **:foreign-host**
- **:accept**
- **:reject**
- **:connected-p**
- **:close-with-reason**
- **:complete-connection**
- **:set-output-exception**
- **:set-input-exception**
- **:check-output-exception**
- **:check-input-exception**

:start-open-auxiliary-stream passes through the request for the new stream to the encapsulated stream, but encapsulates it with a BYTE-STREAM-WITH-MARK after it has been created. If the parameter **:token-list** appears among the *stream options* with a non-**nil** value, a token list BYTE-STREAM-WITH-MARK is created.

In addition to the usual buffered stream messages, a BYTE-STREAM-WITH-MARK supports the **:send-mark** message. When this message is sent to the BYTE-STREAM-WITH-MARK, the latter forces all output it has buffered, that is, all byte stream records (in the non-packet case), sends a mark, and forces all this output into the encapsulated stream.

When, during an input operation from a BYTE-STREAM-WITH-MARK, the BYTE-STREAM-WITH-MARK encounters a mark, it signals the error condition of flavor **neti:mark-seen**.

The higher-level application must handle this error and interpret it in accordance with its usage of marks. The signalling routines expect to be aborted after this condition is signalled. The stream is then in a consistent state, and further input can be read.

neti:mark-seen *Flavor*

Signalled during an attempt to read from a BYTE-STREAM-WITH-MARK when a mark is encountered.

This typically occurs when a **:read-token-list** message is sent to a token list stream.

:stream can be sent to this condition to access the stream of interest.

11.5. Token List Transport Layer

11.5.1. Introduction to the Token List Transport Layer

The token list transport layer is a general-purpose protocol. The token list transport layer sends *tokens* through its underlying stream. Each token usually represents a simple quantity, such as a string or integer.

Tokens can be organized into *token lists*. Special tokens are provided to denote the starting and ending point of lists. The token list transport layer differentiates between *top-level token lists*, which are not contained in other lists, and *embedded token lists*, which are contained in other lists. Using lists makes it convenient to send structured records, such as commands and command responses.

The token list transport layer is a general term that includes two separate but related subjects: the *token list stream* and the *token list data stream*. The token list stream is commonly used for applications that can easily organize the information to be transmitted into tokens and lists. The token list data stream is more appropriate for transmitting a large volume of data that cannot easily be structured into tokens and lists, such as file data, which are just a sequence of characters or bytes.

The following table illustrates the main differences between token list streams and token list data streams:

	Token List Data Stream	*Token List Stream*
Built on:	Token list stream	BYTE-STREAM-WITH-MARK
Accepts these Messages:	Normal stream I/O messages, like **:tyi**, **:tyo**, **:string-in**	**:send-token-list** **:read-token-list**
Transmits:	Stream data	Tokens, token lists
Example of use:	NFILE data channels	NFILE control connection

NFILE and the Token List Transport Layer

NFILE uses the the token list transport layer, and provides an excellent example of its usefulness. The NFILE commands and command responses are sent over the control connection in a token list stream. File data are sent across each data channel in a token list data stream. For more information, see the section "NFILE File Protocol", page 177.

11.5.2. Token List Stream

11.5.2.1. Types of Tokens and Token Lists

All numbers in the token list documentation are represented in decimal notation. Bytes are 8 bits long.

Types of Tokens

Tokens are of the following types:

1. Atomic tokens.

   Atomic tokens are of the following subtypes:

   - Data tokens. A data token consists of a sequence of bytes with an effectively infinite maximum length. In some contexts a data token represents a string; in other contexts, a data token is other arbitrary data.

     Each data token is preceded in the token list stream by a representation of its length in bytes.

     Data tokens that are under 200 bytes long are preceded by one byte containing their length in bytes. That is, a data token of 34 bytes is preceded by one byte of value 34.

     Data tokens 200 bytes or over are preceded by the byte known as PUNCTUATION-LONG, of value 201. After the 201 comes a four-byte-long number (least significant byte first) containing the length of the data token that follows.

   - Numeric tokens. A sequence of bytes that represent and encode a nonnegative binary integer. The largest valid integer is 2^63 - 1.

     Numeric tokens are either short integers (less than 256) or long integers (greater than or equal to 256). Short integers are preceded by the byte known as PUNCTUATION-SHORT-INTEGER, of value 206.

     Long integers are begun by PUNCTUATION-LONG-INTEGER, of value 207. One byte follows, containing the length (in bytes) of the long integer. The integer itself is next, least significant byte first.

   - Keyword tokens. A sequence of bytes that represent and encode a named identifier of the implemented protocol. Keyword tokens are important only for their names.

     Each keyword is preceded by the byte known as PUNCTUATION-KEYWORD, of value 208. The data token following PUNCTUATION-KEYWORD represents the name of the keyword as a string. The characters are in upper-case.

- Boolean truth. A special token that represents the Boolean truth value. This token is known as BOOLEAN-TRUTH, of value 209.

2. Control tokens.

   The token list stream supports four control tokens to delimit token lists, and one padding token.

TOP-LEVEL-LIST-BEGIN	202	This control token appears at the start of every top-level token list.
TOP-LEVEL-LIST-END	203	This control token appears at the end of every top-level token list.
LIST-BEGIN	204	This control token appears at the start of every embedded token list.
LIST-END	205	This control token appears at the end of every embedded token list.
PUNCTUATION-PAD	200	This padding token should be ignored by the token list stream. It can be sent to fill buffers.

Token Lists

A token list consists of a sequence of atomic tokens or token lists. Token lists are begun and ended by control tokens that delimit the token lists. There are three types of token lists:

1. Top-level token lists.

   Top-level token lists begin with TOP-LEVEL-LIST-BEGIN and end with TOP-LEVEL-LIST-END. Top-level token lists are not contained in other lists.

2. Embedded token lists.

   These token lists occur inside other token lists. They begin with LIST-BEGIN and end with LIST-END.

3. The empty token list.

   This is a special example of the embedded token list. In some contexts, the empty token list represents Boolean falsity. An embedded empty token list is

composed of a LIST-BEGIN followed immediately by a LIST-END. A top-level empty token list is composed of TOP-LEVEL-LIST-BEGIN followed immediately by TOP-LEVEL-LIST-END.

11.5.2.2. Token List Stream Example

This section contains an example of some data that can appear on a token list stream. The example is a top-level token list encoding an NFILE DELETE command.

The DELETE command is composed of the following pieces: a TOP-LEVEL-LIST-BEGIN, the keyword DELETE, a data token containing the transaction identifier, a LIST-BEGIN, a LIST-END, a data token containing a pathname, and a TOP-LEVEL-LIST-END. Let's use T105 as the transaction identifier, and /usr/max/temp as the pathname.

All numbers in this section are expressed in decimal notation.

The pieces of the command are displayed here in order:

1. TOP-LEVEL-LIST-BEGIN
2. The keyword token whose name is DELETE
3. The data token containing the characters: T105
4. LIST-BEGIN
5. LIST-END
6. The data token containing the characters: /usr/max/temp
7. TOP-LEVEL-LIST-END

Now, let's translate each piece of the command into the bytes that are transmitted through the token list stream.

1. TOP-LEVEL-LIST-BEGIN

   202 represents TOP-LEVEL-LIST-BEGIN

2. The keyword token whose name is DELETE.

   A keyword token is begun by PUNCTUATION-KEYWORD, which is represented in the token list stream as the byte 208.

   A data token follows, containing the string "DELETE". A data token under 200 bytes long is begun by one byte containing its length in bytes. The length of this data token is 6 bytes.

   The data token continues with the Symbolics character set representation of each character in the string DELETE:

   208 represents PUNCTUATION-KEYWORD
   006 represents the length of this data token
   068 represents "D"

069 represents "E"
076 represents "L"
069 represents "E"
084 represents "T"
069 represents "E"

3. The data token containing the characters: T105

   This data token is begun by its length in bytes (4), and continues with the Symbolics character set representation of each character in the string:

   004 represents the length of this data token
   084 represents "T"
   049 represents "1"
   048 represents "0"
   053 represents "5"

4. LIST-BEGIN

   204 represents LIST-BEGIN

5. LIST-END

   205 represents LIST-END

6. The data token containing the characters: /usr/max/temp

   013 represents length of this data token
   047 represents "/"
   117 represents "u"
   115 represents "s"
   114 represents "r"
   047 represents "/"
   109 represents "m"
   097 represents "a"
   120 represents "x"
   047 represents "/"
   116 represents "t"
   101 represents "e"
   109 represents "m"
   112 represents "p"

7. TOP-LEVEL-LIST-END

   203 represents TOP-LEVEL-LIST-END

11.5.2.3. Mapping of Lisp Objects to Token List Stream Representation

The Genera interface to the token list stream sends Lisp objects through the underlying BYTE-STREAM-WITH-MARK and produces Lisp objects on the other end. Not all Lisp objects can be sent in this way: specifically, compound objects other than lists are not handled. An appropriate analogy is the sending and reconstruction of list structure via printed representation. These are the types of objects that can be sent, and their representations:

- Lisp strings are represented as data tokens in the Symbolics character set. Only 8-bit strings can be sent; no fat-strings can be sent.
- Keyword symbols are represented as keyword tokens. Although identifiable and reconstructable as keyword symbols, only their names are sent; their properties, bindings, and so on are not sent.
- **t** is represented as `BOOLEAN-TRUTH`.
- **nil** is represented as the empty token list.
- Lists are represented as token lists. The ambiguity between **nil** and the empty list presents no problems for Symbolics Machines, although this concession of the protocol to Symbolics Machines can present problems on other systems. Circular lists cannot be sent.
- Integers are represented as numeric tokens. Only nonnegative integers less than 2^63 can be sent.

11.5.2.4. Flavors and Messages Related to the Token List Stream

This section describes the flavors and messages of the Symbolics implementation of the token list transport layer.

Token list streams are created in two ways:

- If no underlying stream is present, token list streams are implemented as instances of the flavor **neti:token-list-stream**.
- If an underlying stream is present, it is not possible to compose a new flavor to support the token list functionality. For this purpose, the flavor **neti:buffered-token-stream** is provided.

In both cases, the token list stream is built on a BYTE-STREAM-WITH-MARK. For more information, see the section "BYTE-STREAM-WITH-MARK Network Medium", page 159.

The flavor **neti:mark-seen** is part of the BYTE-STREAM-WITH-MARK layer, but it might be of interest to users of token list streams. For more information, see the flavor **neti:mark-seen**, page 164.

neti:token-list-stream *Flavor*

Token list streams are implemented as instances of the **neti:token-list-stream** flavor.

This flavor expects to be mixed into the instantiation of its underlying BYTE-STREAM-WITH-MARK. It expects the stream into which it is mixed to implement the following messages:

- **:listen**
- **:tyi**
- **:string-in**
- **:send-mark**
- **:tyo**
- **:string-out**
- **:force-output**
- **:read-input-buffer**
- **:advance-input-buffer**
- **:get-output-buffer**
- **:advance-output-buffer**
- **:finish**

Often an existing underlying stream is present, and it is not possible to compose a new flavor to support the token list stream functionality. For this purpose, the flavor **neti:buffered-token-stream** is provided.

neti:buffered-token-stream *Flavor*

When an 8-bit binary bidirectional stream is present (usually not a BYTE-STREAM-WITH-MARK), the flavor **neti:buffered-token-stream** can be instantiated. The init keyword **:raw-stream** identifies the stream to be encapsulated. The result is a token list stream built on a BYTE-STREAM-WITH-MARK, using the record (non-packet) mode of BYTE-STREAM-WITH-MARK. The BYTE-STREAM-WITH-MARK is built on the stream supplied with the **:raw-stream** init keyword.

For a description of record mode of BYTE-STREAM-WITH-MARK,

see the section "Introduction to BYTE-STREAM-WITH-MARK Network Medium", page 159.

:send-token-list *object &optional mark-p* *Message*

object is a simple Lisp object to be sent through the token list stream. All token list streams support this message. The given object is sent in its entirety before any other data is allowed to be sent by the stream (perhaps from another process).

Not all Lisp objects can be sent through the token list. For more information, see the section "Mapping of Lisp Objects to Token List Stream Representation", page 170.

If *mark-p* is non-**nil**, a mark is sent on the underlying BYTE-STREAM-WITH-MARK before the supplied datum is sent. It is an error to send this message if the stream is unsafe (on the output side) unless *mark-p* is non-**nil**. If the stream is unsafe on the output side and *mark-p* is non-**nil**, the stream is declared to be safe again. If the execution of this message is aborted, the stream becomes unsafe on the output side.

Note that since the token list stream is built on its underlying stream (in the flavor sense), miscellaneous control messages need not be forwarded. Implementations should not ask the underlying stream to send a mark via **:send-mark**; use **:send-token-list** instead.

:read-token-list *&optional discard-until-mark dont-wait-but-return-this* *Message*

Reads from the token list stream, and returns the representation of one data object. All token list streams support this message.

If the beginning of a top-level list is encountered, the whole list is read, constructed according to the mapping of token list representations to Lisp objects and returned. For more information, see the section "Mapping of Lisp Objects to Token List Stream Representation", page 170.

If a mark is encountered instead of the data object being read, or at any point inside it, the **neti:mark-seen** condition is signalled, and the stream is marked unsafe on the input side. Note that this implies that a second mark must be forthcoming to resynchronize the stream. It is an error to issue this message to a stream that is unsafe on the input side, unless *discard-until-mark* is non-**nil**. If the execution of this message is aborted, the stream becomes unsafe on the input side.

discard-until-mark specifies that all data are to be discarded until a mark is read. At that point, the stream is to be declared safe again, on the input side. When the stream is unsafe on the input side, this is the only valid operation (other than closing the stream). Note that the only valid response to receiving an unexpected mark is to supply this argument. This implies that resynchronizations via marks must either be initiated by some other communication channel, (as in NFILE), or involve two marks, the first one of which is no more than an instruction to read for the second. Higher level protocols usually want to send some kind of meaningful identifier immediately following the mark.

dont-wait-but-return-this allows the caller to determine if input is present. If this argument is non-**nil** (it should be some object that cannot be transmitted via the token list medium, such as a specific list or a non-keyword symbol), it is returned as the return value of this message if and only if input is not available. While it might seem that this duplicates the functionality of **:listen**, the locking and other aspects of the potential multiprocess nature of applications of the token-list stream require this more sophisticated technique. (If **:listen** were used instead, there would be a race between processes that had determined that input was available, and the loser of the race would block.)

neti:token-io-unsafe *Flavor*

This condition is signalled when any I/O operation is attempted on a token list stream that is unsafe in the given direction. For example, an input operation was attempted on an unsafe input token list stream.

:stream can be sent to this condition to access the stream of interest.

:direction can be sent to this condition, to determine the direction (**:input** or **:output**) of the stream.

neti:token-stream-data-error *Flavor*

This condition is signalled during **:read-token-list**, if the data being read do not conform to the defined token list stream organization. For example, mismatched token list delimiters would signal **:neti-token-stream-data-error**. That is, a `TOP-LEVEL-LIST-END` was found that does not correspond to a `TOP-LEVEL-LIST-BEGIN`.

:stream can be sent to this condition to access the stream of interest.

This condition indicates a serious problem. The problem could be:

- A hardware problem.
- A bug in the implementation of the token list stream (on either side).
- A bug in any protocol or network underlying the token list stream.

11.5.2.5. Aborting and the Token List Stream

A token list stream accrues the benefits of the abort management policy of the BYTE-STREAM-WITH-MARK on which it is built. In order to fully realize this benefit, some simple rules must be obeyed by any implementation of the token list stream.

The term *transmission*, used often in the following paragraphs, means a complete top-level token list. The transmission starts with the control token `TOP-LEVEL-LIST-BEGIN` and ends with `TOP-LEVEL-LIST-END`. The top-level token list can contain embedded token lists.

The interface that writes to the token list stream must be capable of writing the representation of entire transmissions. When this interface is called, it must effectively *lock* the token list stream, excluding access by other processes until the entire transmission has been encoded and sent.

If the sending is aborted while the stream is locked, the stream enters an *unsafe* state. Trying to send data while the stream is unsafe signals an error. The application and the token list stream must send a mark to cause resynchronization, and allow the token list stream to be used again. When the reading side encounters this mark, it resynchronizes itself according to whatever higher-lever protocol is in use.

Similarly, the interface that reads from the token list stream must be capable of reading entire transmissions. When this interface is called, it must lock the stream, excluding access by other processes until the entire transmission has been read.

If the reading is aborted while the stream is locked, the stream enters an *unsafe* state. The only exit from this unsafe state is by means of receiving a mark. When the stream is unsafe, the only valid operation that can be performed upon it is "read and discard all tokens until a mark is encountered; read and discard that mark; declare the stream safe again".

Depending on the higher-level protocol, the receipt of a mark might cause the reading side to read for further marks. NFILE implements the resynchronization of token list streams, and serves as a useful example. For more information, see the section "NFILE Control Connection Resynchronization", page 186.

The implementation has implemented the two mark-handling primitives in this way:

1. Send token (or list) preceded by a mark. When the stream is in the unsafe state (on the output side), this is the only permitted output operation (other than closing).

2. Read through to a mark and read the token (or list) following the mark. When the stream is in the unsafe state (on the input side), this is the only permitted input operation (other than closing).

11.5.3. Token List Data Stream

The token list data stream is a facility to transmit stream data through a token list stream. The Symbolics implementation avoids consing the data tokens as strings on the receiving side.

Format of Data Transmitted

The token list data stream imposes the following protocol on the data transmitted:

- Data are sent in the format of loose data tokens, not contained in token lists.
- The keyword token EOF indicates that the end of data has been reached.
- Token lists can be transmitted through the token list data stream.
- No loose tokens other than data tokens or the keyword token EOF can be sent.

The token list data stream is most appropriate for sending file data. It is expected (but not required) that its typical mode of use is to send a large number of data tokens, with an occasional token list. The design intent was that token lists would be used by the application program to indicate exceptional situations.

Data tokens, the keyword token EOF, and token lists are defined in the token list stream documentation. For more information, see the section "Types of Tokens and Token Lists", page 166.

Normal Stream I/O Messages Are Accepted

There are no special messages to token list data streams; their whole purpose is to allow normal I/O stream messages to be used to transfer data through token list streams. A program can copy files or other massive data through a token list stream, using normal stream operations and tools such as **stream-copy-until-eof**. Data can be read out of the token list data input stream by normal stream operations without consing strings. The **:eof** message to an output token list data stream sends the keyword token EOF, which is in turn recognized by the receiving side as the end of file indicator.

The Underlying Token List Stream

The token list data stream encapsulates an existing token list stream. As with most encapsulating streams, **:force-output** and **:eof** implicitly force output through the encapsulated stream, as well. Control messages are not forwarded; for those purposes the program must deal directly with the underlying token list stream. The **:raw-stream** message to a token list data stream accesses the encapsulated stream.

NFILE's Use of the Token List Data Stream

The NFILE file protocol provides a good example of the use of token list data streams. NFILE sends file data through token list data streams; each NFILE data channel is a token list data stream. Errors such as disk errors during the reading of a file are conveyed as token lists through the token list data stream.

11.5.3.1. Flavors Related to the Token List Data Stream

neti:token-list-input-data-stream *Flavor*

Instantiating this flavor creates a token list input data stream. The underlying token list stream must be supplied via the **:raw-stream** init option.

Stream data are transmitted though the underlying token list stream by a token list output data stream, and can be read by normal stream operations. End of file is indicated when the keyword token EOF is encountered.

If you should want to use a token list input data stream after receiving the end-of-file indicator, **:clear-eof** must be sent to the token list input data stream.

This stream expects to encounter only loose data tokens, whose undifferentiated data content is treated as stream data, and the keyword token EOF, which is treated as an end-of-file indicator. If a list is encountered, a condition of flavor **neti:token-data-was-list** is signalled. For more information, see the flavor **neti:token-data-was-list**, page 176.

When a **:proceed** message is sent to the error object, the token list stream data can once again be read. This capability can be used to embed asynchronous signals in stream data. Any other kind of token is an error, and marks are not intercepted or dealt with by the token list data stream at all.

neti:token-list-output-data-stream *Flavor*

Instantiating this flavor creates a token list output data stream. The underlying token list stream must be supplied via the **:raw-stream** init option.

The token list output data stream accepts data via normal stream output operations. The data are sent as undifferentiated loose data tokens through the encapsulated stream. The tokens bear no correspondence to the order or type of output operations.

:force-output forces the data through the encapsulated stream as well as the outer stream that receives the message. **:eof** sends a **:force-output**, and sends and forces through the keyword token EOF.

Should the program wish to send a token list through the underlying stream in the midst of data, it must force the output of the token list data stream, and send the list through the encapsulated stream.

No special action must be taken to reuse a token list data output stream. The message **:clear-eof** is an input stream message only: do not send it to output streams, or to bidirectional streams to address output state.

neti:token-list-bidirectional-data-stream *Flavor*

Instantiating this flavor creates a bidirectional token list data stream. The underlying token list stream must be supplied via the **:raw-stream** init option.

It combines the behaviors of the token list data input and token list data output streams, encapsulating one bidirectional stream. It is important not to confuse input and output messages when using a bidirectional stream.

neti:token-data-was-list *Flavor*

This condition is signalled when a token list is encountered in a token list stream as it is being read by a token list data stream.

The **:list** message can be sent to this condition to access the value of the list that was read.

:stream can be sent to identify the erring stream.

The proceed type **:proceed** can be sent to proceed the condition, to continue use of the token list data stream after handling the just-read list.

11.6. NFILE File Protocol

11.6.1. Introduction to NFILE

NFILE is a file protocol that enables you to perform a large set of operations on files and directories on remote systems, including:

- Read and write entire files
- Read and write portions of files
- Delete files
- Rename files
- Create links
- List directories
- Create directories
- Expunge directories
- Obtain properties of files
- Change properties of files

NFILE can be used over any reliable byte-stream medium, such as TCP or CHAOS. It performs better than the older QFILE protocol in the following areas:

- NFILE is not restricted to the Chaos medium, as is QFILE. NFILE can be used over any reliable byte-stream medium, including Chaos and TCP.

- NFILE can transfer data faster than QFILE can. NFILE's performance running on TCP is better than either its performance or QFILE's running on the Chaos medium.

- NFILE has a robust scheme for handling aborts on the user side; QFILE is vulnerable to aborts.

- The NFILE server provides more complete information about errors than does the QFILE server.

- NFILE commands return useful values; in some cases, the analogous QFILE command does not return any value.

- NFILE offers 25 commands, in comparison to QFILE's 18.

At present, NFILE server programs are provided only for Symbolics machines. Therefore, the NFILE file protocol runs only between two Symbolics machines, unless you write an NFILE server program for another system.

As part of the generic network system, NFILE is invoked when the user needs to read or write a file on a remote host; NFILE then does its job invisibly. For example, when a user in the Zmacs editor uses the Find File command, the generic network system goes to work to find the file and bring it into the user's environment. In certain circumstances NFILE is be called upon to transmit the data in

the file residing on a remote Symbolics Machine to the user's Symbolics Machine, or from the Symbolics Machine to a remote Symbolics Machine.

If you wish to set up your site to use NFILE: See the section "Starting to Use NFILE", page 178.

If you intend to write NFILE server or user programs for another system: See the section "Reference Information on NFILE", page 178.

11.6.2. Starting to Use NFILE

The NFILE file protocol is used to communicate between two Symbolics machines when the namespace database at the site contains the information that NFILE is available on the server machine.

To set up your site to use NFILE, edit the namespace database. If you are unfamiliar with the namespace database: See the section "Setting Up and Maintaining the Namespace Database" in *Site Operations*.

Edit the host object for each Symbolics machine that will run the NFILE server. Any Symbolics machine used as a file server (that is, one machine providing file service to many other machines at the site) should be set up to run the NFILE server. It is not necessary to edit the host objects for Symbolics machines that will use NFILE protocols only to get FILE service from other machines.

The NFILE protocol provides the FILE service over the Chaos and TCP media. Therefore, since all Symbolics machines use the Chaos medium, all sites should add this entry to the host objects:

`Service:` ***Set:*** `FILE CHAOS NFILE` *Global-name*

Sites that use the TCP medium should add this entry to the host objects:

`Service:` ***Set:*** `FILE TCP NFILE` *Global-name*

Sites that use both Chaos and TCP should add both service entries.

Once the NFILE entry or entries are included in the namespace database, the NFILE protocol is invoked automatically.

If a site runs both NFILE and QFILE, the network usually chooses NFILE over QFILE.

11.6.3. Reference Information on NFILE

11.6.3.1. NFILE Concepts

NFILE is a layered file protocol. The NFILE commands and command responses constitute the top layer. These commands and responses are transmitted in *token lists*; the token list transport layer is the middle level of protocol. The token list transport layer is built upon the BYTE-STREAM-WITH-MARK network medium. Both the token list transport layer and the BYTE-STREAM-WITH-MARK network medium were originally designed for NFILE, but are general layers of protocol that are intended to be used for other applications as well.

See the section "Token List Transport Layer", page 165.
See the section "BYTE-STREAM-WITH-MARK Network Medium", page 159.

Throughout the NFILE documentation, the data types of arguments, return values, asynchronous error descriptions, and notifications are described as being:

- strings
- keywords
- keyword lists
- integers
- Boolean values
- dates
- time intervals
- date-or-never's

However, a string as such is not transmitted over the token list stream; the string must be expressed in token list representation. Each of the conceptual data types must be mapped into the appropriate token list representation.

See the section "Mapping Data Types into Token List Representation", page 183.

An NFILE session is a dialogue between two hosts. The host that initiates the NFILE session is known as the *user side*, and the other host is the *server side*. The user side sends all NFILE commands. The server receives each command, processes it, and responds to it, indicating the success or failure of the command.

The user side keeps track of commands sent and command responses received by using *transaction identifiers* to identify each command. The user side generates a unique transaction identifier for each command, and sends the transaction identifier to the server along with the command. Each NFILE server response includes the transaction identifier of the command with which the response is associated. The server need not respond to commands in the same order that the user gave them.

See the section "NFILE Command Descriptions", page 189.
See the section "NFILE Commands", page 193.

The user side sends NFILE commands over a bidirectional network connection called the *control connection*. The server sends its command responses on the same control connection. All communication over the control connection is in the format of token lists. The control connection governing the NFILE session is established at the beginning of the session. If the control connection is ever broken, the NFILE session is ended.

Whereas NFILE commands and responses are transmitted on the control connection, file data are transferred over *data channels*. An *input data channel* is used to send data from server to user; an *output data channel* is used to send data from user to server. Each input data channel is associated with an output data channel; together these two channels constitute a data connection. Most communication over data channels is in the format of loose data tokens. In some cases, token lists are transmitted over the data channels.

See the section "NFILE Control and Data Connections", page 183.

In the case of a user program abort, control connections and data channels can be marked *unsafe*. Any unsafe control connection or data channel must be made *safe* again before further use, by undergoing a *resynchronization* procedure.

See the section "NFILE Resynchronization Procedure", page 186.

11.6.3.2. NFILE File Transfer Philosophy

This section describes how files are transferred from one system to another, using the NFILE file protocol. NFILE supports two ways of transferring file data, *data stream mode* and *direct access mode*.

Data Stream Mode

Data stream mode of file transfer is the default mode of NFILE's OPEN command. Data stream mode is appropriate when the entire file is transferred, either from user to server, or from server to user. Data stream mode is more common than direct access mode.

When a data stream opening is requested with the OPEN command, a stream is opened and the data begin to flow immediately. The OPEN command requires a *handle* argument to be supplied, which specifies the data channel to be used to transfer the data. The handle is used in future commands to reference the open stream.

The sending side transmits the entire contents of the specified file over the specified data channel as fast as the network medium and path allow. When the sending side reaches the end of the file, it transmits a special control token to signal end of file. The receiving side expects an uninterrupted stream of bytes to appear immediately on its side of the data channel.

The user gives the CLOSE command to terminate a data stream transfer. CLOSE results in closing the open stream.

Direct Access Mode

Direct access mode enables reading and writing data from specific starting points in a file, through a specified number of bytes. In direct access mode, data are requested and sent in individual transactions. To request a direct access mode opening, the OPEN command is used with a DIRECT-FILE-ID argument. (In data stream mode, no DIRECT-FILE-ID is supplied.) The direct file identifier is used in later commands to reference the direct access stream.

When a file is opened in direct access mode, the flow of data does not start immediately. Rather, the user gives either a READ command (to request data to flow from server to user) or a DIRECT-OUTPUT command (to request data to flow from user to server). In either case, the user specifies the starting point and the number of bytes of data to transfer. The user can give many READ and DIRECT-OUTPUT commands, one after another.

The user side terminates the direct access transfer by using the CLOSE command. The ABORT command prematurely terminates a direct access transfer.

Direct access file streams are supported by LMFS. For further information on how LMFS supports direct access file streams: See the section "Direct Access File Streams" in *Symbolics Common Lisp Programming Constructs*.

11.6.3.3. NFILE Character Set Translation

NFILE was designed to provide access between two Symbolics computers, and to provide access from Symbolics computers to ASCII-based file systems. Symbolics computers support 8-bit characters and have 256 characters in their character set. This causes difficulties when communicating with ASCII machines which have 7-bit characters.

NFILE file transfers are always done using the 8-bit Symbolics computer character set.

In this section, all numbers designating values of character codes are to be interpreted in octal.

Servers on machines not using the Symbolics computer character set are required to perform character set translations for any character opening. Two Symbolics Computers communicating with NFILE need not perform any character set translation.

Table 1 shows the translations between Symbolics computer 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 NFILE Command", page 205.

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:
In SUPER-IMAGE mode as well, stray CR is translated to

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

Symbolics character M.

11.6.3.4. Mapping Data Types into Token List Representation

The following list shows how each conceptual data type is expressed in token list representation. This mapping is also illustrated by an extended example of translating an NFILE command and its arguments into its token list representation: See the section "Token List Stream Example", page 168.

Keyword — Transmitted as a keyword token.

Keyword list — Transmitted as a token list of keyword tokens.

Integer — Transmitted as a numeric data token.

String — Transmitted as a data token containing the characters of the string in the Symbolics character set.

Boolean Truth — Transmitted as the token known as `BOOLEAN-TRUTH`.

Boolean False — Transmitted as the empty token list.

Dates — Transmitted as numeric data tokens. The date is expressed in Universal Time format, which measures a time as the number of seconds since January 1, 1900, at midnight GMT.

Date-or-never — Can be either a date or the empty token list, representing "never". "Never" is used for values such as the time a directory was last expunged, if it has never been expunged.

Time interval — Transmitted as a numeric data token. The time interval is expressed in seconds. A time interval of zero seconds (including the concept of "never") is represented by the empty token list.

11.6.3.5. NFILE Control and Data Connections

The user and server communicate through a single *control connection* and zero or more *data connections*. The user side sends NFILE commands to the server over the control connection. The server responds to every user command, also over this control connection. The actual file data are transmitted over the data connections.

User aborts can disturb the normal flow of data on the control connection and data connections. An important aspect of any file protocol is the way it handles user aborts. NFILE supports a resynchronization procedure to bring the affected control connection or data channel from an unknown, unsafe state into a known state, enabling the control connection or data channel to be reused. See the section "NFILE Resynchronization Procedure", page 186.

The Control Connection

The control connection is established at the beginning of the NFILE session. See the section "Establishing an NFILE Control Connection", page 185. The control connection is the vehicle used by the user to send its commands, and the server to send its command responses.

These types of communication occur over the NFILE control connection:

- The user side sends NFILE commands.
- The server sends command responses.
- The server sends notifications.
- The server sends asynchronous errors.
- During resynchronization (a special circumstance) either the user or server sends a mark.

For further information on each type of communication:

See the section "NFILE Command Descriptions", page 189.
See the section "Notifications from the NFILE Server", page 185.
See the section "NFILE Error Handling", page 226.
See the section "NFILE Resynchronization Procedure", page 186.

Format of Control Connection Communication

All commands, command responses, and other data flowing over the NFILE control connection are transmitted in the format of *top-level token lists*. The control connection expects never to receive *loose tokens*; that is, tokens not contained in token lists. For a definition of token lists:

See the section "Token List Transport Layer", page 165.

Data Connections

Data connections are established and discarded at user request, by means of two NFILE commands: DATA-CONNECTION and UNDATA-CONNECTION. Each data connection is associated with a specific control connection, which is the same control connection that caused the data connection to be established.

See the section "DATA-CONNECTION NFILE Command", page 199.
See the section "UNDATA-CONNECTION NFILE Command", page 225.

Each data connection is composed of two *data channels*. Each data channel is capable of sending data in one direction. The term *input channel* refers to the data channel that sends data from the server to the user side; *output channel* refers to the data channel that sends data from the user to the server side. Throughout the NFILE documentation, the terms input and output channels are seen from the perspective of the user side.

Data channels can be used for many data transfers, in sequence.

Format of Data Channel Communication

The data being transferred on the data channels are typically loose tokens, that is, tokens not contained in a token list. When the end of data is reached, the keyword token EOF is sent. Occasionally, token lists are transmitted over the data channels. For example, notifications and asynchronous error descriptions are token lists that are transmitted on data channels. The format of the data transferred on the data channels is defined as a *token list data stream*:

See the section "Token List Data Stream", page 174.

11.6.3.6. Establishing an NFILE Control Connection

NFILE is built upon the BYTE-STREAM-WITH-MARK medium, which is implemented to use either the Chaos or TCP protocol. This section gives the necessary information on how to establish a control connection on Chaos and TCP.

The NFILE user program connects to a remote host and establishes a network connection. This is the control connection of the dialogue that has just begun.

NFILE's Chaos Contact Name

The contact name referring to NFILE on Chaos is: NFILE.

Other sections describe the significance and use of the contact name in establishing a Chaos connection:

See the section "Chaosnet Contact Names", page 234.
See the section "Chaosnet Connection Establishment", page 245.

NFILE's Well-known TCP Port

The well-known port for NFILE on TCP is 59.

Symbolics does not document the TCP protocol, since documentation on TCP and the other Internet protocols is readily available elsewhere.

11.6.3.7. Notifications from the NFILE Server

The NFILE server can send asynchronous notifications to the user side over the control connection. The text of the notification contains information of interest to the person using NFILE, such as a warning that the server's operating system will be going down soon. Notifications can come from the server side at any time that the server is not sending something else.

The format of NFILE notifications is:

```
(NOTIFICATION "" text)
```

The empty string "" takes the place of a transaction identifier. Notifications are initiated by the server, and are not associated with any transaction originated by the user side.

Servers should not allow aborting during the sending of notifications. A server abort could cause the control connection to become unsafe on the server side.

11.6.3.8. NFILE Resynchronization Procedure

Ordinarily, the user side sends NFILE commands to the server side over the control connection; the server side responds to every user command, and file data is transmitted over the data channels. This section describes a resynchronization procedure that takes place when something disturbs the usual course of events.

First, if the server side aborts while sending or receiving data, nothing can be done to salvage the connection between the two hosts. The control connection and any data channels associated with this connection are broken. This happens rarely, if at all.

It is not unusual for the user side to abort file operations, either commands or data transfer. On a Symbolics machine, the user could do this by pressing c-ABORT. An important aspect of any file protocol is the way it handles the situation when the user side aborts file operations.

NFILE reacts to user side aborts by immediately marking the connection *unsafe*. When a control connection is unsafe, it must be resynchronized before it can be used again. Data channels can also be marked unsafe, and must also be resynchronized before further use. The resynchronization process rids the connection (whether control or data connection) of data that are now unwanted, and thus cleans up the channel so it can be used again.

NFILE Control Connection Resynchronization

NFILE requires any *unsafe* control connection to undergo a resynchronization procedure before further use. Therefore, the resynchronization does not necessarily occur immediately after the control connection is marked unsafe. NFILE control connections are marked unsafe by the user side upon aborting, for example, when a person using NFILE on a Symbolics machine presses c-ABORT. The user side initiates the control connection resynchronization when another operation on the control connection is attempted.

User Side Steps: Control Connection Resynchronization

1. The user side sends a *mark* over the control connection to the server.

2. The user side sends the ASCII characters USER-RESYNC-DUMMY (as a data token) to the server.

3. The user side sends a second mark to the server.

4. The user side declares the control connection safe (at the token list level).

5. The user side generates and sends a unique data token to the server.

6. The user side then waits, expecting to detect a mark followed by the unique data token. The user side reads and discards all tokens and marks until the desired match is found.

Once the user side detects the mark and unique data token, the control connection has been fully resynchronized, and can be used again.

Server Side Steps: Control Connection Resynchronization

1. The server side detects a mark instead of the token list normally received from the user side. The server is thus alerted that the control connection is unsafe, and that resynchronization is in progress.

2. The server continues to read data coming from the user side until it detects the second mark, and the token following it.

3. The server checks to see if the token following the mark is USER-RESYNC-DUMMY. This rare situation occurs if the user aborts during the course of the resynchronization itself. If so, the server side discards the USER-RESYNC-DUMMY token. The control connection is still unsafe, and the user side restarts the resynchronization procedure; the server side therefore begins at Step 2 again.

4. If the token following the mark is not USER-RESYNC-DUMMY (this is the expected circumstance), the server should have received a single data token that is the unique data token generated by the user side.

5. a. The server sends a mark to the user side.

   b. The server declares control connection safe (at the token list level).

   c. The server sends the unique data token to the user side.

6. If the server detects something following the mark that was neither USER-RESYNC-DUMMY nor a single data token, a protocol error has occurred.

NFILE Data Connection Resynchronization

The NFILE data channel resynchronization procedure is similar to the NFILE control connection resynchronization. Both procedures are based on a mark signalling the unsafe condition, then a second mark followed by a unique identifier. One important difference between the two procedures is the circumstances in which they occur. Control connections are put into unsafe states only when the user aborts during control connection I/O operations. Data channels are made unsafe by a larger set of circumstances:

- User aborts occur during the file protocol operations that assign and deassign data channels. This is the most common cause of data channels becoming unsafe.

- A server receives a CLOSE command (with *abort-p* supplied as Boolean truth) specifying an open file that has not finished transmitting data. That is, file reading is aborted.

- The ABORT command is issued, causing data channels to be made unsafe.

- The FILEPOS command is issued, causing the input data channel to become unsafe.

The resynchronization clears the data channel of unwanted data from aborted operations and puts the data channel in a known state. The data channel resynchronization procedure is invoked when the user side gives the RESYNCHRONIZE-DATA-CHANNEL command over the control connection.

In the Symbolics machine implementation, the user side initiates resynchronization only if it needs the data channel, having first tried to use a free data channel that does not require resynchronization. Also, the user side periodically resynchronizes all unsafe data channels.

In giving the RESYNCHRONIZE-DATA-CHANNEL command, the user side indicates which data channel should be resynchronized. Data channels are unidirectional, which means that depending on the direction (either input or output) of the data channel, either the user side or the server side sends the resynchronization data. This is another difference from the resynchronization of the control connection, in which the resynchronization data is always sent by the user side. The resynchronization steps for input data channels are different than the steps for output data channels.

Input Data Channel Resynchronization

1. The user side gives the RESYNCHRONIZE-DATA-CHANNEL command on the control connection, with only one argument, the handle of the data channel to be resynchronized.

2. The server side of the data channel generates a unique identifier, and sends that data token in its regular command response to the user side.

3. The server side sends a mark over the data channel.

4. The server side sends the unique identifier token over the data channel.

5. The user side reads until it detects a mark followed by the unique identifier token. The resynchronization is then complete. The data channel is no longer in an unsafe state.

Output Data Channel Resynchronization

1. The user side gives the RESYNCHRONIZE-DATA-CHANNEL command on the control connection, with two arguments: the handle of the data channel to be resynchronized, and a unique identifier that it has just generated.

2. The user side of the data channel sends a mark.

3. The user side of the data channel sends a dummy identifier token. The dummy identifier can be any token that the server could not interpret as being the unique identifier. One suggestion is the data token DUMMY-IDENTIFIER.

4. The server side of the data channel was alerted by the RESYNCHRONIZE-DATA-CHANNEL command that resynchronization is in progress. The server side now reads the data, seeking the first mark.

5. The server side reads and discards the first mark and the dummy identifier.

6. The user side sends a second mark.

7. The user side sends the unique identifier.

8. The server side recognizes the mark and the unique identifier that follows, and the resynchronization is complete. The data channel is no longer in the unsafe state.

11.6.3.9. NFILE Command Descriptions

Conventions Used in NFILE Command Descriptions

This section defines the conventions used in the NFILE command descriptions and explains some syntax rules that apply to NFILE commands and responses. A complete understanding of this section is necessary before you begin to write an NFILE server.

The conceptual data types mentioned in the command descriptions must be mapped into token list representation to be transmitted in the token list stream. Arguments and return values are defined as being a "string", "integer", "keyword", "keyword list", "Boolean truth", and so on. To determine the mapping of these conceptual data types into token list representation: See the section "Mapping Data Types into Token List Representation", page 183.

Command and Response Format

Each of the command descriptions begins by giving the command format and response format. Here is the beginning of the DATA-CONNECTION command description:

Command Format:

(DATA-CONNECTION *tid new-input-handle new-output-handle*)

Response Format:

(DATA-CONNECTION *tid connection-identifier*)

The command descriptions follow these conventions:

1. NFILE commands and responses are transmitted as top-level token lists.

   Top-level token lists are enclosed in parentheses in these command descriptions. These parentheses are not sent literally across the control or data connections, but are a shorthand representation of special control tokens that delimit top-level token lists. Specifically, TOP-LEVEL-LIST-BEGIN starts a top-level token list; TOP-LEVEL-LIST-END ends a top-level token list.

2. NFILE command names are keywords.

   The command name is required in every command and command response. All NFILE command names are *keywords*. Keywords appear in the NFILE documentation as their names in uppercase. For example, DATA-CONNECTION and DELETE are NFILE command names.

3. A unique transaction identifier (*tid*) identifies each command.

   The transaction identifier is a string made up by the user side to identify this particular *transaction*, which is composed of the command and the response associated with this command. The transaction identifier is abbreviated in the command descriptions as *tid*. Transaction identifiers are limited to fifteen characters in length. The transaction identifier is required in every command and command response.

4. Italics are used for placeholder arguments.

   The transaction identifier, command arguments, and command return values are italicized to indicate that they are placeholders for real values.

Optional Arguments

Many NFILE commands have *optional arguments*. Optional arguments can be supplied (with appropriate values), or left out. If optional arguments are left out, their omission must be made explicit by means of substituting the empty token list in their place. Any optional arguments or return values that are trailing can be omitted without including the empty token list.

For example, the text of the DELETE command description explains that either a *handle* or a *pathname* must be supplied, but not both; therefore, one of them is an optional argument. Here is the command format of DELETE:

(DELETE *tid handle pathname*)

If you supply a *handle* and no *pathname*, the command format is:

(DELETE *tid handle*)

If you supply a *pathname* and no *handle*, the command format is:

(DELETE *tid empty-token-list pathname*).

The empty token list in the token list stream appears as a `LIST-BEGIN` followed immediately by a `LIST-END`.

Optional Keyword/Value Pairs

Four NFILE commands have *optional keyword/value pairs*. These commands are: COMPLETE, LOGIN, OPEN, and READ. Optional keyword/value pairs can be either included in the command or omitted entirely. There is no need to explicitly omit optional keyword tokens, unlike optional arguments. The order of the option keyword/value pairs is not significant.

If included, optional keyword/value pairs are composed of the keyword itself, followed by its associated value. The values associated with the keywords can be keywords, lists, strings, Booleans, integers, dates, date-or-never's, and time intervals. The text of each command description states what type of value is appropriate for each optional keyword.

Optional keyword/value pairs appear in the text as the keyword only, in italicized uppercase letters. For example, here is the format of the LOGIN command:

Command Format:

(LOGIN *tid user password FILE-SYSTEM USER-VERSION*)

FILE-SYSTEM and *USER-VERSION* are two optional keywords associated with the LOGIN command. The user side can supply *USER-VERSION*, and omit *FILE-SYSTEM* as shown in this example:

(LOGIN T105 tjones abc123 USER-VERSION 2)

As seen above, the optional keyword/value pair *USER-VERSION*, if supplied in a command, is replaced by the keyword USER-VERSION, followed by the value to be used for that keyword (in this example, 2).

Data Channel Handles and Direct File Identifiers

Several NFILE commands require an argument that specifies an open stream. This kind of argument is called a *handle* in the command description. It is always a string type argument. A handle can be either a *data channel handle* or a *direct file identifier*, depending on the mode of the opening:

Data Stream	The handle must identify a data channel that is bound to an open stream.
Direct Access	The general rule is that the handle must be a direct file identifier. A direct file identifier specifies a direct access stream. It is the same as the value supplied in the *DIRECT-FILE-ID* keyword/value pair in the OPEN command. It is used for all operations that identify an open server stream rather than a data channel. Two NFILE commands applicable to direct access openings are exceptions to the general rule. The handle supplied in ABORT and CONTINUE cannot be a direct file identifier, but must be a data channel handle instead.

Full Pathname Syntax of the Server Host

Some arguments and return values in the NFILE command descriptions are *strings in the full pathname syntax of the server host*. These pathnames contain no host identifiers of any kind. These pathnames are fully defaulted, in the sense that they have a directory and file name (and file type, if the server operating system supports file types). If appropriate, a device is referenced in the pathname. If the server file system supports version numbers, there is always an explicit version number, even if that number or other specification is that system's representation of "newest" or "oldest".

Format of NFILE File Property/Value Pairs

Several NFILE commands request information regarding the properties of files or directories. These commands include: DIRECTORY, MULTIPLE-FILE-PLISTS, PROPERTIES, and CHANGE-PROPERTIES. This section describes how file property information is conveyed over the token list stream.

File property information is usually sent in *property/value pairs*, where the *property* identifies the property, and the following *value* gives the value of that property for the specified file. For a list of keywords related to file properties, and the type of value associated with each keyword: See the section "Recognized Keywords Denoting File Properties", page 193.

Each *property* is denoted either by a keyword or an integer. You can mix both ways of specifying properties (keyword or integer) within a single description. An integer is interpreted as an index into the *Property Index Table*, an array of property keywords. The server can optionally send a Property Index Table to the user during the execution of the LOGIN command, although it is not required.

In command arguments, file properties cannot be specified with integers; keywords must be used to specify file properties in command arguments. Integers can be used to denote file properties only in command responses.

Property *values* can be any of the following: keywords, keyword lists, integers, strings, Boolean values, dates, date-or-never's, or time intervals. For information on how each type of value is mapped into token list representation: See the section "Mapping Data Types into Token List Representation", page 183.

Recognized Keywords Denoting File Properties

This section lists the keywords associated with file properties. This list is not intended to be restrictive. If a programmer implementing NFILE needs a new keyword, a new keyword (not on this list) can be invented. The type of value of any new keywords is by default string.

The keywords are sorted here by type. For further information on the meaning of each keyword: See the function **fs:directory-list** in *Program Development Utilities*.

Integers	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, SUPERSEDE-PROTECT, NOT-BACKED-UP, OFFLINE, TEMPORARY, CHARACTERS, DIRECTORY
Strings	ACCOUNT, AUTHOR, LINK-TO, PHYSICAL-VOLUME, PROTECTION, VOLUME-NAME, PACK-NUMBER, READER, DISK-SPACE-DESCRIPTION, and any keywords not on this list

11.6.3.10. NFILE Commands

It is important to understand the conventions used in each of the following command descriptions. See the section "NFILE Command Descriptions", page 189.

ABORT NFILE Command

Command Format:

(ABORT *tid input-handle*)

Response Format:

(ABORT *tid*)

ABORT cleanly interrupts and prematurely terminates a single direct access mode data transfer initiated with READ. The required *input-handle* string argument identifies a data channel on which an input transfer is currently taking place; this must be a direct access transfer. *input-handle* must identify a data channel; it cannot be a direct file identifier.

Upon receiving the ABORT command, the server checks to see if a transfer is still active on that channel. If so, the server terminates the transfer by telling the data connection logical process to stop transferring bytes of data. The user side need issue this command only when there are outstanding unread bytes. This excludes the case of the data channel having been deestablished or reallocated by the user side.

Whether or not a transfer is active on that channel, the user side puts the data channel into the unsafe state. Before the data channel can be used again, it must be resynchronized.

CHANGE-PROPERTIES NFILE Command

Command Format:

(CHANGE-PROPERTIES *tid handle pathname property-pairs*)

Response Format:

(CHANGE-PROPERTIES *tid*)

CHANGE-PROPERTIES changes one or more properties of a file. Either a *handle* or a *pathname* must be given, but not both. Whichever one is given must be supplied as a string. *handle* identifies a data channel that is bound to an open file. *pathname* identifies a file on the server machine.

property-pairs is a required token list of keyword/value pairs, where the name of the property to be changed is the keyword, and the desired new property value is the value.

The properties that can be changed are host-dependent, as are any restrictions on the values of those properties. The properties that can be changed are the same as those returned as *settable-properties*, in the command response for the PROPERTIES command. See the section "PROPERTIES NFILE Command", page 221.

The server tries to modify all the properties listed in *property-pairs* to the desired new values. There is currently no definition about what should be done if the server can successfully change some properties but not others.

For further information on file property keywords and associated values:

See the section "Format of NFILE File Property/Value Pairs", page 192.

See the section "Recognized Keywords Denoting File Properties", page 193.

CLOSE NFILE Command

Command Format:

(CLOSE *tid handle abort-p*)

Response Format:

(CLOSE *tid truename binary-p other-properties*)

CLOSE terminates a data transfer, and frees a data channel. The *handle* must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. If a data channel is given, a transfer must be active on that *handle*. If *abort-p* is supplied as Boolean truth, the file is *close-aborted*, as described below.

"Closing the file" has different implications specific to each operating system. 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.

The server sends the keyword token EOF on the data channel, to indicate that the end of data has been reached.

In either data stream or direct access mode, 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 might 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.

An Unsuccessful CLOSE Operation

For the normal CLOSE operation (not a close-abort), after writing every last bit sent by the user to disk, and before closing the file, the server checks the data channel specified by *handle* to see if an asynchronous error description is outstanding on that channel. That is, the server must determine whether it has sent an asynchronous error description to the user, to which the user has not yet responded with a CONTINUE command. If so, the server is unable to close the file, and therefore sends a command error response indicating that an error is pending on the channel. The appropriate three-letter error code is EPC. See the section "NFILE Error Handling", page 226.

A Successful CLOSE Operation

The return values for OPEN and CLOSE are syntactically identical, but the values might have changed somewhat between the file being opened and closed. For example, the *truename* return value is supplied after all the close-time renaming of output files is done and the version numbers resolved (for operating systems supporting version numbers). Therefore, on some systems the *truename* when the file was opened is different than the *truename* after it has been closed.

For a description of the CLOSE return values: See the section "NFILE OPEN Response Return Values", page 219.

If the user gives the CLOSE command with *abort-p* supplied as Boolean truth, thus requesting a close-abort of the file, the server need not check whether an asynchronous error description is outstanding on the channel. The server simply close-aborts the file.

COMPLETE NFILE Command

Command Format:

(COMPLETE *tid string pathname DIRECTION NEW-OK DELETED*)

Response Format:

(COMPLETE *tid new-string success*)

COMPLETE performs file pathname completion.

string is a partial filename typed by the user and *pathname* is the default name against which it is being typed. Both *string* and *pathname* are required arguments, and are of type string.

The other arguments are optional keyword/value pairs. *NEW-OK* is Boolean; if followed by Boolean truth, the server should allow either a file that already exists, or a file that does not yet exist. The default of *NEW-OK* is false; that is, the server does not consider files that do not already exist.

DELETED is a Boolean type argument; if followed by Boolean truth, the server is instructed to look for files that have been deleted but not yet expunged, as well as non-deleted files. The default is to ignore soft-deleted files.

DIRECTION can be followed by READ, to indicate that the file is to be read. If the file is to be written, *DIRECTION* can be followed by WRITE. The default is READ.

The filename is completed according to the files present in the host file system, and the expanded string *new-string* is returned. *new-string* is always a string containing a file name: either the original string, or a new, more specific string. The value of *success* indicates the status of the completion. Either OLD or NEW means complete success, whereas the empty token list means failure. The following keyword values of *success* are possible:

OLD The string completed to the name of a file that exists.

NEW The string completed to the name of a file that could be created.

Empty token list
The operation failed for one of the following reasons:

- The file is on a file system that does not support completion. *new-string* is supplied as the unchanged string.
- There is no possible completion. *new-string* is supplied as the unchanged string.

- There is more than one possible completion. The given string is completed up to the first point of ambiguity, and the result is supplied as *new-string*.

- A directory name was completed. Completion was not successful because additional components to the right of this directory remain to be specified. The string is completed through the directory name and the delimiter that follows it, and the result is returned in *new-string*.

Filename completion is a host-dependent operation. Genera performs filename completion with the function **fs:complete-pathname**. The documentation on that function contains some useful guidelines: See the function **fs:complete-pathname** in *Program Development Utilities*.

CONTINUE NFILE Command

Command Format:

(CONTINUE *tid handle*)

Response Format:

(CONTINUE *tid*)

CONTINUE resumes a data transfer that was temporarily suspended due to an asynchronous error. Each asynchronous error description has an optional argument of RESTARTABLE, indicating whether it makes any sense to try to continue after this particular error occurred. CONTINUE tries to resume the data transfer if the error is potentially recoverable, according to the RESTARTABLE argument in the asynchronous error description. For a discussion of asynchronous errors: See the section "NFILE Error Handling", page 226.

handle is a required string-type argument that refers to the handle of the data channel that received an asynchronous error. That data channel could have been in use for a data stream or direct access transfer. *handle* cannot be a direct file identifier.

If the asynchronous error description does not contain the RESTARTABLE argument, and the user issues the CONTINUE command anyway, the server gives a command error response.

CREATE-DIRECTORY NFILE Command

Command Format:

(CREATE-DIRECTORY *tid pathname property-pairs*)

Response Format:

(CREATE-DIRECTORY *tid dir-truename*)

CREATE-DIRECTORY creates a directory on the remote file system. The required *pathname* argument is a string identifying the pathname of the directory to be created. The return value *dir-truename* is the pathname of the directory that was

successfully created. Both of these pathnames are examples of *pathname as directory*. For a discussion of the concept of pathname as directory: See the section "Directory Pathnames and Directory Pathnames as Files" in *Program Development Utilities*.

property-pairs is a keyword/value list of properties that further define the attributes of the directory to be created; the allowable keywords and associated values are operating system dependent. If *property-pairs* is supplied as the empty token list, default access and creation attributes apply and should be assured by the server.

For further information on file property keywords and associated values:

See the section "Format of NFILE File Property/Value Pairs", page 192.

See the section "Recognized Keywords Denoting File Properties", page 193.

CREATE-LINK NFILE Command

Command Format:

(CREATE-LINK *tid pathname target-pathname property-pairs*)

Response Format:

(CREATE-LINK *tid link-truename*)

CREATE-LINK creates a link on the remote file system.

pathname is the pathname of the link to be created; *target-pathname* is the place in the file system to which the link points. Both are required arguments. The return value *link-truename* names the resulting link.

If a server on a file system that does not support links receives the CREATE-LINK command, it sends a command error response.

The arguments *pathname* and *target-pathname*, and the return value *link-truename*, are all strings in the full pathname syntax of the server host. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

The required *property-pairs* argument is a token list of keyword/value pairs. These properties and their values specify certain attributes to be given to the link. If no property pairs are given in the command, the server should apply a reasonable default set of attributes to the link.

For further information on file property keywords and associated values:

See the section "Format of NFILE File Property/Value Pairs", page 192.

See the section "Recognized Keywords Denoting File Properties", page 193.

DATA-CONNECTION NFILE Command

Command Format:

(DATA-CONNECTION *tid new-input-handle new-output-handle*)

Response Format:

(DATA-CONNECTION *tid connection-identifier*)

DATA-CONNECTION enables the user side to coordinate the establishment of a new data connection. The user side supplies two required string arguments, *new-input-handle* and *new-output-handle*. These arguments are used by future commands to reference the two data channels that constitute the data connection now being created. *new-input-handle* describes the server-to-user data channel, and *new-output-handle* describes the user-to-server channel. *new-input-handle* and *new-output-handle* cannot refer to any data channels already in use.

Upon receiving the DATA-CONNECTION command, the server arranges for a *logical port* (called *socket* or *contact name* on some networks) to be made available on the foreign host machine. When the server has made that port available, it must inform the user of its identity. The server relays that information in the command response, in the required *connection-identifier*, a string. The server then listens on the port named by *connection-identifier*, and waits for the user side to connect to it.

Upon receiving the success command response, the user side supplies the *connection-identifier* to the local network implementation, in order to connect to the specified port. The data connection is not fully established until the user side connects successfully to that port. This command is unusual in that the successful command response does not signify the completion of the command; it indicates only that the server has fulfilled its responsibility in the process of establishing a data connection.

The *connection-identifier* is used only once; it provides the the user with the correct identity of the logical port that the server has provided. NFILE expects the *connection-identifier* to be a string, but places no further restrictions on the nature or character of the *connection-identifier*; the network and its implementation determine the *connection-identifier*. In all future NFILE commands that need to reference either of the data channels that constitute this data connection, the *new-input-handle* and *new-output-handle* are used.

The DATA-CONNECTION command is used to establish a data connection whenever one is needed. The two data channels that comprise this data connection can be used either for data stream transfers or direct access transfers.

For more information about data connections: See the section "NFILE Control and Data Connections", page 183.

DELETE NFILE Command

Command Format:

(DELETE *tid handle pathname*)

Response Format:

(DELETE *tid*)

DELETE deletes a file on the remote file system.

Either a *handle* or a *pathname* must be supplied, but not both. If given, the *handle* must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. *pathname* is a string in the full pathname syntax of the server host. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

With a *pathname* supplied, the DELETE command causes the specified file to be deleted. DELETE has different results depending on the operating system involved. That is, DELETE causes 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 the *handle* argument is supplied to DELETE, the server deletes the open file bound to the data channel specified by *handle* at close time. This is true in both the output and input cases.

The NFILE DELETE command differs from the QFILE DELETE command, specifically when the *handle* argument is supplied, to indicate that a stream is to be "deleted". In QFILE, when a DELETE command is sent to a stream while it is open, the file is "close-aborted" instead of closed normally. NFILE also offers a way to close-abort a file: give the NFILE CLOSE command and supply the *abort-p* argument as Boolean truth. The NFILE DELETE command cannot be used to close-abort a file.

DIRECT-OUTPUT NFILE Command

Command Format:

(DIRECT-OUTPUT *tid direct-handle output-handle*)

Response Format:

(DIRECT-OUTPUT *tid*)

DIRECT-OUTPUT starts and stops output data flow for a direct access file opening. DIRECT-OUTPUT explicitly controls binding and unbinding of an output data channel to an open direct stream.

direct-handle is a required argument, and *output-handle* is optional.

If supplied, *output-handle* is a request to bind a currently free, user-side-selected output data connection (indicated by the *output-handle*) to the open direct stream designated by the *direct-handle*. The server binds the data channel and begins accepting data from that connection and writing it to the stream.

If the *output-handle* is omitted, this is a request to unbind the channel and terminate the active output transfer.

DIRECTORY NFILE Command

Command Format:

(DIRECTORY *tid input-handle pathname control-keywords properties*)

Response Format:

(DIRECTORY *tid*)

DIRECTORY returns a directory listing including the identities and attributes for logically related groups of files, directories, and links. If the command is successful, a single token list containing the requested information is sent over the data channel specified by *input-handle*, and the data channel is then implicitly freed by both sides. For details on the format of the token list: See the section "NFILE DIRECTORY Data Format", page 202.

pathname specifies the files that are to be described; it is a string in the full pathname syntax of the server host. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

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 returns information 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 of servers on systems that do not allow wildcard matches, when presented with a wildcard.

properties is a token list of keyword/value pairs. If *properties* is omitted or supplied as the empty token list, the server sends along all properties. If any properties are supplied, the user is requesting the server to send only those properties. However, it is never an error for the server to send more information than is requested.

control-keywords Argument to DIRECTORY

control-keywords is a token list of keyword/value pairs. The *control-keywords* affect the way the DIRECTORY command works on the server machine. Although some of the options below request the server to limit (by some filter) the data to be returned, it is never an error if the server returns more information than is requested.

The following keywords are recognized:

DELETED — Treats soft-deleted files as though they still exist. 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. DELETED is ignored on systems that do not support soft deletion.

FAST — Speeds up the operation and data transmission by not listing any properties for the files concerned.

NO-EXTRA-INFO — Specifies that the server is to suppress listing those properties that are generally more difficult or expensive to obtain. For example on Symbolics computers, NO-EXTRA-INFO speeds up the File System Editor (FSEdit) when listing the top level of hierarchical directory systems. This option affects the appearance of directories in the listing by shortening 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 the file system go to the actual directory file to extract information (as opposed to extracting information from the 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 option changes the semantics of DIRECTORY fairly drastically. Normally, the server returns information about all files, directories, and links whose pathnames match 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 the supplied pathname, but whose pathnames expressed as *directory pathnames* match the (potentially wildcarded) *directory portion* of the supplied pathname. The description of the PROBE-DIRECTORY keyword that can be supplied as the *direction* argument of the OPEN command discusses this: See the section "OPEN NFILE Command", page 211.

It is not yet established what servers on hosts that do not support this type of action natively are to do when presented with DIRECTORIES-ONLY and a pathname with a wildcard directory component.

SORTED — This causes the directory listing to be sorted. In a sorted directory listing, multiple versions of a file are consecutive in increasing version number.

NFILE DIRECTORY Data Format

If the NFILE DIRECTORY command completes successfully, a single token list containing the requested directory information is sent on the data channel specified by the *input-handle* argument in the DIRECTORY command. This section describes the format of that single token list, and gives further detail on the *properties* argument to DIRECTORY.

The token list is a top-level token list, so it is delimited by TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END. The top-level token list contains embedded token lists. The first embedded token list contains the empty token list followed by property/value pairs describing property information of the file system as a whole rather than of a specific file. NFILE requires one property of the file system to be present: DISK-SPACE-DESCRIPTION is a string type property describing the amount of free file space available on the system. The following embedded token lists contain the pathname of a file, followed by property/value pairs describing the properties of that file.

The following example shows the format of the top-level token list returned by DIRECTORY, for two files. It is expected that the server return several property/value pairs for each file; the number of pairs returned is not constrained. In this example, two property/value pairs are returned for the file system, two pairs are returned for the first file, and only one pair is returned for the second file.

```
TOP-LEVEL-LIST-BEGIN
LIST-BEGIN     -- the first embedded token list starts here
LIST-BEGIN     -- an empty embedded token list
LIST-END
prop1/value1       -- property/value pairs of file system
prop2/value2
LIST-END
LIST-BEGIN
pathname1          -- pathname of the first file
prop1/value1       -- property/value pairs of first file
prop2/value2
LIST-END
LIST-BEGIN
pathname2          -- pathname of the second file
prop1/value1       -- property/value pairs of second file
LIST-END
TOP-LEVEL-LIST-END
```

The following example is designed to better show the structure of the top-level token list by depicting TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END by parentheses and LIST-BEGIN and LIST-END by square brackets. respectively. The indentation, blank spaces, and newlines in the example are not part of the token list, but are used here to make the structure of the token list clear.

```
([   [ ]     prop1 value1 prop2 value2 ]
 [pathname1 prop1 value1 prop2 value2 ]
 [pathname2 prop1 value1 prop2 value2 ])
```

The *pathname* is a string in the full pathname syntax of the server host. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

For further information on file property/value pairs: See the section "Format of NFILE File Property/Value Pairs", page 192. See the section "Recognized Keywords Denoting File Properties", page 193.

DISABLE-CAPABILITIES NFILE Command

Command Format:

(DISABLE-CAPABILITIES *tid capability*)

Response Format:

(DISABLE-CAPABILITIES *tid cap-1 success-1 cap-2 success-2 ...*)

DISABLE-CAPABILITIES causes a capability to be disabled on the server machine. *capability* is a string naming the capability to be disabled. The meaning of the capability is dependent on the operating system.

The return values *cap-1*, *cap-2*, and so on, are strings specifying names of capabilities. If the capability named by *cap-1* was successfully disabled, the corresponding *success-1* is supplied as Boolean truth; otherwise it is the empty token list.

Although the user can specify only one capability to disable, it is conceivable that the result of disabling that particular capability is the disabling of other, related capabilities. That is why the command response can contain information on more than one capability.

ENABLE-CAPABILITIES NFILE Command

Command Format:

(ENABLE-CAPABILITIES *tid capability password*)

Response Format:

(ENABLE-CAPABILITIES *tid cap-1 success-1 cap-2 success-2 ...*)

ENABLE-CAPABILITIES causes a capability to be enabled on the server machine. The *password* argument is optional, and should be included only if it is needed to enable this particular *capability*. Both *password* and *capability* are strings. The meaning of the capability is dependent on the operating system.

The return values *cap-1*, *cap-2* and so on, are strings specifying names of capabilities. If the capability named by *cap-1* was successfully enabled, the corresponding *success-1* is supplied as Boolean truth; otherwise it is the empty token list.

Although the user can specify only one capability to enable, it is conceivable that the result of enabling that particular capability is the enabling of other, related capabilities. That is why the command response can contain information on more than one capability.

EXPUNGE NFILE Command

Command Format:

(EXPUNGE *tid directory-pathname*)

Response Format:

(EXPUNGE *tid number-of-server-storage-units-freed*)

EXPUNGE causes the directory specified by *pathname* to be *expunged*. Expunging means that any files that have been *soft deleted* are to be permanently removed.

For file systems that do not support soft deletion, the command is to be *ignored*; a success command response is sent, but no action is performed on the file system. In this case, the *number-of-server-storage-units-freed* return value should be omitted.

directory-pathname is a required string argument in the *pathname as directory* format. The *directory-pathname* must refer to a directory on the server file system, and not to a file. For a discussion of pathname as directory: See the section "Directory Pathnames and Directory Pathnames as Files" in *Program Development Utilities*.

The return value *number-of-server-storage-units-freed* is an integer specifying how many records, blocks, or whatever unit is used to measure file storage on the host system, were recovered. This return value should be omitted if the server does not know how many storage units were freed.

The protocol does not define whether *directory-pathname* is really a pathname as directory or a wildcard pathname of files to be expunged. The protocol does not define 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).

FILEPOS NFILE Command

Command Format:

(FILEPOS *tid handle position resync-uid*)

Response Format:

(FILEPOS *tid*)

FILEPOS sets the file access pointer to a given *position*. The *handle* indicates the file to be affected. *handle* must be a data channel handle for a data stream opening, or a direct file identifier for a direct access opening. Both *handle* and *position* are required arguments.

position is an integer indicating to which point in the file the file access pointer is to be reset. *position* is either a byte number according to the current byte size being used, or characters for character openings. Position zero is the beginning of the file. If this is a character opening, *position* is measured in server units, not in Symbolics units.

If the FILEPOS command is given on an input data channel (that is, a data channel currently sending data from server to user), the affected data channel must be resynchronized after the FILEPOS is accomplished. The *resync-uid* is a unique identifier associated with the resynchronization of the data channel. *resync-uid* must be supplied if *handle* is an input handle, but it is not supplied otherwise. For more information on the resynchronization procedure, see the section "NFILE Data Connection Resynchronization", page 187.

In the output case, the user must somehow indicate to the server, on the output data channel, when the data have come to an end. The user side sends the keyword token EOF to do so. Upon receiving that control token, the server is free to position the file pointer according to the *position* given. When the new file position is established, the server resumes accepting data at the new file position.

In most cases, using the direct access mode of transfer is more convenient and efficient than using FILEPOS with a data stream opening.

There are problems inherent in trying to set a file position of a character-oriented file on a foreign host, if one machine is a Symbolics computer and the other is not. Character set translation must take place. See the section "NFILE Character Set Translation", page 181. Because of these difficulties, FILEPOS might not be supported in the future on character files. FILEPOS is not problematic for binary files.

Implementation Hint for FILEPOS NFILE Command

This section provides an implementation hint from the designers and implementors of the Symbolics Lisp Machine NFILE. This section is useful for any programmer implementing an NFILE server program.

The server processing of this command (by the control channel handler) must not attempt to wait for the resynchronization procedure to complete. It is possible that the user could abort between sending the FILEPOS command and reading for the mark and resynchronization identifier. That scenario could leave the sender of the resynchronization identifier, on the server side, blocked for output indefinitely.

Only two commands received on the control connection can break the data channel out of the blocked state described above: CLOSE with *abort-p* supplied as Boolean truth, and the RESYNCHRONIZE-DATA-CHANNEL. Therefore, the control connection must not wait for the control channel to finish performing the resynchronization procedure. This wait should instead be performed by the process managing the data channel.

FINISH NFILE Command

Command Format:

(FINISH *tid handle*)

Response Format:

(FINISH *tid truename binary-p other-properties*)

FINISH closes a file and reopens it immediately with the file position pointer saved, thus leaving it open for further I/O. The arguments, results, and their meaning are identical to those of the CLOSE command. See the section "CLOSE NFILE Command", page 194. FINISH requires a *handle*, which has the same meaning as the *handle* of the CLOSE command.

In the output case, for both direct mode and data stream mode of openings, the server writes out all buffers and sets the byte count of the file. The server sends

the keyword token EOF on the data channel, to indicate that the end of data has been reached. The server leaves the file in such a state that if the system or server crashes anytime after the FINISH command was given, it would later appear as though the file had been closed by this command. However, the file is not closed now; it is left open for further I/O operations. FINISH is a reliability feature.

FINISH is somewhat pointless in the input case, but valid. 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.

An Unsuccessful FINISH Operation

After writing every last bit sent by the user to disk, and before closing the file, the server checks the data channel specified by *handle* to see if an asynchronous error description is outstanding on that channel. That is, the server must determine whether it has sent an asynchronous error description to the user, to which the user has not yet responded with a CONTINUE command. If so, the server is unable to finish the file, and it must send a command error response response, indicating that an error is pending on the channel. The appropriate three-letter error code is EPC. See the section "NFILE Error Handling", page 226.

A Successful FINISH Operation

After the user receives the successful response from the server, active data transfer is resumed. That is, for a data stream input opening, or a direct opening with an input channel active, the data channel is reactivated and resumes sending data from the file at the point where the control channel interrupted it. In the case of a data stream output opening, or a direct opening with an output channel active, the output channel is set back into a state where it is prepared to receive data to transmit to the file at the point where it was interrupted by the FINISH command.

HOME-DIRECTORY NFILE Command

Command Format:

(HOME-DIRECTORY *tid user*)

Response Format:

(HOME-DIRECTORY *tid directory-pathname*)

HOME-DIRECTORY returns the full pathname of the home directory on the server machine for the given *user*.

user is a string that should be recognizable as a user's login name on the server operating system. *directory-pathname* is a string in the pathname as directory format. For a discussion of pathname as directory: See the section "Directory Pathnames and Directory Pathnames as Files" in *Program Development Utilities*.

LOGIN NFILE Command

Command Format:

(LOGIN *tid user password FILE-SYSTEM USER-VERSION*)

Response Format:

(LOGIN *tid keyword/value-pairs*)

LOGIN logs the given *user* in to the server machine, using the *password* if necessary. Both *user* and *password* are string arguments; *user* is required, *password* is optional. An omitted password is valid if the host allows the specified *user* to log in without a password. Depending on the operating system and server, it might be necessary to log in to run a program (in this case the NFILE server program) on the host. LOGIN establishes a user identity that is used by the operating system to establish the file author and determine file access rights during the current session.

The server has the option to reject with an 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. The portion of the NFILE server running in the special login environment could conceivably be capable only of processing the LOGIN command; this is an implementation detail.

FILE-SYSTEM and *USER-VERSION* are optional keyword/value pairs. The *FILE-SYSTEM* keyword/value pair has the same effect as does QFILE's SET-FILE-SYSTEM command; it selects the identity of the file system to which all following commands in this session are to be directed. This argument has meaning only if the server host machine has multiple file systems, and the targeted file system is other than the default file system that a user would get by initiating a dialogue with that host. The *FILE-SYSTEM* argument is an arbitrary token list. If the server does not recognize it, the server gives an appropriate command error response.

Currently, the only use of *FILE-SYSTEM* is for Symbolics Lisp Machine servers to select the FEP hosts. In this case, the first element in the token list is the keyword FEP, and the second element in the token list is an integer, indicating the desired FEP disk unit number. If the server discovers there is no such file system, the server gives a command error response including the three-letter code NFS, meaning "no file system".

The user tells the server what version of NFILE it is running by including the optional *USER-VERSION* keyword/value pair. The value associated with USER-VERSION can be a string, an integer, or a token list. This document describes NFILE user version 2 and server version 2.

Upon receiving the representation of the user version, the server can either adjust certain parameters to handle this particular version, or simply ignore the user version altogether. Currently, the only released versions of NFILE are user version 2 and server version 2.

LOGIN Return Values: *keyword/value-pairs*

The *keyword/value-pairs* is a token list composed of keywords followed by their values. The server includes any or all of the following keywords and their values; they are all optional. The following keywords are recognized:

NAME The value associated with NAME is a string specifying the user identity, in the server host's terms.

PERSONAL-NAME The value associated with PERSONAL-NAME is a string representing the user's personal name, last name first. For example: "McGillicuddy, Aloysius X.".

HOMEDIR-PATHNAME
The value associated with HOMEDIR-PATHNAME is a string in the pathname as directory format, indicating the home directory of the user. For a discussion of pathname as directory: See the section "Directory Pathnames and Directory Pathnames as Files" in *Program Development Utilities*.

GROUP-AFFILIATION
The value associated with GROUP-AFFILIATION is a string specifying the group to which the user belongs.

SERVER-VERSION The value associated with SERVER-VERSION can be a string, an integer, or a token list. The value is a representation of the version of the server is running. Upon receiving the server version, the user can: adjust certain parameters to handle this particular version; accept the version; or close the connection. Currently, the only released versions of NFILE are user version 2 and server version 2.

PROPERTY-INDEX-TABLE
The value associated with PROPERTY-INDEX-TABLE is a token list of keywords. This return value enables the server to inform the user which file properties are meaningful on its file system. The keywords in PROPERTY-INDEX-TABLE can be used by the DIRECTORY command (a user request for information on file properties of a specified directory or directories). The server can specify a certain property by giving an integer that is the index of that file property into the PROPERTY-INDEX-TABLE. This reduces the volume of data sent during directory listings. The first element in PROPERTY-INDEX-TABLE is indexed by the number 0. See the section "DIRECTORY NFILE Command", page 201.

MULTIPLE-FILE-PLISTS NFILE Command

Command Format:

(MULTIPLE-FILE-PLISTS *tid input-handle pathlist characters properties*)

Response Format:

(MULTIPLE-FILE-PLISTS *tid*)

MULTIPLE-FILE-PLISTS returns file property information of one or more files. The server sends the information in a data structure (the format is described later in this section) on the given *input-handle*. *pathlist* is a token list composed of the pathnames in which the user is interested. The pathnames in *pathlist* are strings in the full pathname syntax of the server host. Unlike for the DIRECTORY command, wildcards are not allowed in these pathnames. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

characters is either Boolean truth (indicating that each file is a character file), the empty token list (each file is a binary file), or the keyword DEFAULT. DEFAULT indicates that the server itself is to figure out whether a file is a character or binary file. For more information on the meaning of the DEFAULT keyword: See the section "OPEN NFILE Command", page 211. The value of *characters* can influence some servers' idea of a file's length.

properties is a token list of keywords indicating which properties the user wants returned. The server is always free to return more properties than those requested in the *properties* argument. If *properties* is supplied as the empty token list, the server should transmit all known properties on the files. For a list of keywords associated with file properties: See the section "Recognized Keywords Denoting File Properties", page 193.

The server transmits as much of the requested information as possible on the given *input-handle*. The information is contained in a top-level token list of elements. Each element corresponds with a supplied pathname; the order of the original *pathlist* must be retained in the returned token list. An element is an empty token list if the corresponding file or any of its containing directories does not exist. The elements that correspond to successfully located files are lists composed of *truename* followed by any *properties*. *properties* are keyword/value pairs. *truename* is a string in the full pathname syntax of the server host.

The following example shows TOP-LEVEL-LIST-BEGIN and TOP-LEVEL-LIST-END as parentheses, and LIST-BEGIN and LIST-END with square brackets.

For example, the user supplied a *pathlist* argument resembling:

[*file1 file2 file3*]

The server could not locate *file1* or *file3*, but did locate *file2*, and found the length and author of *file2*. The top-level token list transmitted by the server is:

([] [*truename-of-file2* LENGTH 381 AUTHOR williams] [])

For further details on how file properties and values are expressed: See the section "Format of NFILE File Property/Value Pairs", page 192.

OPEN NFILE Command

Command Format:

(OPEN *tid handle pathname direction binary-p*
TEMPORARY RAW SUPER-IMAGE DELETED PRESERVE-DATES
SUBMIT DIRECT-FILE-ID ESTIMATED-LENGTH BYTE-SIZE
IF-EXISTS IF-DOES-NOT-EXIST)

Response Format:

(OPEN *tid truename binary-p other-properties*)

OPEN 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 OPEN command has the most complicated syntax of any NFILE command. The OPEN command has *required arguments*, an *optional argument*, and many *optional keyword/value pairs*. For details on the sytnax of each of these parts of the OPEN command, See the section "NFILE Command Descriptions", page 189.

The following arguments are required: *pathname*, *direction*, and *binary-p*. *handle* is an optional argument, which must either be supplied or explicitly omitted by means of substituting in its place the empty token list.

The OPEN command has many optional keyword/value pairs, which encode conceptual arguments to the server file system for the OPEN operation. The OPEN optional keyword/value pairs include:

- *TEMPORARY*
- *RAW*
- *SUPER-IMAGE*
- *DELETED*
- *PRESERVE-DATES*
- *SUBMIT*
- *DIRECT-FILE-ID*
- *ESTIMATED-LENGTH*
- *BYTE-SIZE*
- *IF-EXISTS*
- *IF-DOES-NOT-EXIST*

For a detailed description of all the supported OPEN optional keywords: See the section "NFILE OPEN Optional Keyword/Value Pairs", page 214.

The OPEN return values reflect information about the file opened, when the opening is successful. In the case of a probe-type opening, this information is returned when the given file (or link, or directory) exists and is accessible, even though the file (or link, or directory) is not actually opened. For detail on the OPEN return values: See the section "NFILE OPEN Response Return Values", page 219.

The *pathname* OPEN Argument

The *pathname* is a required argument specifying the file to be opened. *pathname* is a string in the full pathname syntax of the server host. See the section "Full Pathname Syntax of the Server Host", page 192.

For some purposes (for example, when the OPEN argument *direction* is supplied as PROBE-DIRECTORY), only the directory specified by this pathname is utilized. See the section "NFILE OPEN Optional Keyword/Value Pairs", page 214.

The *handle* OPEN Argument

The *handle* argument of the OPEN command specifies a data channel to be used for the transfer. Future commands in this session use the same *handle* to specify the open stream that is created by opening the file. It is the user side's responsibility to ensure that *handle* refers to an existing and free data channel that does not require resynchronization before use. A *handle* must be supplied, unless a probe-type opening is desired (that is, the *direction* is supplied as PROBE, PROBE-DIRECTORY, or PROBE-LINK) or a direct access opening is being requested (that is, a *DIRECT-FILE-ID* is supplied). In those cases, the empty token list is supplied for *handle*.

The *direction* OPEN Argument

The *direction* argument must be supplied as one of these keywords: INPUT, OUTPUT, IO, PROBE, PROBE-DIRECTORY, and PROBE-LINK. The meanings of the *direction* keywords are as follows:

INPUT	Specifies that the file is to be opened for input (server-to-user transfer). To request a direct access opening, supply a value for *DIRECT-FILE-ID*. If no *DIRECT-FILE-ID* is supplied, the opening is a data stream opening.
OUTPUT	Specifies that the file is to be opened for output (user-to-server transfer). To request a direct access opening, supply a value for *DIRECT-FILE-ID*. If no *DIRECT-FILE-ID* is supplied, the opening is a data stream opening.
IO	Specifies that interspersed input and output will be performed on the file. This is only meaningful in direct access mode. A *DIRECT-FILE-ID* must also be supplied. See the section "NFILE OPEN Optional Keyword/Value Pairs", page 214.

If *direction* is supplied as PROBE, PROBE-LINK, or PROBE-DIRECTORY, the opening is said to be a probe-type opening. The *DIRECT-FILE-ID* option is meaningless and an error for probe-type openings. The file *handle* must be supplied as an empty token list for probe-type openings.

PROBE	Specifies that the file is not to be opened at all, but simply 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 for an INPUT opening. If the file does exist, the successful command response contains the same information as it would have if the file had been opened for INPUT. If it is a link, the link is followed to its source.

PROBE-LINK Like PROBE, with one difference. PROBE-LINK specifies that if the *pathname* is found to refer to a link, that link is not to be followed, and information about the link itself is to be returned.

PROBE-DIRECTORY

PROBE-DIRECTORY requests information about the directory designated by the *pathname* argument. In the PROBE-DIRECTORY case, the *pathname* argument refers to the directory on which information is requested. In all other cases, the *pathname* refers to a file to be opened. If *pathname* contains a file name and file type, these parts of the *pathname* are ignored for PROBE-DIRECTORY openings 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.

The *binary-p* OPEN Argument

The *binary-p* argument is supplied as Boolean truth (meaning that the data to be transferred are binary data), the empty token list (meaning that character type data are to be transferred), or the keyword DEFAULT. The value of *binary-p* affects the mode in which the server opens the file, as well as informing it whether or not character set translation must be performed.

If *binary-p* is supplied as the empty token list, the opening is said to be a character opening. 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. See the section "NFILE Character Set Translation", page 181. The check (described in the DEFAULT OPEN option) for Symbolics object files is not performed.

If *binary-p* is supplied as Boolean truth, the opening is said to be a binary opening. The user side supplies the byte size via the BYTE-SIZE option; if not supplied, the default byte size is 16 bits. If byte size is less than 8, the file data are transferred byte by byte. If the byte size is 8 or greater, the server transfers each byte of the file as two eight-bit bytes, low-order first. The check for Symbolics object files is not performed.

binary-p can also be supplied as the keyword DEFAULT. DEFAULT specifies that the server itself is to determine whether to transfer binary or character data. DEFAULT is meaningful only for input openings; it is an error for OUTPUT, IO, or probe-type 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 for the file specified by *pathname*.

When *binary-p* is supplied as DEFAULT, on file systems that do not maintain this information, the server is required to perform a heuristic check for Symbolics object files on the first two 16-bit bytes of the file. If the file is determined to be a Symbolics object file, the server performs a BINARY opening with BYTE-SIZE of 16; otherwise, it performs a CHARACTER opening.

The details of the check are as follows: 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), the file is recognized as a Symbolics object file. In any other case, it is not.

NFILE OPEN Optional Keyword/Value Pairs

The OPEN command has many optional keyword/value pairs that encode conceptual arguments to the file system for the OPEN operation.

The following options are used often:

BYTE-SIZE

Must be followed by an integer between 1 and 16, inclusive, or the empty token list. *BYTE-SIZE* is meaningful only for binary openings. *BYTE-SIZE* can be ignored for probe-type openings. It can be omitted entirely for character openings, but if supplied, must be followed by the empty token list. If *binary-p* is supplied as DEFAULT, *BYTE-SIZE* can be omitted entirely, or followed by the empty token list.

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 byte-size on binary input is 16.

For file systems that maintain the innate byte-size of each file, 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 or host operating system can choose any packing scheme that complies with this rule.

Operating systems that do not support byte size must ensure 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, it would be appropriate for a Multics NFILE server to pack:

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 must be specified.

DELETED If supplied as Boolean truth, *DELETED* specifies that "deleted" files are to be treated as though they were not "deleted". *DELETED* is meaningful only for operating systems that support "soft deletion" and subsequent "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.

DELETED can also be followed by the empty token list, which has the same effect as omitting the *DELETED* keyword/value pair entirely. For output openings, *DELETED* is meaningless and an error if supplied.

DIRECT-FILE-ID If supplied, the *DIRECT-FILE-ID* indicates that the opening is to be a direct access mode opening. If not supplied, the opening is a data stream opening. The value of *DIRECT-FILE-ID* is a string, generated by the user, never before used as a *DIRECT-FILE-ID*, and not designating any data channel. The *DIRECT-FILE-ID* is a unique identifier for the direct access stream. It is used for all operations that identify an open server stream rather than a data channel. The *DIRECT-FILE-ID* is used to identify a stream for a direct access opening, just as a file handle is used to identify an open stream for a data stream opening. The PROPERTIES, CLOSE, and RENAME commands use the *DIRECT-FILE-ID* in this way. There are only two NFILE commands applicable to direct access openings (ABORT and CONTINUE) that do not use the *DIRECT-FILE-ID*, but use a data channel handle instead.

PRESERVE-DATES

If supplied as Boolean truth, *PRESERVE-DATES* specifies that the server is to attempt to prevent the operating system from

updating the "reference date" or "date-time used" of the file. This is meaningful only for input openings, and is an error otherwise.

Genera invokes this option for operations such as Show File in the editor, where it wishes to assert that the user did not "read" the file, but just "looked at it". Servers on operating systems that do not support reference dates or users revising or suppressing update of the reference dates must ignore this option.

ESTIMATED-LENGTH

The value of *ESTIMATED-LENGTH* is an integer estimating the length of the file to be transferred. This option is meaningful and permitted only for output openings. *ESTIMATED-LENGTH* enables 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 which it is not useful or interesting. The units of the estimate are characters for character openings, and bytes of the agreed-upon byte 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 "NFILE Character Set Translation", page 181.

IF-EXISTS

Meaningful only for output openings, ignored otherwise, but not diagnosed as an error. The value of *IF-EXISTS* is a keyword that specifies the action to be taken if a file of the given name already exists. 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-EXISTS* option and its value are ignored.

If the user side does not give the *IF-EXISTS* option, the action to be taken if a file of the given name already exists 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) or NEW-VERSION (if the version in the file pathname is the newest version). For file systems not supporting versions, the default is SUPERSEDE. These actions are described below.

IF-EXISTS provides the mechanism for overwriting or appending to files. With the default setting of *IF-EXISTS*, new files are created by every output opening.

Operating systems supporting soft deletion can take different actions if a "deleted" file already exists with the same name

(and type and version, where appropriate) as a file to be created. The Symbolics Lisp Machine file system (LMFS) effectively uses SUPERSEDE, 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 to prevent 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 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 start replacing old data at the beginning of the file; however, the file is not truncated 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 removed or deleted and 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-type 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 a derivative of Common Lisp. The default is as follows: If this is a probe-type opening 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.

The following optional keyword/value pairs are rarely used, if ever:

RAW — If supplied as Boolean truth, *RAW* specifies that character set translation is not to be performed, but that characters are to be transferred intact, without inspection. This option is meaningful only for character openings; it is an error otherwise. It is also an error to supply *RAW* as Boolean truth for probe-type openings. Servers operating natively in the Symbolics character set (for example, Symbolics computers) can ignore this option. *RAW* can also be followed by the empty token list, which has the same effect as if the *RAW* keyword/value pair were omitted entirely.

TEMPORARY — Used by the TOPS-20 server only. *TEMPORARY* 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-type openings. TEMPORARY can also be followed by the empty token list, which has the same effect as if the *TEMPORARY* keyword/value pair were omitted entirely.

SUPER-IMAGE If supplied as Boolean truth, *SUPER-IMAGE* specifies that Rubout quoting is not to be performed. This operation is meaningful only for character openings; it is an error otherwise. It is also an error for probe-type openings. *SUPER-IMAGE* can also be followed by the empty token list, which has the same effect as if the *SUPER-IMAGE* keyword/value pair were omitted entirely.

SUPER-IMAGE mode causes the server to read or write 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 "NFILE Character Set Translation", page 181.

SUBMIT *SUBMIT* is meaningful for output only. If supplied as Boolean truth, *SUBMIT* causes the server to submit the contents of the file being written to the operating system as a job, after the file is closed. VMS is an example of an operating system that could conveniently support *SUBMIT*. *SUBMIT* can also be followed by the empty token list, which has the same effect as if the *SUBMIT* keyword/value pair were omitted entirely. Servers that do not implement this option should give an error response if requested to submit a file to the operating system.

NFILE OPEN Response Return Values

The results of a successful OPEN operation are reported in the command response. Here is the specification of the OPEN response format:

Response Format:

(OPEN *tid truename binary-p other-properties*)

The return values for OPEN and CLOSE are syntactically identical, but the values can change in the time between open and close time.

truename is a string representing the pathname of the file in the full pathname syntax of the server host. It should be determined 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, this string always contains an explicit version number. It always contains 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. On these systems the *truename* when the file is opened is different than the *truename* after it has been closed.

The return value *binary-p* indicates whether the opening is a binary or character opening. For binary openings, *binary-p* is supplied as Boolean truth; for character openings it is the empty token list.

other-properties is a list of keyword/value pairs. *other-properties* must contain CREATION-DATE and LENGTH. AUTHOR should be included if the server operating system has a convenient mechanism for determining the author of the file. The other properties described here can be included if desired.

CREATION-DATE The creation date of the file. The date is expressed in Universal Time format, which measures a time as the number of seconds since January 1, 1900, at midnight GMT. 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.

Some types of file systems support a user-settable quantity, which the user can 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 quantity, in the OPEN response for an output file, is disregarded by the user side, but must nevertheless be present.

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.

LENGTH An integer reporting the length of the file, in characters for character openings and in bytes of the agreed-upon size for binary openings. LENGTH should be reported as zero 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.

AUTHOR The value of AUTHOR is a string representing the name of the author of the file. This is some kind of user identifier, whose format is highly system-specific.

In the best possible case, AUTHOR 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 software also uses AUTHOR 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. The value of BYTE-SIZE is an integer. For details on the ramifications of BYTE-SIZE: See the section "NFILE OPEN Optional Keyword/Value Pairs", page 214. This parameter is only meaningful for BINARY openings. However, if FILEPOS is returned in the *other-properties* list, BYTE-SIZE should also be included, even for character openings.

FILEPOS — An integer giving the position of the logical file pointer, in characters or bytes as appropriate for the type of 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 "NFILE Character Set Translation", page 181.

PROPERTIES NFILE Command

Command Format:

(PROPERTIES *tid handle pathname control-keywords properties*)

Response Format:

(PROPERTIES *tid property-element settable-properties*)

PROPERTIES requests the property information about one file. The file is identified by the *pathname* argument or the *handle* argument, but not both. If *pathname* is supplied, it is a string in the full pathname syntax of the server host. For further details on full pathname syntax: See the section "Full Pathname Syntax of the Server Host", page 192.

If *handle* is supplied, its value is a string identifying an open stream, which implicitly identifies a file. For direct access mode openings, *handle* must be a direct file identifier.

control-keywords is reserved in the current design. However, it is a required argument, and must be supplied as the empty token list. Its presence in the NFILE specification allows for future expansion. In the future the value of *control-keywords* might affect the listing mode.

properties is a token list of keywords indicating the properties the user wants returned. (In command arguments, properties cannot be specified with integers that are indices into the Property Index Table). For a list of keywords associated with file properties: See the section "Recognized Keywords Denoting File Properties", page 193.

The server is always free to return more properties than those requested in the *properties* argument. If *properties* is supplied as the empty token list, the server transmits all known properties on the file.

PROPERTIES Command Response

The server returns the property information for the given file in the command response. The PROPERTIES command does not use any data channels. If the specified file does not exist or is not accessible, the server signals an error and includes an appropriate three-letter error code in the command error response. See the section "NFILE Error Handling", page 226.

The return value *property-element* is a token list. The first element in that token list is the *pathname* of the file, in the full pathname syntax of the server host. The following elements of the *property-element* token list are property/value pairs. The server is expected to return several property/value pairs; the number of pairs is not constrained. For further details on file properties and their associated values: See the section "Format of NFILE File Property/Value Pairs", page 192.

The return value *settable-properties* is a token list of keywords. The number of keywords is not constrained. (Note that integers cannot be used in *settable-properties* to indicate the file property; keywords are to be used instead.) Each keyword supplied in *settable-properties* identifies a property considered settable by the server. The server is implicitly guaranteeing a mechanism for changing the properties reported as settable. The user can change any of the settable properties for this file by using the CHANGE-PROPERTIES command. See the section "CHANGE-PROPERTIES NFILE Command", page 194.

The following example shows the format of the PROPERTIES command response. Remember that the number of property/value pairs and keywords is not constrained; this example has two property/value pairs and three *settable-properties* keywords returned:

```
TOP-LEVEL-LIST-BEGIN
PROPERTIES          -- the name of the command
tid                -- the transaction identifier
LIST-BEGIN
pathname of file
prop1/value1        -- property/value pairs of the file
prop2/value2
LIST-END
LIST-BEGIN
keyword-1           -- file's settable properties
keyword-2
keyword-3
LIST-END
TOP-LEVEL-LIST-END
```

The following example is designed to better show the structure of the top-level token list by depicting `TOP-LEVEL-LIST-BEGIN` and `TOP-LEVEL-LIST-END` by parentheses and `LIST-BEGIN` and `LIST-END` by square brackets. The indentation and newlines in the example are not part of the token list, but are used here to make the structure of the token list clear.

(PROPERTIES *tid* [*pathname prop1 value1 prop2 value2 ...*]
[*keyword1 keyword2 keyword3 ...*])

READ NFILE Command

Command Format:

(READ *tid direct-file-id input-handle count FILEPOS*)

Response Format:

(READ *tid*)

READ requests input data flow for direct access openings. The *direct-file-id* is the same as the DIRECT-FILE-ID argument that was given when opening the file; it designates the open stream from which the characters or bytes are to be transferred. The *input-handle* specifies which data channel should be used for the transfer of data from server to user. The data channel should have been already established, cannot have been deestablished, and must not currently be in use.

count is an integer specifying how many bytes (or Symbolics unit characters, as appropriate) to read. *count* can be supplied as the empty token list, meaning read to the end of the file. If the user specifies a *count* greater than the number of bytes remaining in the file, the server sends the keyword EOF to mark the end of the file.

FILEPOS is an optional keyword/value pair. If the keyword FILEPOS is supplied, it must be followed by an integer. Before any data are transferred, the open stream is positioned to the point specified by the value of *FILEPOS*. The position of the point is measured in server units for character openings; for binary openings it is measured in binary bytes. See the section "FILEPOS NFILE Command", page 205.

Upon receiving the READ command, the server binds the data channel to the open stream and immediately begins transferring data. The server stops when they are all transferred. After the server sends the last requested byte, it unbinds the data channel, freeing it for other use. When the user side has processed the last byte, the user side assumes that the data channel can now be reused for another data transfer.

RENAME NFILE Command

Command Format:

(RENAME *tid handle pathname to-pathname*)

Response Format:

(RENAME *tid from-pathname to-pathname*)

RENAME requests the server to give a file a new name. This is NFILE's interface to the system's native rename operation, with all of its system-specific semantics and constraints.

Either a *handle* or a *pathname* (but not both) specifies the file that is to receive a new name. The argument *to-pathname* designates that new name. The return value *from-pathname* gives the full original name of the file, and *to-pathname* gives the full new name of the file. For systems that support version numbers, the return values can differ in version number from the values of the arguments given to RENAME.

The arguments *pathname* and *to-pathname* and the return values *from-pathname* and *to-pathname* are strings in the full pathname syntax of the server host. See the section "Full Pathname Syntax of the Server Host", page 192.

If the file to be renamed is specified by a *pathname*, the file should be renamed immediately. If the file is specified by *handle*, it is acceptable to wait until close-time to rename the file.

Some operating systems can rename only within a directory. Nevertheless, the *to-pathname* of the RENAME must be fully specified; the server on these systems must check for and reject an attempted cross-directory rename.

RESYNCHRONIZE-DATA-CHANNEL NFILE Command

Command Format for an Input Handle:

(RESYNCHRONIZE-DATA-CHANNEL *tid handle*)

Response Format for an Input Handle:

(RESYNCHRONIZE-DATA-CHANNEL *tid identifier*)

Command Format for an Output Handle:

(RESYNCHRONIZE-DATA-CHANNEL *tid handle identifier*)

Response Format for an Output Handle:

(RESYNCHRONIZE-DATA-CHANNEL *tid*)

RESYNCHRONIZE-DATA-CHANNEL begins a prescribed procedure between user and server over the unsafe data channel specified by *handle*. The resynchronization procedure clears the data channel of any unwanted data, and restores the data channel to a safe state, ready to transfer data again.

All arguments to RESYNCHRONIZE-DATA-CHANNEL are required.

For a detailed description of how the user and server coordinate the resynchronization of data channels: See the section "NFILE Data Connection Resynchronization", page 187.

Implementation Hints for RESYNCHRONIZE-DATA-CHANNEL NFILE Command

This section provides implementation hints from the designers and implementors of Symbolics NFILE. This section is useful for any programmer implementing an NFILE server program.

Resynchronizing an Output Data Channel

- The server will probably want to dispatch the looping and reading to the logical data process. Looping reading for the resynchronization identifier in the control channel is not a viable option. If the user side fails to send the resynchronization identifier (for example, due to a user abort) the control channel can never be broken out of this loop.

- The user side can either send the control channel command first, or send the marks and identifiers first.

  Sending the marks first is problematic, because the data channel at the other end might not be reading them (for it has not yet been so instructed by the control channel). The user might then become blocked for output, thus prohibiting sending of the RESYNCHRONIZE-DATA-CHANNEL command.

  On the other hand, sending the control channel command first requires that the user side can send the marks and identifiers between sending the control channel command and receiving a response for it. The response will never come until the marks and identifiers have been successfully received. The user implementation must allow for this one case of a command where a subroutinal "send command and wait for response" is inapplicable.

Resynchronizing an Input Data Channel

- The server control process should dispatch the data process to send the mark, and not wait, lest the data process become blocked for output due to a user abort. The control process must go back to its command loop, to possibly receive a command that might break the data process out of that block.

UNDATA-CONNECTION NFILE Command

Command Format:

(UNDATA-CONNECTION *tid input-handle output-handle*)

Response Format:

(UNDATA-CONNECTION *tid*)

UNDATA-CONNECTION explicitly deestablishes a data connection from the user side. The user side has the option of deestablishing data connections at its discretion. There is no place in the protocol where deestablishment of data connections is required, other than at the end of the session, where it is implicit.

The data connection to be deestablished is the one designated by the *input-handle* and *output-handle* arguments. These two handles must refer to the same data connection.

It is not permitted to explicitly deestablish a data connection either of whose channels is active. If the session is terminated by the breaking of the control connec-

tion, 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.

The Symbolics user implementation deestablishes data connections that have not been used for a long time.

For more information about data connections: See the section "NFILE Control and Data Connections", page 183.

11.6.3.11. NFILE Error Handling

NFILE recognizes two types of errors: *command response errors* and *asynchronous errors*.

Command response errors:

- Signify an error associated with the command
- Occur frequently in normal operations

Asynchronous errors:

- Are not related to any specific command
- Are associated with an erring data channel
- Typically indicate a problem in the transfer, such as running out of disk space or allocation, or a bad disk record
- Occur rarely in normal operations

NFILE Command Response Errors

NFILE command response errors are sent over from the server to the user across the control connection as top-level token lists, in this format:

(ERROR *tid three-letter-code error-vars message*)

ERROR is a keyword. The *tid* is the transaction identifier of the command that encountered this error. The arguments *three-letter-code*, *error-vars*, and *message* are all required.

The *three-letter-code* provides the information on what kind of an error was encountered. For a table of the three-letter codes and their meanings: See the section "NFILE Three-letter Error Codes", page 228.

message is a string that is displayed to the human user of the protocol.

error-vars is a keyword/value list. The three possible keywords are: PATHNAME, OPERATION, and NEW-PATHNAME. Before transmitting an error, the server looks at the type of error to see if it can easily determine the value of any of the keywords. If so, the server includes the keyword/value pair in its error. If not, the keyword/value pair is omitted. The value associated with OPERATION is the keyword naming the NFILE command that failed. The values associated with PATHNAME and NEW-PATHNAME are strings in the full pathname syntax of the server host.

For example, the server failed in an attempt to rename a file. The server can then determine the pathname of the original file, the operation (RENAME), and the new pathname (the target pathname) of the file; the server includes all three keywords and their values in its error description.

NFILE Asynchronous Errors

When a data channel process, in either direction, encounters an error condition, the server sends an asynchronous error description. An asynchronous error description consists of a top-level token list. Typically, asynchronous errors indicate error conditions in the transfer, such as running out of disk space or allocation, or a bad disk record.

The format of asynchronous error descriptions is:

(ASYNC-ERROR *handle three-letter-code error-vars message)*

ASYNC-ERROR is a keyword. The *handle* argument identifies the erring data channel. The arguments *three-letter-code*, *error-vars*, and *message* are all required. Their meanings are the same as in NFILE command error responses: See the section "NFILE Command Response Errors", page 226.

When the server detects an asynchronous error on an input data channel, the server sends an asynchronous error description on that data channel itself. When an asynchronous error occurs on an output data channel, the asynchronous error description is sent on the control connection.

Some asynchronous errors are *restartable*. In this context, restartable means it makes sense to try to resume the operation. One example of a restartable error is an attempt to write a file to a file system that is out of room. The server side indicates whether an asynchronous error is restartable by prepending the keyword RESTARTABLE and the associated value Boolean truth to the *error-vars* list. To proceed from a restartable error, the user side sends a CONTINUE command over the control connection.

On any asynchronous error, either input or output, the data channel on the server side enters an *asynchronous error received* state. The server can exit that state in one of two ways: by receiving a CONTINUE command or a CLOSE command with the *abort-p* argument supplied as Boolean truth.

On a normal CLOSE (not a close-abort), the server side checks the channel it was requested to close. If an asynchronous error description has been sent on the data channel, but not yet processed by CONTINUE, the server side does not close the channel, but sends a command error response. The same thing happens on a FINISH command received on a channel that has an asynchronous error pending. In both cases, the *three-letter code* included in the command error response is EPC, for Error Pending on Channel.

NFILE Three-letter Error Codes

NFILE recognizes a set of *three-letter codes*, each one representing an error condition. The set of codes enables all operating systems to use one error-reporting mechanism. Some operating systems will never encounter certain of the error conditions. Upon detecting an error, the NFILE server should characterize the error by choosing the three-letter code that best describes the error. The three-letter code is an argument in both the command response error and asynchronous error messages from the server to the user.

Some errors fit logically into two error codes. For example, suppose the server could not delete a file because the file was not found. This error could be considered either CDF (Cannot Delete File) or FNF (File Not Found). In this case, File Not Found gives more specific and valuable information than Cannot Delete File. Since the protocol does not allow more than one error code to be reported when an error occurs, the server must choose the most appropriate error code.

This is the error table:

ACC *Access error.* This indicates a protection-violation error.

ATD *Incorrect access to directory.* A directory could not be accessed because the user's access rights to it did not permit this type of access.

ATF *Incorrect access to file.* A file could not be accessed because the user's access rights to it did not permit this type of access.

BUG *File system bug.* This includes all protocol violations detected by the server, as well as by the host file system.

CCD *Cannot create directory.* An error occurred in attempting to create a directory.

CDF *Cannot delete file.* The file system reported that it cannot delete a file.

CCL *Cannot create link.* An error occurred in attempting to create a link.

CIR *Circular link.* An operation was attempted on a pathname that designates a link that eventually links back to itself.

CRF *Cannot rename file.* An error occurred in attempting to rename a file.

CSP *Cannot set property.* An error occurred in attempting to change the properties of a file. This could mean that you tried to set a property that only the file system is allowed to set, or a property that is not defined on this type of file system.

DAE *Directory already exists.* A directory or file of this name already exists.

DAT *Data error.* The file system contains bad data. This could mean data errors detected by hardware or inconsistent data inside the file system.

DEV *Device not found.* The device of the file was not found or does not exist.

DND *"Don't delete" flag set.* Deleting a file with a "don't delete" flag was attempted.

DNE *Directory not empty*. An invalid deletion of a nonempty directory was attempted.

DNF *Directory not found*. The directory was not found or does not exist. This refers specifically to the containing directory; if you are trying to access a directory, and the actual directory you are trying to access is not found, you should signal FNF, for *File Not Found*.

EPC *Error pending on channel*. The server cannot close the channel in attempting to close or finish the channel. This code is used only by NFILE, and not by QFILE. See the section "CLOSE NFILE Command", page 194. See the section "FINISH NFILE Command", page 206.

FAE *File already exists*. The file could not be created because a file or directory of this name already exists.

FNF *File not found*. The file was not found in the containing directory. The TOPS-20 and TENEX "no such file type" and "no such file version" errors should also report this condition.

FOO *File open for output*. Opening a file that was already opened for output was attempted.

FOR *Filepos out of range*. Setting the file pointer past the end-of-file position or to a negative position was attempted.

FTB *File too big*. File is larger than the maximum file size supported by the file system.

HNA *Host not available* The file server or file system is intentionally denying service to user. This does not mean that the network connection failed; it means that the file system is explicitly not available.

IBS *Invalid byte size*. The value of the "byte size" option was not valid.

ICO *Inconsistent options*. Some of the options given in this operation are inconsistent with others.

IOD *Invalid operation for directory*. The specified operation is invalid for directories, and the given pathname specifies a directory, in directory pathname as file format.

IOL *Invalid operation for link*. The specified operation is invalid for links, and this pathname is the name of a link.

IP? *Invalid password*. The specified password was invalid.

IPS *Invalid pathname syntax*. This includes all invalid pathname syntax errors.

IPV *Invalid property value*. The new value provided for the property is invalid.

IWC *Invalid wildcard*. The pathname is not a valid wildcard pathname.

LCK *File locked*. The file is locked. It cannot be accessed, possibly because it is in use by some other process.

LIP *Login problems.* A problem was encountered while trying to log in to the file system.

MSC *Miscellaneous problems.*

NAV *Not available.* The file or device exists but is not available. Typically, the disk pack is not mounted on a drive, the drive is broken, or the like. Operator intervention is probably required to fix the problem, but retrying the operation is likely to succeed after the problem is solved.

NER *Not enough resources.*

NET *Network problem.* The file server had some sort of trouble trying to create a new data connection, or perform some other network operation, and was unable to do so.

NFS *No file system.* The file system was not available. For example, this host does not have any file systems, or this host's file system cannot be initialized or accessed for some reason, or the file system simply does not exist.

NLI *Not logged in.* A file operation was attempted before logging in. Normally the file system interface always logs in before doing any operation, but this problem can occur in certain unusual cases in which logging in has been aborted.

NMR *No more room.* The file system is out of room. This can mean any of several things:

- The entire file system is full.
- The particular volume involved is full.
- The particular directory involved is full.
- The allocate quota has been exceeded.

RAD *Rename across directories.* The devices or directories of the initial and target pathnames are not the same, but on this file system they are required to be.

REF *Rename to existing file.* The target name of a rename operation is the name of a file that already exists.

UKC *Unknown operation.* An unsupported file system operation was attempted, or an unsupported command was attempted.

UKP *Unknown property.* The property is unknown.

UNK *Unknown user.* The specified user name is unknown to this host.

UUO *Unimplemented option.* An option to a command is not implemented.

WKF *Wrong kind of file.* This includes errors in which an invalid operation for a file, directory, or link was attempted.

WNA *Wildcard not allowed.*

11.7. Namespace Protocols

11.7.1. 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 the time from which 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" in *Site Operations*.

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

11.7.2. 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 an RFC/ANS transaction. The RFC specifies the name of the namespace and the corresponding ANS contains the timestamp as characters representing a decimal number.

11.8. Chaosnet

The documentation in this section describing Chaosnet was originally part of the Massachusetts Institute of Technology Artificial Intelligence Lab Memo 628 copyright June, 1981.

11.8.1. Introduction to Chaosnet

Chaosnet is a *local network*, that is, a system of communications 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.

All Symbolics computers support Chaosnet. In Symbolics terminology, Chaos is a type of network. If a site supports Chaosnet:

- The site's namespace database has a network object of type Chaos.
- Hosts have Chaosnet addresses; the addresses are stored in the **address** attribute of the host objects.
- Hosts can communicate with other hosts on the Chaosnet using Chaos protocols; these protocols are stored in the **service** attributes of the host object.

The design of Chaosnet was greatly simplified by ignoring problems irrelevant to local networks. Chaosnet contains no special provisions for problems 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 features unnecessary for local-area networks, such as multiple levels of service or secure communication (other than by end-to-end encryption).

The original design of Chaosnet consisted of two parts—the hardware and the software—which, while logically separable, were designed for each other. Symbolics no longer uses the Chaosnet-specific hardware, but uses standard Ethernet hardware instead.

Network nodes contend for access to an Ethernet cable, over which they can transmit packets addressed to other network nodes. The software defines higher-level protocols in terms of packets.

See the section "Format of Chaosnet Addresses", page 27.

11.8.1.1. References to Chaosnet Protocol Specifications

The Symbolics documentation describing Chaosnet was originally part of the Massachusetts Institute of Technology Artificial Intelligence Lab Memo 628, copyright June 1981.

Chaosnet implements several standard Arpanet protocols, which are documented as ARPANET Requests for Comments. See the section "References to IP/TCP Protocol Specifications", page 156.

For information on NFILE: See the section "NFILE File Protocol", page 177.

The following documents are of some related interest:

[CPR] C. Ryland, TOPS-20 Chaosnet Manual, unpublished.

[UNIBUS] PDP-11 Peripherals Handbook, Digital Equipment Corporation.

11.8.2. Overview of the Chaosnet Software Protocol

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 Chaosnet protocol was designed to be simple, for the sake of reliability and to allow its use by modest computer systems. A minimal implementation exists for a single-chip microcomputer. It was important to design out bottlenecks like those that were found in the Arpanet prior to the advent of IP/TCP, such as the control-link that was shared between multiple connections and the need to acknowledge each message before the next message could be sent.

11.8.2.1. Chaosnet Connections

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 can break the connection off entirely, informing both user processes. User programs can either deal in terms of packets, or ignore packet boundaries and treat the connection as two unidirectional 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 network provides flow control, includes a way by which prospective communicants can get in touch with each other (called *contacting* or *rendezvous*), and provides various network maintenance and housekeeping facilities.

11.8.2.2. Chaosnet Contact Names

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.

The capability of automatically discovering which host to connect to in order to obtain a particular service is a subject for higher-level protocols and for further research. Chaosnet makes no provisions for this capability.

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.

When 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 file system limits the names of automatically started servers 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 "Chaosnet Connection Establishment", page 245.

11.8.2.3. Chaosnet Addresses and Indices

Each node (or host) on the network is identified by a unique address: See the section "Format of Chaosnet Addresses", page 27.

These addresses are used in the routing of packets. There is a table that relates symbolic host names to numeric host addresses; for Symbolics computers this is the namespace database.

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 or Ethernet cables, 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 "Chaosnet Routing", page 239.

When a host is connected to an Ethernet cable, its hardware address and Chaosnet address are coordinated through Address Resolution Protocol [ARP]. When a host is connected to a Chaosnet cable, the host's hardware address on that Chaosnet cable 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 that 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 unique-guarantee 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 unique-guarantee 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 namespace.

11.8.2.4. Chaosnet Packet Numbers

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 "Chaosnet Flow and Error Control", page 242.) Uncontrolled packets are simply transmitted; they 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 wraparound 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 (LOGTEST #o(00000 (- A B))
    <A is less>)
```

Do not use the LESSP (or <) function.

11.8.2.5. Chaosnet Packet Contents

A packet consists of a header, which is eight 16-bit words, and zero or more 8-bit or 16-bit bytes of accompanying data.

The following are the eight 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 "Technical Details of the Chaosnet Software Protocol", page 245.

  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 "Chaosnet Routing", page 239.

  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 con-

nection 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 "Chaosnet Flow and Error Control", page 242.

11.8.2.6. Chaosnet Data Formats

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 *Symbolics Common Lisp Language Concepts*. 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 that 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 that 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 see the data in a packet, and its header in the native form of the machine they are running on. The Chaosnet automatically applies the necessary conversions. 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 that 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 checksums could detect, such as software bugs in a Network Control Program, either do not occur or will produce symptoms so obvious they will be detected and fixed immediately.

11.8.2.7. Chaosnet Routing

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 can be a multistep process involving transmission through several subnets, since there might 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 connected to more than one subnet acts as a *bridge* and *forwards* packets from one subnet to another when necessary. 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 that is also performing other functions, and makes reliable bridge software easy to implement.

*Bridge*s and *gateway*s differ, in our terminology, in this way: 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 handle connections or flow control. A gateway, on the other hand, interconnects two networks with differing protocols and must understand and translate the information passing through it. Gateways might also have to handle 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 recovers discarded packets, or decides 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 can have come in from the network or can 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 can 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. The routing mechanism discovers which bridge is best to use. *Fixed Bridge* is the same, except that the automatic mechanism does not 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 that depends on the type. For a direct connection, this identifies the piece of hardware that 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 existing 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 are 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 more than the cost of two hops. Otherwise the routing algorithm is not well-behaved. Suppose subnet A has two bridges (α and β) on it, and bridge α is connected to subnet B but bridge β is not (it goes to some irrelevant subnet). Then if the costs increase too fast and bridges α and β do not broadcast their RUT packets exactly simultaneously, sometimes packets for subnet B might be sent to bridge β because its cost appears lower. Bridge β then sends them to bridge α, 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 to which this bridge can get, 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 broadcast. Adding the subnet cost eliminates loops, and selects 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 than or equal to the current cost, the cost and the address of the bridge are entered into the routing table, provided that the subnet's routing table entry is not of the Direct or Fixed Bridge type.

When multiple equivalent bridges exist, 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 more complex routing tables, which remember more than one possible route and use them according to their relative costs. So far, however, 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 that accesses the Chaosnet through a front-end PDP-11, we define the interface between the two computers as a subnet, and regard the

PDP-11 as a bridge that 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 that are to be accessed through the Chaosnet, since they can simply look like hosts on the PDP-11's private subnet.

In the case of a host attached to more than one subnet, it is undesirable for the host to have more than one address, since this would complicate user programs that 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 that can forward to that host. Sometimes, we simplify the routing by inventing a new subnet that contains only that host and has no physical realization. The host's address is an address on the fake subnet. All of the host's network attachments are regarded as bridges that 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.

11.8.2.8. Chaosnet Flow and Error Control

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 that are damaged in transmission, that won't fit in buffers, that are duplicated or out-of-sequence, or that 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 does not have to buffer any packets, but acknowledges 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 that 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 that 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 might 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 often enough, the data does not flow smoothly, because the window often appears full to the sender when it is not. If receipts are not sent often enough, there are 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 that 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 that should have been acknowledged but have not been is more than one third the window size, an STS is generated to acknowledge them. Thus the preferred batch size for acknowledgement is one third the window size. The advantage of this size is that if one STS is lost, another is generated before the window fills up (at the two-thirds point).

When a packet is received with the same packet number as one that 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 stimulates 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 of a second in ITS.) Retransmission occurs every half 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 can cause a faster retransmission 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 an 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 that have unacknowledged packets outstanding (a nonempty window) and on connections that 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 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 one half second's 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 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 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.

11.8.3. Technical Details of the Chaosnet Software Protocol

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 a three-letter code.

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.

11.8.3.1. Chaosnet Connection Establishment

This section presents the protocols and packet types associated with creating and destroying connections. First the various connection-establishment protocols are described and then the packets are detailed.

There are several connection-initiation protocols implemented in Chaosnet. In addition to those described here, there is also a broadcast mechanism. For more information, see the section "Chaosnet Broadcast", page 251.

Note that Chaosnet does not have a symmetric close protocol. For more information, see the section "Chaosnet Connection Closing", page 251.

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 that 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. For more information, see the section "Chaosnet Connection States", page 253.

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. For more information, see the section "Chaosnet Connection States", page 253. 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 that is not in the RFC-sent state, it is simply discarded and an STS is sent. For more information, see the section "Chaosnet Connection States", page 253. This happens if the connection is opened while a retransmitted OPN packet is in transit through the network, or if the STS that 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 might be lost. Protocols that require a reliable end-of-data indication should use the mechanism for that before sending CLS. For more information, see the section "Chaosnet End-of-Data", page 250.

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 that sends it might 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.

FWD is a response to RFC that indicates that the desired service is not available from the process contacted, but might 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.

ANS 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.

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 might 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 is retransmitted. If the STS is lost, the OPN will be retransmitted. If the OPN is lost, the RFC is retransmitted superfluously and the OPN is 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 can begin to flow through the connection. The user process can begin transmitting data when it sees the OPN. The server process can 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 an LOS, believing them to be a violation of protocol, and this would destroy the connection before it was fully established.

Once data packets begin to flow, they are subject to the flow and error control protocol. For more information, see the section "Chaosnet Flow and Error Control", page 242. 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 a CLS rather than an 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 an 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 re-

liable-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 database.

A connection is 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.

An 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 that 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. For more information, see the section "Chaosnet Connection States", page 253.

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 an 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. For more information, see the section "Chaosnet Connection States", page 253. The server reads the RFC and decides whether or not to open the connection.

An OPN is the usual positive response to an 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 an OPN is the same as that of an 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 is no longer retransmitted.

An 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 that is not in the RFC-sent state, it is simply discarded and an STS is sent. For more information, see the section "Chaosnet Connection States", page 253. This happens if the connection is opened while a retransmitted OPN packet is in transit through the network, or if the STS that acknowledges an OPN is lost in the network.

A CLS is the negative response to an 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.)

A CLS is also used to close a connection after it has been open for a while. Any data packets in transit might be lost. Protocols requiring a reliable end-of-data indication should use the mechanism for that before sending a CLS. For more information, see the section "Chaosnet End-of-Data", page 250.

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.

A CLS is an uncontrolled packet, so the program that sends it might go away immediately afterwards, leaving nothing to retransmit the CLS. Since there is no error recovery or retransmission mechanism for a CLS, its use 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 an RFC that indicates that the desired service is not available from the process contacted, but might be available at a 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. An FWD is an uncontrolled packet; if it is lost in the network, the retransmission of the RFC presumably stimulates 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. An ANS is an uncontrolled packet; if it is lost in the network, the retransmission of the RFC presumably stimulates an identical ANS.

11.8.3.2. Chaosnet Status Packets

An STS is an uncontrolled packet that 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 that sent the STS). The byte count is currently always 4. This will change if the protocol is revised to add additional items to the STS packet.

An 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. For more information, see the section "Chaosnet Flow and Error Control", page 242.

An LOS is an uncontrolled packet that 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 an LOS whose destination corresponds to an existing 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. LOSs that don't correspond to connections are simply ignored.

An LOS is sent in response to situations such as the arrival of:

- A data packet or an STS for a connection that does not exist or is not open
- A packet from the wrong source for its destination
- 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 can 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 an SNS to a connection not in the Open state, nor in response to an LOS to a nonexistent or broken connection. These rules are important to make the protocols work without timing errors. An OPN or an SNS to a nonexistent connection elicits an LOS.

11.8.3.3. Chaosnet Data

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.

UNC is an uncontrolled packet with user data in 8-bit bytes in the data field. It exists so that user-level programs can 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 might be necessary to avoid running out of buffers. A connection cannot have more input packets queued, awaiting the attention of the user program than the window size of the connection, except that you can always have one UNC packet queued. If no normal data packets are in use, up to one more UNC packet than the window size can be queued.

UNC packets are also used by the standard protocol for encapsulating packets of foreign protocols for transmission through Chaosnet. For more information, see the section "Using Foreign Protocols in Chaosnet", page 257.

11.8.3.4. Chaosnet End-of-Data

An EOF is a controlled packet that 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 can 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 simply ignore any data in an EOF.

EOF packets are used in the recommended protocol for closing a Chaosnet connection. For more information, see see the section "Chaosnet Connection Closing", page 251.

11.8.3.5. Chaosnet Connection Closing

This section describes the recommended way to determine reliably that all data have been transferred before closing a connection (for applications where that is an important consideration).

The important issue is that neither side can 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 (a 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 end of the timeout.

To use this protocol in a bidirectional fashion, where both parties to the connection are sending data simultaneously, you must 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. This asymmetrical protocol guarantees that each side gets a chance to know that both sides agree all the data have been transferred. The first CLS is only sent after both sides have waited for their (first) EOF to be acknowledged.

11.8.3.6. Chaosnet Broadcast

Chaosnet includes a generalized broadcast facility, intended to satisfy such needs 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 that understand the service it is looking for respond. The response can be any valid response

to an RFC. Typically, a BRD is used in a simple-transaction mode, and the response is an ANS packet. Actually, it can be any number of ANS packets since multiple hosts can 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 is an OPN packet; only the first OPN succeeds in opening a connection. A 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. For more information, see the section "Application-Level Chaosnet Protocols", page 254. 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 a 1 for each desired subnet (often all of them).

In the packet header, the destination host and index are 0. The source host and index are the intended recipient of the reply (ANS or OPN). The acknowledgement field contains the number of bytes in the bit map (this is normally 32). 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 the declared length of the bit map are considered 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 RFCs, even to the extent that a contact name of STATUS must retrieve the host's network throughput and error statistics. BRD packets are never refused with a CLS, however; broadcast requests to nonexistent servers should simply be ignored, and no CLS reply should be sent. Most operating systems 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 remains 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 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 an 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 also apply to most duplicated BRDs.

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.

11.8.3.7. Chaosnet Low-level Details

MNT is a special packet type reserved for the use of network maintenance programs. Normal NCPs should discard any MNT packets they receive. MNT packets are an escape mechanism to allow special programs to send packets guaranteed not to get confused with normal packets. MNT packets are forwarded by bridges, although usually one would not depend 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 from 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. For more information, see the section "Chaosnet Routing", page 239.

11.8.3.8. Chaosnet Connection States

A user process gets to Chaosnet by means of a capability or channel (dependent on the host operating system) that 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 can 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 can 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 an 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 changes 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 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 an LOS reply as usual, as do any ANSs, CLSs, and so on, received at this point.

Lost The connection has been broken by receipt of an LOS packet.

Incomplete Transmission The connection has been broken because the other end has ceased to transmit and to respond to the SNS. Either the network or the foreign host is down. (This can also happen when 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. For more information, see the section "Using Foreign Protocols in Chaosnet", page 257.

11.8.4. Application-Level Chaosnet Protocols

This section briefly documents the higher-level protocols of the most general interest. All protocols other than STATUS are optional and are implemented only by hosts that need them. All hosts are required to implement the STATUS protocol since it is used for network maintenance.

11.8.4.1. Chaosnet Status Protocol

The STATUS protocol is used to:

- Determine whether a host is up.
- Determine whether an operable path through the network exists between two hosts.

- Monitor network error statistics.
- Debug new Network Control Programs and new Chaosnet hardware.

The **zl:hostat** function and the Show Hosts command use this protocol.

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 used primarily for network maintenance. To provide a rapid response, the reply to a STATUS request should be generated by the Network Control Program, rather than by starting up a server process.

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, 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.

All items are optional, according to the count field, and extra items not defined here can 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, or for any other reason.
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, or any other reason involving data arriving faster than the host can store it.
words 10-11	The number of incoming packets from this subnet with CRC errors. These were either transmitted wrong from the start, or damaged in transmission.
words 12-13	The number of incoming packets from this subnet that 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. This is zero on most Ethernet hardware.

words 14-15	The number of incoming packets from this subnet that 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 block format. The identification is a subnet number and the counts are as above, except that they are only 16 bits instead of 32 and consequently might overflow. This format should no longer be sent by any hosts.

Identification numbers of 1000 octal and up are reserved for future use.

11.8.4.2. Chaosnet Telnet and Supdup Protocols

The standard Internet Telnet and Supdup protocols exist in identical form in Chaosnet. These protocols provide **:login** service, allowing 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 to 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.

11.8.4.3. Chaosnet File Access Protocols

The NFILE and QFILE protocols provide **:file** service, enabling Symbolics computers to access files on network file servers. NFILE has a higher desirability than QFILE, and is the recommended Chaosnet file access protocol. Because NFILE is built on the **:byte-stream-with-mark** medium, it provides enhanced reliability (especially against interrupts) when compared to QFILE, which is built on **:chaos**.

For a complete description of NFILE: See the section "NFILE File Protocol", page 177.

Some computers running ITS, TOPS-20, UNIX, or VAX/VMS are equipped to act as file servers for QFILE. A user end for QFILE also exists for each of these systems, and is used for general-purpose file transfer.

11.8.4.4. 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 "Chaosnet End-of-Data", page 250.

The fact that the RFC got an affirmative response indicates that the recipient is in fact present and accepting messages. The message text should begin with a suitable header, naming the user who 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.

11.8.4.5. Chaosnet Name Protocol

The standard Internet Name/Finger protocol 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 Symbolics 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:minutes, the user's full name, and the user's group affiliation.

11.8.4.6. Chaosnet Time Protocol

The standard Internet Time protocol 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, which do not have hardware calendar-clocks, use this protocol to find out the date and time when they first come up.

11.8.5. Using Foreign Protocols in Chaosnet

Foreign protocols that 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 connec-

tion 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.

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 that 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 a 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 a UNC packet through a Chaosnet channel that is in the *Closed* state, the channel goes into the *Foreign* state and the NCP assumes that the user is not using normal Chaosnet protocol, but is using Chaosnet to transport packets of some other protocol. See the section "Chaosnet Connection States", page 253. The NCP fills in the source address and index in these packets, but accepts 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 are 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 a 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 a 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 use 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 that 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 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 a 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 a program that is not specifically supposed to deal with such packets would never receive one.

11.8.6. Symbolics Implementation of Chaosnet

The Symbolics implementation of Chaosnet consists of a set of Lisp functions and data structure definitions in the **chaos** package. There are three important data structures:

chaos:conn Represents a connection.

chaos:pkt Represents a packet.

chaos: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 that 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."

11.8.6.1. Opening and Closing Chaosnet Connections

Opening and Closing Chaosnet Connections on the User Side

chaos:connect *host contact-name* &optional *window-size timeout* *Function*

Opens a stream connection, and returns a **chaos:conn** if it succeeds, or signals an error. *host* can 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. **chaos:simple** returns an ANS packet or signals an error. 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 *conn* is open, a CLS packet is sent containing the string *reason*. To reject RFCs, use **chaos:reject** instead of this function.

chaos:open-foreign-connection *host index* &optional *pkt-allocation distinguished-port* *Function*

Creates a **chaos:conn** that can 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 that can be buffered. It defaults to 10. If *distinguished-port* is supplied, the local index is set to it. This is necessary for protocols that define the meanings of particular index numbers.

Opening and Closing Chaosnet Connections on the Server Side

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 **chaos: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**), the **chaos:conn** is returned immediately without waiting for an RFC to arrive.

chaos:accept *conn* *Function*

conn must be in the *RFC Received* state. An OPN packet is transmitted and *conn* enters 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* is sent and *conn* is removed.

chaos:answer-string *conn string* *Function*

conn must be in the *RFC Received* state. An ANS packet containing *string* is sent and *conn* is 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 **chaos:conn** is created.

chaos:enable-monitor-screen-server &key *(:server-enabled* **:no***)* *Function*
(:wholine-appearance **:visible***)*

Allows someone else to monitor your screen using the Monitor Screen CP command. Calling this function without any arguments disables monitoring your screen (the default). Note that, if your screen is already being monitored, the current monitoring session will *not* be terminated.

:server-enabled One of **:yes**, **:no**, or **:notify**. The default is **:no**. **:no** means no one can monitor your screen. **:yes** means anyone can monitor your screen, and you will not see a notification that they are doing so. **:notify** means that anyone can monitor your screen, but you will see a notification first when they start.

:wholine-appearance
One of **:visible** or **:invisible**. The default is **:visible**. When **:server-enabled** is **:yes** or **:notify**, **:visible** means that the wholine area will display the fact that the server is operating if someone is monitoring your

screen. **:invisible** means that the wholine area will not note whether anyone is monitoring your screen.

Note that the server side of this capability has existed for many releases, though no easily-accessible user side existed. Even then, the default for the server was to disallow access.

Site administrators, and others who build worlds, should *not* enable this facility in their world-building scripts, since allowing arbitrary monitoring of others' screens is generally considered a serious invasion of privacy. Users may wish to ensure that the worlds they run do not have this turned on by default.

This is a Chaos-only protocol, meaning that only Chaosnet users can monitor screens. This means that sites which communicate with the outside world via TCP/IP, rather than Chaos (that is essentially all sites) do not have to worry about users elsewhere on the Internet monitoring any screens at their site, regardless of whether individual users enable monitoring.

11.8.6.2. Functions for Chaosnet Connection States

The following two functions return information on the state of the Chaosnet connection (**chaos:state**), and implement a wait-or-timeout functionality (**chaos:wait**).

chaos:state *conn* *Function*

Returns the current state of the specified connection, as one of the following symbols:

chaos:inactive-state
A **chaos: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
An 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
An 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"**.

11.8.6.3. Chaosnet Stream I/O

chaos:make-stream *connection* &key (*direction* **':bidirectional**) *Function*
(*characters* **t**) (*byte-size* **nil**) (*ascii-translation* **nil**)
(*accept-p* **t**) (*token-list* **nil**)

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** returns **nil** or signals 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.
:byte-size	**8** or **16**. The default is **16**. **:byte-size** can be specified only if **:characters nil** is specified.
: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**.
:token-list	When not **nil**, this stream is a token list stream. You can operate on the stream with token list stream and BYTE-STREAM-WITH-MARK messages.

11.8.6.4. Chaosnet Packet I/O

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 **chaos:pkt** data structure. The system controls allocation of **chaos:pkt**s; each **chaos:pkt** that it gives you must be given back. There are functions to convert between **chaos:pkt**s and strings. A **chaos:pkt** is a **sys: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 **chaos: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** is an example of this. 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 **chaos:pkt** for use by the user.

chaos:return-pkt *pkt* *Function*

Deallocates a **chaos:pkt**.

chaos:send-pkt *conn pkt* &optional (*opcode* **chaos:dat-op**) *stream* *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 if *conn* is not open. **chaos:send-pkt** automatically returns the packet via **chaos:return-pkt**.

chaos:send-unc-pkt *conn pkt* &optional *pkt-number ack-number* *Function*

Transmits *pkt*, a 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 that returns **t** if there is any space in the window.

chaos:finish-conn *conn* &optional (*whostate* **"chaos finish"**) *Function*
stream

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** is returned if there are no packets available or the connection is in a bad state. Otherwise an error is signalled if the connection is in a bad state, with condition name **chaos:host-down**, **chaos:lost-received-state**, or **chaos:read-on-closed-connection**. If no packets are available and *no-hang-p* is **nil**, **chaos:get-next-pkt** waits 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 that returns **t** if any input packets are available from *conn*.

11.8.6.5. Chaosnet Connection Interrupts

chaos:interrupt-function *conn* *Function*

This attribute of a **chaos: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 **zl: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 can 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 waiting. 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 can now be transmitted with **chaos:send-pkt** without waiting. 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 want to look at the queued input packets of a connection when they get an **: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**.

11.8.6.6. Chaosnet Information and Control

chaos:host-data &optional *host* *Function*

host can be a number or a known host name, and defaults to the local host. Two values are returned: the host name and 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.

zl:hostat &rest *hosts* *Function*

Asks each of the *hosts* for its status, and prints the results. If no hosts are specified, asks all hosts on the Chaosnet. 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 Symbolics host 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.

See the section "Show Hosts Command" in *Genera Handbook*.

To abort the host status report produced by **zl:hostat** or `FUNCTION H`, press `c-ABORT`.

chaos:print-conn *conn* &optional (*verbose* **t**) *Function*

Prints everything the system knows about the connection. If *verbose* is non-**nil** it also prints everything the system knows about each queued input and output packet on the connection.

chaos:print-pkt *pkt* &optional (*verbose* **t**) (*indent* **0**) *Function*

Prints everything the system knows about the packet, except its data field. If *verbose* is **nil**, only the first line of the information is printed.

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.

neti:general-network-reset *Function*

Disables and resets the local networks as does **neti:reset**, and resets the namespace system. Resetting the namespace system clears information related to the namespace system from memory. Your host then requests any needed information from the namespace system. This cures problems that would occur if that information was somehow corrupt. (Resetting the namespace system is also done at warm and cold boot.)

After using **neti:general-network-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**.

chaos:host-up *host* &optional *timeout* *Function*

Asks a host whether or not it is up (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.

net:notify-local-lispms &optional *message* &key *(:error-p) (:report) (:output-stream)* *Function*

Sends *message* to all Symbolics machines at your site based upon information it gets from the namespace database about the Symbolics 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.

Keyword arguments are:

:error-p
Setting this keyword to **t** enables the function to report all errors encountered. Specifying **nil** (default) for this keyword enables the function to ignore all errors encountered.

:report
Setting this keyword to **t** (default) enables the function to report whether it succeeded in delivering the message. Specifying **nil** enables the function to only report failures in delivering messages.

:output-stream
Using this keyword enables you to redirect output to a specific stream.

net:notify *host* &optional *message* &key *(:error-p) (:report) (:error-stream)* *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.

Keyword arguments are:

:error-p
Setting this keyword to **t** enables the function to report all errors encountered. Specifying **nil** (default) for this keyword enables the function to ignore all errors encountered.

:report
Setting this keyword to **t** enables the function to report whether it succeeded in delivering the message. Specifying **nil** (default) enables the function to only report failures in delivering messages.

:error-stream
Using this keyword enables you to redirect error output to a specific stream.

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 **net:local-finger-location**, which is set from the local host's **finger-location** attribute in its host object. When the variable has a string value, it overrides the value in **net:local-finger-location**.

net:finger-local-lispms *Function*

Displays a list of who is using each of the Symbolics computers at the current site.

net:finger-all-lispms *Function*

Displays a list of who is using each of the Symbolics computers in the host table.

Index

* string, 86
:* keyword symbol, 86
Aborting and the Token List Stream, 173
ABORT NFILE Command, 193
:accept message, 132
:accept-p property, 143
:accept-p stream option for **net:define-server**, 105
acknowledgement, 242
Acknowledgement packet header field, 237, 242, 249
Activities That Use the Network, 14
:add-network message, 126
:add-network message to interfaces, 129
:address option for **net:define-server**, 105
:address-resolution-parameters message, 132
:allocate-packet message, 127
ANS Answer to a simple transaction packet, 245
answer to STATUS request, 254
Application-Level Chaosnet Protocols, 254
ARPA Internet references, 156
Arpanet INR/INS attention-getting feature, 256
Arpanet Name/Finger protocol, 257
Arpanet Telnet and Supdup protocols, 256
Arpanet Time protocol, 257
A Sample Host Object in the Namespace Database, 10
:ascii-translation stream option for **net:define-server**, 105
asynchronous RPC operations, 59
background process, 259
Basic Concepts of RPC, 55
bit remote type, 65
bit numbering convention, 238
Black Box , 79
boolean remote type, 65
BOOLEAN-TRUTH, 166
Bootstrapping Sync-link Gateways, 81
Bootstrapping Sync-Link Gateways with a Single Namespace, 82

Bootstrapping Sync-Link Gateways with Separate Namespaces, 82
BRD Broadcast packet, 251
bridge connection type, 239
Broadcast Sent connection state, 253
:byte-stream medium, 42, 143
:byte-stream medium type, 105
Byte Stream Conventions, 132
:byte-stream-with-mark, 163
:byte-stream-with-mark medium, 42
BYTE-STREAM-WITH-MARK Abortable States, 161
BYTE-STREAM-WITH-MARK marks, 159
BYTE-STREAM-WITH-MARK Network Medium, 159
BYTE-STREAM-WITH-MARK record format, 159
Calling the Server Function, 143
:change-of-state interrupt reason, 266
CHANGE-PROPERTIES NFILE Command, 194
channels attached to user processes, 235
:chaos medium, 42, 105
chaos: package, 259
chaos:answered-state connection state, 262
chaos:cls-received-state connection state, 262
chaos:foreign-state connection state, 262
chaos:host-down-state connection state, 262
chaos:inactive-state connection state, 262
chaos:listening-state connection state, 262
chaos:los-received-state connection state, 262
chaos:open-state connection state, 262
chaos:rfc-received-state connection state, 262
chaos:rfc-sent-state connection state, 262
chaos:server-alist, 105
chaos:accept function, 261
chaos:add-contact-name-for-protocol function, 108
chaos:answer function, 261
chaos:answer-string function, 261
chaos:assure-enabled function, 267
chaos:close-conn function, 260
chaos:connect function, 260
chaos:conn-finished-p function, 265
chaos:data-available function, 265
chaos:enable-monitor-screen-server function, 261
chaos:fast-answer-string function, 261
chaos:finish-conn function, 265
chaos:get-next-pkt function, 265

chaos:get-pkt function, 264
chaos:host-data function, 266
Chaos host number, 27
chaos:host-up function, 268
chaos:interrupt-function function, 266
chaos:listen function, 260
chaos:make-stream function, 263
chaos:may-transmit function, 265
Chaosnet, 233
Chaosnet Addresses and Indices, 235
Chaosnet bridges, 239
Chaosnet Broadcast, 251
Chaosnet Connection Closing, 251
Chaosnet Connection Establishment, 245
Chaosnet Connection Interrupts, 266
Chaosnet Connections, 234
Chaosnet Connection States, 253
Chaosnet Contact Names, 234
Chaosnet Data, 250
Chaosnet Data Formats, 238
Chaosnet End-of-Data, 250
Chaosnet File Access Protocols, 256
Chaosnet Flow and Error Control, 242
Chaosnet Information and Control, 266
Chaosnet Low-level Details, 253
Chaosnet Name Protocol, 257
Chaosnet Network Control Program, 234
Chaosnet Packet Contents, 237
Chaosnet Packet I/O, 264
Chaosnet Packet Numbers, 236
Chaosnet RFC/ANS time protocol, 102
Chaosnet Routing, 239
Chaosnet Send Protocol, 256
Chaosnet Status Packets, 249
Chaosnet Status Protocol, 254
Chaosnet Stream I/O, 263
Chaosnet Telnet and Supdup Protocols, 256
Chaosnet Time Protocol, 257
Chaosnet UNC encapsulation interface, 127
chaos:open-foreign-connection function, 260
chaos:pkt-link function, 266
chaos:pkt-nbytes function, 264
chaos:pkt-opcode function, 264
chaos:pkt-string function, 264
chaos:print-conn function, 267
chaos:print-pkt function, 267
chaos:read-pkts function, 266
chaos:reject function, 261

chaos:remove-conn function, 260
chaos:return-pkt function, 265
chaos:send-pkt function, 265
chaos:send-unc-pkt function, 265
chaos:set-pkt-string function, 264
chaos:simple function, 260
:chaos-simple medium, 42
chaos:state function, 262
Chaos subnet, 235
Chaos subnet number, 27
chaos:wait function, 263
character sets, 238, 256
Choosing a Network Addressing Scheme, 34
:class message, 88
:clear-eof operation, 263
cl-neti::link-namespaces function, 87
:close operation, 263
Closed connection state, 253
CLOSE NFILE Command, 194
closing a connection, 245, 250, 260
CLS Close connection packet, 245
Commands That Use the Network, 13
Commonly Used Arguments to Mediums, 143
Commonly Used Arguments to Servers, 143
Common Protocols, 4
COMPLETE NFILE Command, 196
Concept of Namespace Objects, 9
Concept of Network Addresses, 7
Concept of **service** Attribute, 9
Concepts of Service, Medium, and Protocol, 3
Concepts of Symbolics Networks, 1
Concepts of the Namespace System, 8
Configuring Chaos-only Sync-Link Gateways, 80
Configuring Sync-Link Gateways, 79
Configuring Sync-Link Gateways with Both Chaos and Internet, 80
conn, 259
:conn option for **net:define-server**, 105
Connecting to a Remote Host over the Network, 14
connection address in routing table, 239
connection cost in routing table, 239
connection index, 235
connection-initiation protocols, 245
connection interrupt functions, 266
Connection Keywords in the Terminal Program, 15
connection type in routing table, 239
CONTINUE NFILE Command, 197
controlled packets, 236, 242

control packets, 259
control tokens, 166
Conventions Used in NFILE Command Descriptions, 189
:Cost Keyword in the Dialnet Subnets File, 33
Count packet header field, 237
CREATE-DIRECTORY NFILE Command, 197
CREATE-LINK NFILE Command, 198
C semicolon characters, 56
DAT 16-bit Data packet, 250
DAT 8-bit Data packet, 250
database data types, 85
database deletion request, 231
data byte count, 237
Data Channel Handles and Direct File Identifiers, 191
data channel resynchronization, 186
DATA-CONNECTION NFILE Command, 199
:datagram medium, 42, 143
data packets, 259
data tokens, 166
:default-services message, 142
Defining a Network, 128
Defining a New Network Service, 97
Defining Namespace Classes, 90
delete indicator, 231
DELETE NFILE Command, 200
delivering packets, 126
Descriptions of Defined Generic Services, 48
Descriptions of Defined Mediums, 42
Design Goals of the Network System, 1
:desirability message, 141
:desirability option for **net:define-protocol**, 102
Desirability of Network Protocols, 115
Destination Address packet header field, 237
Destination Index packet header field, 237
Determining What Kinds of Connections a Remote Host Can Make, 110
Determining What Kinds of Connections a Symbolics Computer Can Make, 110
:dial medium, 42
:Dial Keyword in the Dialnet Subnets File, 32
Dial Network Medium, 159
Differences Between Local and Remote Function Calling, 54
direct access mode, 191
direct connection type, 239

:direction stream option for **net:define-server**, 105
DIRECTORY NFILE Command, 201
DIRECT-OUTPUT NFILE Command, 200
:disable message, 132
DISABLE-CAPABILITIES NFILE Command, 204
disabling a serial terminal, 22
:dna medium, 42
DNA area number, 29, 36
DNA Networks, 157
DNA node number, 29, 36
DNA Protocols Supported by Symbolics Computers as Servers, 48
DNA Protocols Supported by Symbolics Computers as Users, 48
Dynamic Window Features of The Terminal Program, 16
embedded token list, 166
empty token list, 166
:enable message, 131
ENABLE-CAPABILITIES NFILE Command, 204
Enabling and Disabling Network Services, 51
enabling a serial terminal, 22
encapsulation interface, 125
:eof operation, 263
EOF End of File packet, 250
:error-disposition option for **net:define-server**, 105
:errors suboption to **rpc:define-remote-c-program**, 62
escape characters, 22
establishing a connection, 234, 245, 260
Establishing an NFILE Control Connection, 185
Ethernet attributes, 132
Example of Defining a Server Using Network Server Code, 99
Example of Finding a Path to a Host, 112
Example of Programming with Packets, 122
Examples Using **rpc:define-remote-type**, 76
EXPUNGE NFILE Command, 204
Extensions to Lisp Syntax for RPC, 56
File Access Path, 94
FILEPOS NFILE Command, 205
Finding a Path to a Local Service, 109
Finding a Path to a Service on a Remote Host, 109
Finding a Server Description, 142
Finding the Possible Paths to a Host, 111
FINGER protocol, 257

:finish operation, 263
FINISH NFILE Command, 206
fixed bridge connection type, 239
Flavors and Messages Related to the Token List Stream, 170
Flavors Related to the Token List Data Stream, 175
flow-control, 242
:force-output operation, 263
Foreign connection state, 253
:foreign-host message, 132
foreign packet, 257
Format of Chaosnet Addresses, 27
Format of DNA Addresses, 29
Format of Internet Addresses, 27
Format of NFILE File Property/Value Pairs, 192
forwarded connection, 245
forwarding count, 237
forwarding-count field, 239
freeing packets, 126
free pool of packets, 124
Full Pathname Syntax of the Server Host, 192
FUNCTION H command, 266
Functions for Chaosnet Connection States, 262
Functions for Defining Users and Servers, 102
Functions for Invoking Network Services, 95
Functions for Service Lookup and Invocation, 139
Functions Related to Packets, 119
Functions Related to Starting Servers, 144
Functions Used in Remote Login for ASCII Terminals, 24
FWD Forward a request for connection packet, 245
gateways, 239
Generic and Specific Mediums, 41
Generic Network Services, 44
:get message, 88
global-name, 85
Glossary of Networking Terminology, 10
Hardware Requirements for Sync-Link Gateways, 79
HOME-DIRECTORY NFILE Command, 207
:host option for **net:define-server**, 105
:host option for server, 143
host address, 239
host status report, 266
How a Network Service is Performed, 91
How Domain Names Are Structured, 150
How Genera Uses Internet Domain Names, 150
How the Domain System is Structured, 149

How to Obtain an Internet Address, 34
Implementation Hint for FILEPOS NFILE Command, 206
Implementation Hints for RESYNCHRONIZE-DATA-CHANNEL NFILE Command, 224
Implementation of Network Addresses, 128
Implementation of Network Mediums, 133
Implementation of Networks, 127
Implementation of the Generic Network System, 117
Implementation of the Service Lookup Mechanism, 138
Incomplete Transmission connection state, 253
incremental indicator, 231
Incremental updates, 231
Initializing, Resetting, and Enabling Networks, 130
:input interrupt reason, 266
Interaction with Peek Network Mode, 133
Interfacing to Ethernets, 132
Interfacing to the Generic Network System, 91
Interfacing to the Lisp Machine Byte-Stream-With-Mark, 163
Internet Domain Name Namespace Attribute, 153
Internet Domain Names, 148
Internet Networks, 147
Internet protocol, 257
Introduction to BYTE-STREAM-WITH-MARK Network Medium, 159
Introduction to Chaosnet, 233
Introduction to DNA Networks, 157
Introduction to Internet Domain Names, 148
Introduction to Internet Networks, 147
Introduction to NFILE, 177, 178
Introduction to the Token List Transport Layer, 165
:invoke option for **net:define-protocol**, 102
:invoke-with-stream option for **net:define-protocol**, 102
:invoke-with-stream-and-close option for **net:define-protocol**, 102
invoking a server, 145
Invoking Mediums, 129
Invoking Network Services, 93
IP/TCP, 80
ITS host table, 239
keyword tokens, 166
list remote type, 69
list all supported servers, 144

LIST-BEGIN, 166
LIST-END, 166
Listening connection state, 253
:local medium, 42
:local network medium, 109
local networks, 130
logical end of data, 250
LOGIN NFILE Command, 208
login, remote, 21
LOGIN service, 14
LOS Lossage packet, 249
Lost connection state, 253
LSN Listen packet, 245
Macros for Defining RPC-based Programs, 57
Mapping a Chaos Address into a DNA Address, 36
Mapping an Internet Address into a Chaos Address, 35
Mapping Data Types into Token List Representation, 183
Mapping of Lisp Objects to Token List Stream Representation, 170
:medium option for **net:define-server**, 105
member remote type, 70
Messages Related to Service Lookup, 141
Messages to Namespace Names and Objects, 88
Messages to **net:object**, 88
Messages to **neti:name**, 88
Miscellaneous Notes on Interfaces, 127
MNT Maintenance packet, 253
MULTIPLE-FILE-PLISTS NFILE Command, 209
multiple paths to a service, 95
name, 85
:name message, 88
NAME protocol, 257
:names message, 89
:namespace message, 88
Namespace Database, 8
Namespace Protocols, 231
Namespace Server Access Paths, 89
Namespace Service Entry for NFILE, 178
Namespace System Functions, 86
Namespace System Lisp Data Types, 85
Namespace System Variables, 85
Namespace Timestamp Protocol, 232
net:*local-host* variable, 85
net:*local-site* variable, 85
net:*namespace* variable, 85
net:*namespace-search-list* variable, 86

net:*services-enabled* variable, 52
net:invoke-multiple-services macro, 102
net:abort-service-access-path-future function, 141
net:after-network-initialization-list variable, 131
net:continue-service-access-path-future function, 141
net:define-medium function, 133
net:define-protocol special form, 102
net:define-server function, 105
net:find-object-from-property-list function, 86
net:find-object-named function, 86
net:find-objects-from-property-list function, 86
net:find-paths-to-protocol-on-host function, 140
net:find-paths-to-service function, 96
net:find-paths-to-service-on-host function, 139
net:find-paths-to-service-using-broadcast function, 97
net:find-path-to-protocol-on-host function, 140
net:find-path-to-service-on-host function, 140
net:finger-all-lispms function, 269
net:finger-local-lispms function, 269
net:finger-location variable, 269
net:get-connection-for-service function, 104
neti:*actual-number-of-wired-packet-buffers* meter, 120
neti:*interfaces* variable, 125
neti:*invoke-service-automatic-retry* variable, 95
neti:*local-networks* variable, 131
neti:*new-services-enable* variable, 52
neti:*number-of-unwired-packet-buffers* meter, 120
neti:*servers* variable, 144
neti:*standard-services-enabled* variable, 51
neti:*target-number-of-wired-packet-buffers* variable, 120
neti:*servers* variable, 142
neti:buffered-stream-with-mark, 163
neti:funcall-server-internal-function function, 143
neti:server-protocol-name function, 142
neti:allocate-packet-buffer function, 119
neti:ask-terminal-parameters function, 24
neti:buffered-token-stream flavor, 171
neti:change-server-error-disposition function, 108
neti:deallocate-packet-buffer function, 119

neti:disable function, 267
neti:disable-serial-terminal function, 25
neti:enable function, 267
neti:enable-serial-terminal function, 24
neti:find-network-interfaces function, 126
neti:funcall-server-internal-function function, 145
neti:general-network-reset function, 267
neti:get-sub-packet function, 120
neti:get-sub-packet-maybe-copying function, 122
neti:map-packet-buffers function, 124
:neti-mark-seen, 163
neti:mark-seen flavor, 164
neti:maybe-packet-buffer-panic function, 125
neti:most-desirable-service-access-path function, 140
net:invoke-multiple-services function, 95
net:invoke-service-access-path function, 140
net:invoke-service-on-host function, 95
neti:packet-being-transmitted function, 124
neti:packet-buffer-panic function, 125
neti:raw-packet-buffer-size variable, 119
neti:recompute-all-namespace-server-access-paths function, 89
neti:recompute-namespace-server-access-paths function, 89
neti:reset function, 267
neti:server-argument-descriptions function, 145
neti:server-function function, 144
neti:server-medium-type function, 144
neti:server-number-of-arguments function, 145
neti:server-property-list function, 145
neti:server-protocol-name function, 144
neti:service-enabled-p function, 52
neti:set-terminal-parameters function, 24
neti:show-namespace-server-access-paths function, 89
neti:token-data-was-list flavor, 176
neti:token-io-unsafe flavor, 173
neti:token-list-bidirectional-data-stream flavor, 176
neti:token-list-input-data-stream flavor, 175
neti:token-list-output-data-stream flavor, 176
neti:token-list-stream flavor, 171
neti:token-stream-data-error flavor, 173
neti:with-server-error-disposition function, 108
net:network flavor, 128

net:network-type-flavor property, 128
net:notify function, 268
net:notify-local-lispms function, 268
net:parse-host function, 87
net:remote-login-on function, 24
net:service-access-path-future-connected-p function, 140
net:start-service-access-path-future function, 140
:network option for **net:define-server**, 105
:network option for server, 143
NETWORK X command, 14
Network Addressing, 27
Network Interfaces, 125
Network, Medium, and Protocol Descriptions, 147
Network Mediums, 41
Network Namespace Protocol, 231
network not in namespace database, 128
Network server code, 99
Networks Supported by Symbolics Computers, 6
Network Users and Servers, 39
NFILE abort, 186
NFILE and the token list transport layer, 183
NFILE and token list transport layer, 165
NFILE Asynchronous Errors, 227
NFILE Character Set Translation, 181
NFILE Command Descriptions, 189
NFILE Command Response Errors, 226
NFILE Commands, 193
NFILE Concepts, 178
NFILE Control and Data Connections, 183
NFILE Control Connection Resynchronization, 186
NFILE Data Connection Resynchronization, 187
NFILE data stream mode, 180
NFILE direct access mode, 180
NFILE direct file identifier, 180
NFILE DIRECTORY Data Format, 202
NFILE Error Handling, 226
NFILE File Protocol, 177
NFILE File Transfer Philosophy, 180
NFILE input data channel, 183
NFILE on Chaos, 178
NFILE on TCP, 178
NFILE OPEN Optional Keyword/Value Pairs, 214
NFILE OPEN Response Return Values, 219
NFILE output data channel, 183
NFILE Resynchronization Procedure, 186
NFILE's Chaos contact name, 185

NFILE specification, 178
NFILE's well-known TCP port, 185
NFILE Three-letter Error Codes, 228
:no-close stream option for **net:define-server**, 105
:no-eof property, 143
:no-eof stream option for **net:define-server**, 105
Notifications from the NFILE Server, 185
numeric host addresses, 235
numeric tokens, 166
object, 85
opcode pkt header field, 264
open a stream connection, 260
Open connection state, 253
Opening and Closing Chaosnet Connections, 260
Opening and Closing Chaosnet Connections on the Server Side, 260
Opening and Closing Chaosnet Connections on the User Side, 260
OPEN NFILE Command, 211
OPEN QFILE command, 211
Operation packet header field, 237
OPN Open connection packet, 245
or remote type, 68
:output interrupt reason, 266
Overview of NFILE, 178
Overview of Symbolics RPC, 53
Overview of the Chaosnet Software Protocol, 234
packet buffer panic, 124
packet forwarding, 239
packet header fields, 237, 245
Packet Number packet header field, 237
packet opcode, 237
packet Opcodes, 245
Packet Reception, 129
Packets, 117
packets with array leader, 122
Packet Transmission, 130
packet types, 245
pair, 85
:parse-address message, 129
passing packet error information, 249
:peek message, 133
:peek-header message, 133
pkt, 259, 264
ports, 257
:possible-medium-for-protocol message, 142
:possibly-qualified-string message, 88

Predefined RPC Data Types, 65
:primary-name message, 89
primary name of object, 88
probing, 242, 249
:process-name option for **net:define-server**, 105
PROPERTIES NFILE Command, 221
:property option for **net:define-protocol**, 102
:protocol-address message, 126
Protocols Supported by all Symbolics Computers as Servers, 45
Protocols Supported by all Symbolics Computers as Users, 44
protocol-translating gateway, 257
PUNCTUATION-KEYWORD, 166
PUNCTUATION-LONG-INTEGER, 166
PUNCTUATION-PAD, 166
PUNCTUATION-SHORT-INTEGER, 166
QFILE and NFILE, 177
QFILE protocol, 256
:qualified-string message, 88
Queries to network database, 231
READ NFILE Command, 223
:read-token-list message, 172
receipt, 242
:receive-packet message, 130
receiver process, 259
Recognized Keywords Denoting File Properties, 193
Recovering From a Network Problem, 17
Reference Information on NFILE, 178
References to Chaosnet Protocol Specifications, 233
References to DECnet Protocol Specifications, 158
References to Internet Domain Names documentation, 156
References to IP/TCP Protocol Specifications, 156
refusal, 245
:reject message, 132
:reject-unless-trusted option for **net:define-server**, 105
:reject-unless-trusted property, 143
remote ASCII terminal, 22, 24
remote entry, 55
remote error, 55
Remote Login, 21
remote login with machine in use, 24
remote module, 55
Remote Module Numbers, 58

remote modules, 57
Remote Terminal Commands, 14
remote type, 55
RENAME NFILE Command, 223
rendezvous subprotocols, 257
:request-array option for **net:define-server**, 105
:request-array option to **neti:funcall-server-internal-function**, 143
:request-array-end option to **neti:funcall-server-internal-function**, 143
:request-array-start option to **neti:funcall-server-internal-function**, 143
:reset message, 131
Reset Network command, 17
:response-array option for **net:define-server**, 105
:response-array option to **neti:funcall-server-internal-function**, 143
:response-array-end option to **neti:funcall-server-internal-function**, 143
:response-array-start option to **neti:funcall-server-internal-function**, 143
RESYNCHRONIZE-DATA-CHANNEL NFILE Command, 224
retransmission, 242
RFC/ANS transaction, 232
RFC Received connection state, 253
RFC Request for connection packet, 245
RFC Sent connection state, 253
routing packet, 239
routing table, 239
rpc:language, 70
rpc:original-type, 70
rpc:cardinal-16 remote type, 65
rpc:cardinal-32 remote type, 65
rpc:cardinal-4 remote type, 65
rpc:cardinal-8 remote type, 65
rpc:character-8 remote type, 65
RPC code, 99
rpc:c-string remote type, 69
rpc:define-remote-c-program macro, 62
rpc:define-remote-entry macro, 59
rpc:define-remote-error macro, 61
rpc:define-remote-error-number macro, 64
rpc:define-remote-module macro, 57
rpc:define-remote-type macro, 70
rpc:disable-rpc-trace function, 78
rpc:enable-rpc-trace function, 78
rpc:enumeration remote type, 65

RPCError macro, 59
rpc:integer-16 remote type, 65
rpc:integer-32 remote type, 65
rpc:integer-8 remote type, 65
rpc:opaque-bytes remote type, 69
rpc:pascal-string remote type, 69
rpc:rpc-error macro, 62
rpc:rpc-values macro, 62
rpc:show-rpc-trace function, 77
rpc:spread-vector remote type, 68
RPCValues macro, 59
rpc:write-c-token-list, 56
RS-232, 79
RUT packet, 239
RUT Routing Information packet, 253
SELECT T key, 14
sending a CLS packet, 250
Sending a Packet to an Interface, 126
Sending message to all Symbolics machines at site, 268
:send-mark, 163
SEND protocol, 256
:send-token-list message, 171
server functions, 260
Server functions for datagram protocols, 143
server process, 234
Service Access Path, 93
Service Attributes in the Namespace Database, 40
set, 85
Set Remote Terminal Options Command, 23
Setting the Chaosnet Address, 7
si:*user* variable, 85
si:get-site-option function, 87
simple transaction, 245
single-float remote type, 65
Size in bytes of packet buffer, 119
SNS packets, 242
SNS Sense status packet, 249
Software Interface to the Namespace System, 85
Source Address packet header field, 237
Source Index packet header field, 237
special characters, 22
Standard Communication with Interfaces, 125
Starting Network Servers, 142
Starting to Use NFILE, 178
status packets, 242
STATUS protocol, 251, 254
stream, 259

:stream option for **net:define-server**, 105
stream connection, 245
:stream-options property, 143
:string message, 88
string remote type, 68
structure remote type, 67
STS packets, 242
STS Status packet, 249
stub function, 53
subnet, 239
:Subnet Keyword in the Dialnet Subnets File, 31
Subpackets and Coercing Packets, 120
Summary of Functions for Defining Users and Servers, 101
Summary of Functions for Service Lookup and Invocation, 138
Summary of the Internet Domain Names Facility, 155
Supdup, 21
Supdup program, 14
Supdup protocol, 256
:supports-broadcast message, 141
symbolic host names, 235
Symbolics Computers as Central Name Resolvers, 152
Symbolics Computers as Name Servers, 153
Symbolics Generic Network System, 39
Symbolics Implementation of Chaosnet, 259
Symbolics RPC and Sun RPC, 54
Symbolics RPC Facilities in Lisp, 56
Sync-Link Gateways, 79
sys:disable-services function, 51
sys:disable-services property, 52
sys:enable-services function, 51
sys:enable-services property, 52
:tcp medium, 42
TCP and UDP Protocols Supported by SUN Computers as Servers, 47
TCP and UDP Protocols Supported by Symbolics Computers as Servers, 47
TCP and UDP Protocols Supported by Symbolics Computers as Users, 46
Technical Details of the Chaosnet Software Protocol, 245
Telnet, 21
Telnet program, 14
Telnet protocol, 256
Terminal, 21

The Dialnet Subnets File, 30
The Domain System and the Namespace System, 154
The Packet Pool, 117
The Remote Login Capability for ASCII Terminals, 21
The Remote Procedure Call Facility, 53
The RPC Data Representation Layer, 64
The RPC Data Type Extension Language, 70
TIME protocol, 251, 257
timestamp, 231
timestamp indicator, 231
token, 85
Token List Data Stream, 174
token list data stream, 165
Token List Stream, 166
token list stream, 165
Token List Stream Example, 168
Token List Transport Layer, 165
TOP-LEVEL-LIST-BEGIN, 166
TOP-LEVEL-LIST-END, 166
top-level token list, 166
Tracing RPC Transactions, 77
Transmission Control Protocol, 257
transmit list, 124
:transmit-packet message, 127
transmitting interactive messages, 256
triple, 85
:trusted-p option for **net:define-server**, 105
:trusted-p option for server, 143
type, 70
:type message, 128
Types of Tokens and Token Lists, 166
:udp medium, 42
uncontrolled packets, 236, 242
UNC Uncontrolled Data packet, 250, 257
UNDATA-CONNECTION NFILE Command, 225
Universal Time format, 183
:unparse-address message, 129
unsafe data channel, 186
update-by attribute, 231
updates to network database, 231
User Datagram Protocol, 257
:user-get message, 89
user process, 234
Using Foreign Protocols in Chaosnet, 257
Using Peek to Get Information on Networks, 16
Using the Network, 13

Using the Remote Login Facilities for ASCII Terminals, 22
Using the Terminal Program, 14
V.35, 79
vector remote type, 68
What is a File Server?, 3
What is a host?, 2
What is a Network?, 2
What is a Network Service?, 2
:who-line option for **net:define-server**, 105
wildcard, 86
window into the set of packet numbers, 242
wired and unwired packets, 117
zl:hostat function, 266