EXPLORER NETWORKING REFERENCE

MANUAL REVISION HISTORY

Explorer Networking Reference (2243206-0001*B)

Original Issue .... July 1985

Revision A .... March 1986

Revision B .... June 1987

The system-defined windows shown in this manual are examples of the software as this manual goes into production. Later changes in the software may cause the windows on your system to be different from those in the manual.

Texas Instruments Incorporated
ATTN: Data Systems Group, M/S 2151
P.O. Box 2909
Austin, Texas 78769-2909

THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Little/No Interest ░ Medium Interest ▥ Required ▓

	First Day of Explorer Use	Casual or New Developer	Experienced Developer	Applications Programmer	Systems Manager
Introduction to the Explorer System	▓	▓	░	░	▥
Zmacs Editor Tutorial	▓	▥	░	░	░
Master Index	▥	▓	▓	▓	▥
Lisp Reference	▥	▓	▓	▓	▥
Input/Output Reference	▥	▓	▓	▓	▥
Tools and Utilities	▥	▓	▥	▓	▓
Zmacs Editor Reference	░	▓	▥	▥	░
Window System Reference	░	▥	▓	▓	░
Programming Concepts	▥	▓	▥	▥	▥
Networking Reference	▥	▥	▥	▥	▓
Glossary	▥	▥	░	░	░
System Software Installation	▥	░	░	░	▓
Technical Summary	▥	░	░	░	▥
System Software Design Notes	░	░	▥	▥	░

THE EXPLORER™ SYSTEM SOFTWARE MANUALS

Mastering the Explorer Environment	Explorer Technical Summary	2243189-0001
	Introduction to the Explorer System	2243190-0001
	Explorer Zmacs Editor Tutorial	2243191-0001
	Explorer Glossary	2243134-0001
	Explorer Networking Reference	2243206-0001
	Explorer Diagnostics	2533554-0001
	Explorer Master Index to Software Manuals	2243198-0001
	Explorer System Software Installation Guide	2243205-0001
Programming With the Explorer	Explorer Programming Concepts	2549830-0001
	Explorer Lisp Reference	2243201-0001
	Explorer Input/Output Reference	2549281-0001
	Explorer Zmacs Editor Reference	2243192-0001
	Explorer Tools and Utilities	2549831-0001
	Explorer Window System Reference	2243200-0001
Explorer Options	Explorer Natural Language Menu System User's Guide	2243202-0001
	Explorer Relational Table Management System User's Guide	2243203-0001
	Explorer Grasper User's Guide	2243135-0001
	Explorer Prolog User's Guide	2537248-0001
	Programming in Prolog, by Clocksin and Mellish	2537157-0001
	Explorer Color Graphics User's Guide	2537157-0001
	Explorer TCP/IP User's Guide	2537150-0001
	Explorer LX™ User's Guide	2537225-0001
	Explorer LX System Installation	2537227-0001
	Explorer NFS™ User's Guide	2546890-0001
	Explorer DECnet™ User's Guide	2537223-0001
	Personal Consultant™ Plus Explorer	2537259-0001
System Software Internals	Explorer System Software Design Notes	2243208-0001
	Release Information, Explorer System Software	2549844-0001

THE EXPLORER™ SYSTEM HARDWARE MANUALS

System Level Publications	Explorer 7-Slot System Installation	2243140-0001
	Explorer System Field Maintenance	2243141-0001
	Explorer System Field Maintenance Documentation Kit	2243222-0001
	Explorer System Field Maintenance Supplement	2537183-0001
	Explorer System Field Maintenance Supplement Documentation Kit	2549278-0001
	Explorer NuBus™ System Architecture General Description	2537171-0001
System Enclosure Equipment Publications	Explorer 7-Slot System Enclosure General Description	2243143-0001
	Explorer Memory General Description (8-megabytes)	2533592-0001
	Explorer 32-Megabyte Memory General Description	2537185-0001
	Explorer Processor General Description	2243144-0001
	68020-Based Processor General Description	2537240-0001
	Explorer II Processor and Auxiliary Processor Options General Description	2537187-0001
	Explorer System Interface General Description	2243145-0001
	Explorer NuBus Peripheral Interface General Description (NUPI board)	2243146-0001
Display Terminal Publications	Explorer Display Unit General Description	2243151-0001
	CRT Data Display Service Manual, Panasonic code number FTD85055057C	2537139-0001
	Model 924 Video Display Terminal User's Guide	2544365-0001
143-Megabyte Disk/Tape Enclosure Publications	Explorer Mass Storage Enclosure General Description	2243148-0001
	Explorer Winchester Disk Formatter (ADAPTEC) Supplement to Explorer Mass Storage Enclosure General Description	2243149-0001
	Explorer Winchester Disk Drive (Maxtor) Supplement to Explorer Mass Storage Enclosure General Description	2243150-0001
	Explorer Cartridge Tape Drive (Cipher) Supplement to Explorer Mass Storage Enclosure General Description	2243166-0001
	Explorer Cable Interconnect Board (2236120-0001) Supplement to Explorer Mass Storage Enclosure General Description	2243177-0001
143-Megabyte Disk Drive Vendor Publications	XT–1000 Service Manual, 5 1/4-inch Fixed Disk Drive, Maxtor Corporation, part number 20005 (5 1/4-inch Winchester disk drive, 112 megabytes)	2249999-0001
	ACB–5500 Winchester Disk Controller User's Manual, Adaptec, Inc., (formatter for the 5 1/4-inch Winchester disk drive)	2249933-0001

1/4-Inch Tape Drive Vendor Publications	Series 540 Cartridge Tape Drive Product Description, Cipher Data Products, Inc., Bulletin Number 01-311-0284-1K (1/4-inch tape drive)	2249997-0001
	MT01 Tape Controller Technical Manual, Emulex Corporation, part number MT0151001 (formatter for the 1/4-inch tape drive)	2243182-0001
182-Megabyte Disk/Tape Enclosure MSU II Publications	Mass Storage Unit (MSU II) General Description	2537197-0001
182-Megabyte Disk Drive Vendor Publications	Control Data® WREN™ III Disk Drive OEM Manual, part number 77738216, Magnetic Peripherals, Inc., a Control Data Company	2546867-0001
515-Megabyte Mass Storage Subsystem Publications	SMD/515-Megabyte Mass Storage Subsystem General Description (includes SMD/SCSI controller and 515-megabyte disk drive enclosure)	2537244-0001
515-Megabyte Disk Drive Vendor Publications	515-Megabyte Disk Drive Documentation Master Kit (Volumes 1, 2, and 3), Control Data Corporation	2246129-0002
	Volume 1, General Description, Operation, Installation and Checkout, and Part Data	2246125-0004
	Volume 2, Theory, General Maintenance, Trouble Analysis, Electrical Checks, and Repair Information	2246125-0005
	Volume 3, Diagrams	2246125-0006
1/2-Inch Tape Drive Publications	MT3201 1/2-Inch Tape Drive General Description	2537246-0001
1/2-Inch Tape Drive Vendor Publications	Cipher CacheTape® Documentation Manual Kit (Volumes 1 and 2 With SCSI Addendum and, Logic Diagram), Cipher Data products	2246130-0001
	1/2-Inch Tape Drive Operation and Maintenance (Volume 1), Cipher Data Products	2246126-0001
	1/2-Inch Tape Drive Theory of Operation (Volume 2), Cipher Data Products	2246126-0002
	SCSI Addendum With Logic Diagram, Cipher Data Products	2246126-0003

Control Data is a registered trademark of Control Data Corporation.

WREN is a trademark of Control Data Corporation.

CacheTape is a registered trademark of Cipher Data Products, Inc.

Printer Publications	Model 810 Printer Installation and Operation Manual	2311356-9701
	Omni 800™ Electronic Data Terminals Maintenance Manual for Model 810 Printers	0994386-9701
	Model 850 RO Printer User's Manual	2219890-0001
	Model 850 RO Printer Maintenance Manual	2219896-0001
	Model 850 XL Printer User's Manual	2243250-0001
	Model 850 XL Printer Quick Reference Guide	2243249-0001
	Model 855 Printer Operator's Manual	2225911-0001
	Model 855 Printer Technical Reference Manual	2232822-0001
	Model 855 Printer Maintenance Manual	2225914-0001
	Model 860 XL Printer User's Manual	2239401-0001
	Model 860 XL Printer Maintenance Manual	2239427-0001
	Model 860 Xl Printer Quick Reference Guide	2239402-0001
	Model 860/859 Printer Technical Reference Manual	2239407-0001
	Model 865 Printer Operator's Manual	2239405-0001
	Model 865 Printer Maintenance Manual	2239428-0001
	Model 880 Printer User's Manual	2222627-0001
	Model 880 Printer Maintenance Manual	2222628-0001
	OmniLaser™ 2015 Page Printer Operator's Manual	2539178-0001
	OmniLaser 2015 Page Printer Technical Reference	2539179-0001
	OmniLaser 2015 Page Printer Maintenance Manual	2539180-0001
	OmniLaser 2108 Page Printer Operator's Manual	2539348-0001
	OmniLaser 2108 Page Printer Technical Reference	2539349-0001
	OmniLaser 2108 Page Printer Maintenance Manual	2539350-0001
	OmniLaser 2115 Page Printer Operator's Manual	2539344-0001
	OmniLaser 2115 Page Printer Technical Reference	2539345-0001
	OmniLaser 2115 Page Printer Maintenance Manual	2539356-0001

Communications Publications	990 Family Communications Systems Field Reference	2276579-9701
	EI990 Ethernet® Interface Installation and Operation	2234392-9701
	Explorer NuBus Ethernet Controller General Description	2243161-0001
	Communications Carrier Board and Options General Description	2537242-0001

Omni 800 is a trademark of Texas Instruments Incorporated.

OmniLaser is a trademark of Texas Instruments Incorporated.

Ethernet is a registered trademark of Xerox Corporation.

CONTENTS

Paragraph Title Page

About This Manual

1 Networking Concepts

Appendix Title Page

Figures

Tables

ABOUT THIS MANUAL

Introduction

The *Explorer Networking Reference* manual helps you to understand the various networking facilities that are available with the Explorer system. This manual will teach you how to do the following:

- Understand the basic concepts of networking
- Use networking applications
- Set up an Explorer system as part of a network or as a standalone computer
- Create and maintain a network namespace
- Check on network status and reset the network

Contents of This Manual

This manual includes an index and a glossary and is organized into the following sections.

Section 1: Networking Concepts — Describes the terms and concepts associated with networks, such as protocols, layers, and configuration.

Section 2: Networking Protocols — Briefly describes networking protocols that are available with the Explorer system. These include DECnet, Transmission Control Protocol/Internet Protocol (TCP/IP), Chaosnet, and the Systems Network Architecture (SNA).

Section 3: Network Applications — Describes a number of network applications that are available with the Explorer system. Some of these include:

- Telnet - Allows you to use its window as a terminal to another host.
- VT100™ emulator - Allows you to use the Explorer monitor and keyboard as a VT100 terminal.
- Converse - Allows you to conduct a dialogue with another user.
- Finger - Allows you to see who is logged in at the machines in your network.
- Time - Allows a host to ask the time of day.
- Eval - Allows you to set up a read-eval-print loop on a remote host.
- Remote disk server and band transfers - Allows you to read and write to disks on remote hosts.

VT100 is trademark of Digital Equipment Corporation.

Section 4: Getting On the Network — Describes how to create a new network, update a Release 2 network configuration to a Release 3 network namespace, add an Explorer host to an existing network, and so on.

Section 5: Chaosnet Applications Programming and Networking — Describes how you can use Chaosnet to move data over a network that includes Explorer systems. This section is provided for the applications developer who has a need for the network communication services that Chaosnet offers. It assumes the developer is familiar with the Chaosnet protocol.

Section 6: The Generic Network System — Describes how to make applications work across different protocols. This section includes the Generic Network Interface (GNI) and the Generic Services Interface (GSI).

Section 7: Network Status and Troubleshooting — Discusses how to use the Network and Host-status items of the Peek utility to check on network status. Also describes the commonly used functions for checking network status and resetting the network.

Appendix A: External Data Representation — Describes the Explorer system implementation of Sun Microsystems™ External Data Representation protocol, which allows different types of machines to exchange operands across a network.

Appendix B: Remote Procedure Call — Describes the Explorer system implementation of Sun Microsystem's Remote Procedure Call protocol, which allows different types of machines to interact with each other on a procedure level.

Appendix C: Writing RPC Servers — Describes how to write servers on the Explorer system, using the RPC protocol.

Sun Microsystems is a trademark of Sun Microsystems, Incorporation.

Keystroke Sequences

Many of the commands used with the Explorer are executed by a combination or sequence of keystrokes. In this manual, hyphens connect the names of keys that you should press simultaneously (*chord*). Spaces separate the names of keys that you should press one after the other. The following table illustrates this manual's conventions for describing keystroke sequences.

Keystroke Sequence	Interpretation
META-CTRL-D	Hold the META and CTRL keys while pressing the D key.
SYSTEM HELP	Press and release the SYSTEM key, then press and release the HELP key.
CTRL-X CTRL-F	Hold the CTRL key and press the X key, release the X key, and then press the F key. Alternatively, press CTRL-X, release both keys, and press CTRL-F.
META-X-Find File RETURN	Hold the META key while pressing the key, release the keys, type the letters `Find File` and then press the RETURN key.
TERM–SUPER-HELP	Press the TERM key and release it, press the minus key (–) and release it, then press and hold the SUPER key while pressing the HELP key.

Mouse Clicks

The optical mouse features three buttons that enable you to execute operations from the mouse without returning your hand to the keyboard. Pressing and releasing a button is called *clicking*. The following table illustrates the abbreviations used online to describe clicking the mouse.

Abbreviation	Action
L	Click the left button (press the left button once and release it).
M	Click the middle button (press the middle button once and release it).
R	Click the right button (press the middle button once and release it).
L2 M2 R2	Click the specified button twice quickly. (Press the button, release it, then press it again quickly.) This is called double clicking*.
LHOLD MHOLD RHOLD	Press the specified button and hold it down.

NOTE:

* If you double click too fast, the system sees only one click, If you double click too slowly, the system sees two single clicks. You can use alternative method to prevent this: press and hold the CTRL key while you click the specified button one time.

Lisp Code

Three fonts are used in this manual to denote Lisp code:

- System-defined words and symbols are in **boldface**. System-defined words and symbols include names of functions, variables, macros, flavors, methods, and so on-any word or symbol that appears in the system source code.

- Examples of programs and output are in a special `monowidth` font. System-defined words in an example are also in this font.

- Sample names are in *italics*. Names in italics can be replaced by any value you choose to substitute. (Italics are also used for emphasis and to introduce new terms.)

For example, this sentence contains the word **setf** in boldface because **setf** is defined by the system.

Some function and method names are very long—for example, **get-ucode-version-of-band**. Within the text, long function names may be split over two lines because of typographical constraints. When you code the function name **get-ucode-version-of-band**, however, you should not split it or include any spaces within it.

Within manual text, each example of actual Lisp code is shown in the monowidth font. For instance:

```
(setf x 1 y 2) => 2
(+ x y) => 3
```

> The form `(setf x 1 y 2)` sets the variables `x` and `y` to integer values; then the form `(+ x y)` adds them together.

In this example of Lisp code with its explanation, `setf` appears in the monowidth font because it is part of a specific example.

For more information about Lisp syntax descriptions, see the *Explorer Lisp Reference* manual.

Words for which you can substitute another value are shown in italics, as in the following example:

> The variables *vars* contained in the lambda list of some function *foo* are bound to the argument values of the function invocation.

Occasionally, in examples where you could substitute a specific value, the boldface and italics fonts are used together.

1

NETWORKING CONCEPTS

Introduction

1.1 This section explains the fundamental concepts of networking. It emphasizes the way messages are passed around a network by means of hardware and software protocols that allow Explorer systems and various other computer systems to converse. This section provides a conceptual framework for the later sections on network configuration, network applications programming, and the creation of servers and protocols. If you are familiar with the principles of networking, you can skip this section.

A computer network is a group of computers that can communicate with each other, sharing resources such as programs, data, memory, and peripheral devices, which includes printers, mass storage devices, and often even processors as well.

A network can provide your system with the following benefits:

- Reduced costs in handling tasks
- Decreased transaction time
- Connection with other networks through a gateway
- Electronic mail transaction
- Rapid data transfer between different computers
- Remote login capability, making your workstation a virtual terminal on another system
- Use of a remote Explorer system or other computer in the net as a coprocessor to speed complex computations
- Easy installation, future expansion, and reconfiguration

There are two basic types of networks. A *long haul network* is a group of computers that communicate via satellite or telephone communications link and are typically separated by long distances. However, this manual does not concern itself with long haul networks.

The other type of network is the *local area network* (LAN). The LAN (on which this manual focuses) is a group of computers that are usually connected by a high bandwidth cable within a building or campus.

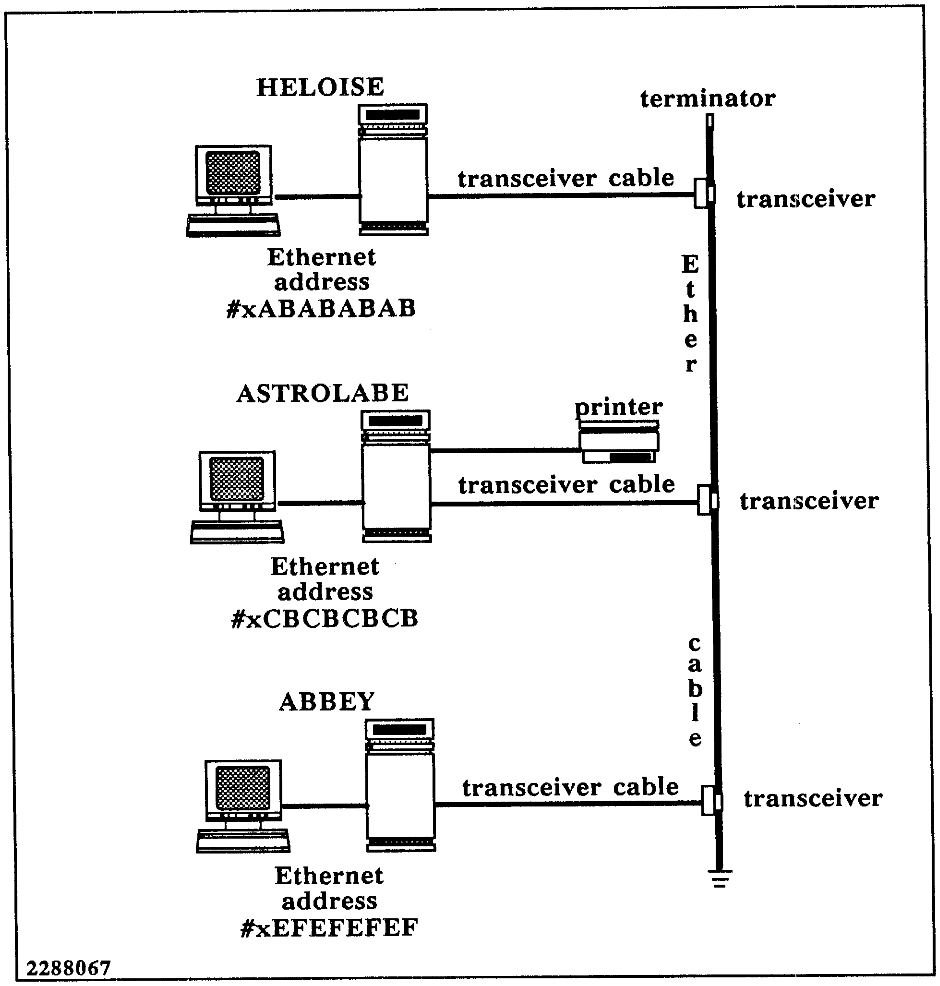
A LAN is a communications link between computers and peripherals such as printers or mass storage devices. The word *local* implies a distance of 5 to 1000 meters between computers. The upper distance limit is determined by the electrical characteristics of the transmission cable and interface devices.

A LAN provides a reliable, high-speed means of information exchange and resource sharing. LANs also expand the possibilities for communications between devices of different vendors and between local and remote networks.

Figure 1-1 shows an example of a simple network whose *site name* is Fleet. The backbone of this network is the Ethernet® cable. Ethernet, a local area network specification, forms the backbone of Explorer networks. The Ethernet specifications prescribe standards that allow various operating systems, communications devices, and computer hardware to operate in a network.

Transceivers are attached at prescribed intervals along the Ethernet cable. A transceiver cable between the transceiver and each Explorer system's NuBus Ethernet controller board provides the link to the LAN. One of the Explorer systems in this network (ASTROLABE) has a printer, which serves as the printer for the entire network.

Figure 1-1 **A Simple Network — Site: FLEET**



Ethernet is a registered trademark of Xerox Corporation.

All of the devices in this network communicate by passing digital information to each other over the Ethernet cable. At the most basic level, information is passed from one machine to another by rapidly changing the electrical state of the network.

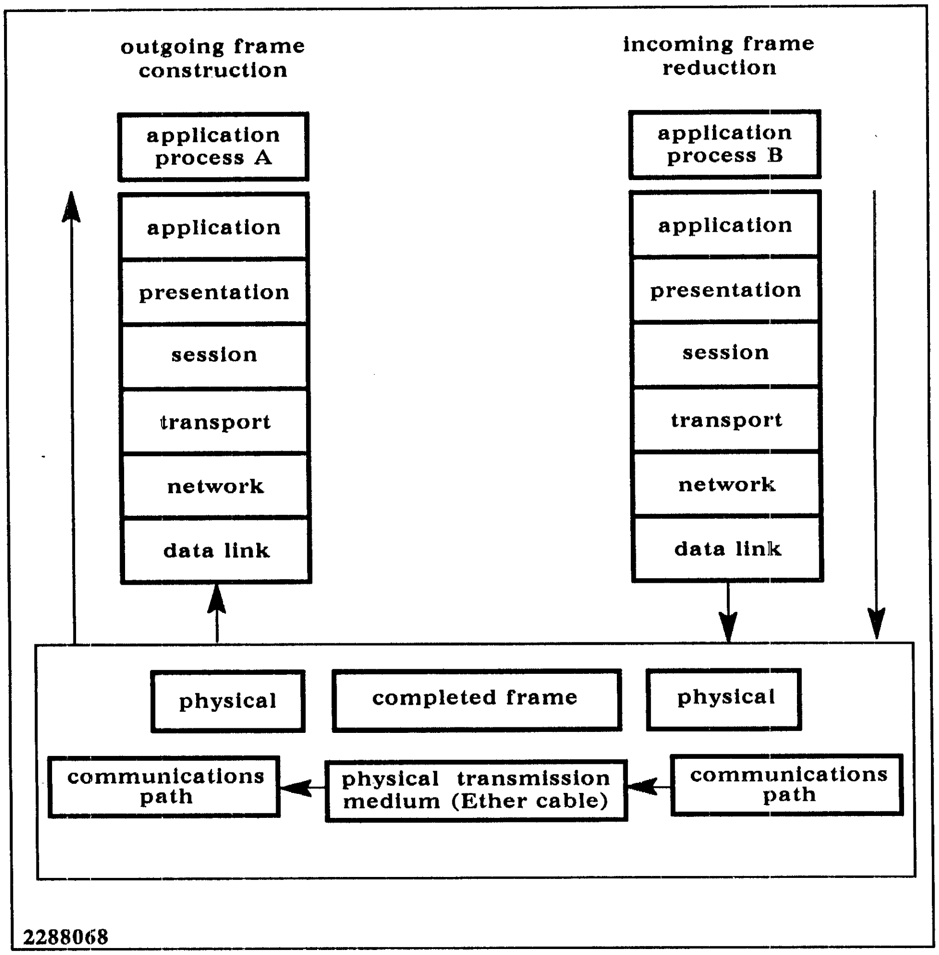
Protocols and the ISO/OSI Model

1.2 Initially, there is no way for one Lisp machine to interpret the rapidly fluctuating changes in the Ethernet's electrical potential. Before meaning can be associated with these changes in the state of the network, systematic rules for interpretation must be imposed in the hardware, in the software, or in both. To assure successful communication, every Explorer system on the network must operate under the same set of rules. The rules that govern the proper composition and interpretation of signals on the network are called *protocols*.

Protocols exist for many different levels of a network. For example, some protocols govern physical transmission of data, some govern interaction with user processes, and others govern areas that fall between these two examples.

The International Standards Organization (ISO) has established a comprehensive formal hierarchical model that arranges different network functionalities into seven levels or *layers*. This model, known as the ISO *Model for Open Systems Interconnection* (OSI), is shown in Figure 1-2. The discussion of networking that follows refers continuously to Figure 1-2 and the small network of Explorer systems shown in Figure 1-1.

Figure 1-2 ISO Reference Model for Open Systems Interconnection



The ISO/OSI model assumes that data is transmitted in *frames*. Each frame consists of various types of information, such as the address of the target computer, the type of the frame, and the actual data. You will see shortly that Ethernet and some other higher-level protocols refer to these frames as *packets*.

Each successively higher level in the OSI model is embedded in its parent level immediately below. A frame at the transport level is embedded within the data segment of a network-layer frame. A user at a given layer on one side of a communications session converses directly with a user or process at the corresponding layer at the other side of the communications session, as indicated by the horizontal arrows in Figure 1-2.

Lower levels are invisible to higher levels. For instance, a user at ABBEY in Figure 1-1 might use an application-layer process to transmit a message to a user at HELOISE. To both users, the process consists of a direct transfer of information from one to the other. They do not concern themselves with the fact that various types of header information are appended to the data by the protocols at each successive layer.

Theoretically, this structured arrangement allows each layer to be implemented in such a way that each successive layer need only respond to the requirements of the previous lower layer. The idea is to provide for modularity of the functions of the network. In practice, however, the various layers are not always clearly demarcated as in the OSI model. Still, most network protocols, including Ethernet, at the lowest levels adhere reasonably well to the proposed ISO/OSI standard. The following paragraphs discuss each of these layers and its implementation and function in Explorer networks.

Lower-Level Protocols

1.3 The two lower-level *protocols* of the OSI model are actually the physical layer and the data link layer.

Physical Layer

1.3.1 The physical layer is the lowest level in the ISO/OSI model. This layer specifies the most essential physical network characteristics controlling how data is encoded, transmitted, received, and then decoded. For Ethernet, the physical-layer protocol includes specifications such as the following:

- Cable lengths — An Ethernet segment can be 500 meters long.
- Distance between hosts — The minimum distance between hosts is 2.5 meters. Marks on the cable every 2.5 meters indicate points at which transceivers may be attached.
- Data rate — Specified as 10 million bits per second on Ethernet.
- Timing — Ethernet uses asynchronous timing.
- Electrical characteristics — Ethernet employs a 50-ohm shielded baseband coaxial cable that is grounded at one end and has a 50-ohm terminator at the other.

Data Link Layer

1.3.2 The second level in the ISO/OSI model, the data link level, is the uppermost Ethernet layer. This level provides the procedural and functional means to establish, maintain, and release data link connections among network entities and to transfer data frames. This layer also detects and may correct errors that might occur in the physical layer. The link level has two major functions in an Ethernet network: *data encapsulation* and *link management*.

Data encapsulation/decapsulation includes the following processes:

- Framing — Defines how a message begins and ends (frame boundary delimitation)
- Addressing — Handles source and destination addresses
- Error detection — Detects physical channel transmission errors

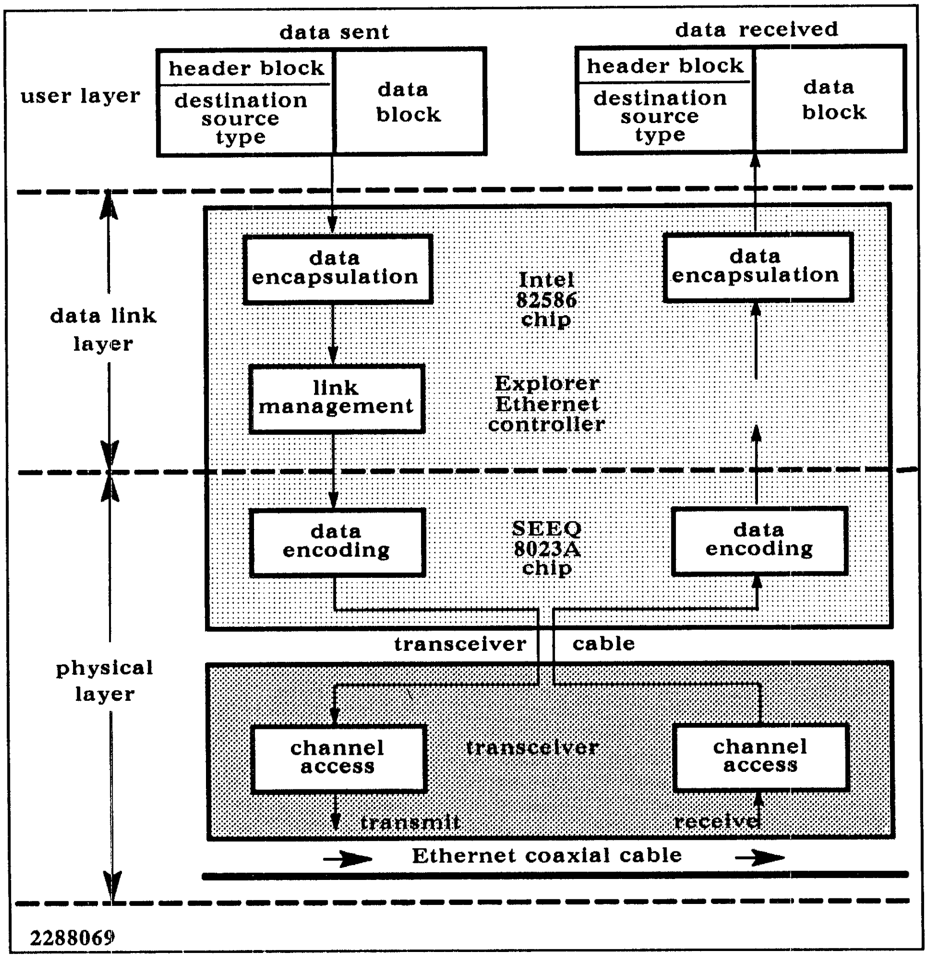
Link management includes two very important functions:

- Channel allocation — Assures collision avoidance
- Contention resolution — Controls collision handling

Data Transfer in Lower-Level Protocols

1.4 In an Explorer network, the functions at the physical and data link layers are carried out by the Explorer Ethernet controller board, the transceiver and transceiver cable, and the Ethernet coaxial cable. The relationship between these hardware components and the ISO/OSI link and physical layers is shown in Figure 1-3. The sequence of events during data transmission flows as shown by the arrow pointing downward on the left side of the diagram. The flow of events during data reception follows the path shown by the arrow pointing upward on the right side of the diagram.

Figure 1-3 **Ethernet Layers**



To understand the ways in which the lower levels of the ISO/OSI model interact, it is best to consider their function during the transmission and reception of data over a network. Transmission is a top-down process: the higher-level protocol passes data to the next lower level, which then passes it down to the next layer, and so on. Eventually, the data link layer receives the data and hands it to the physical layer, which is responsible for the actual electrical transmission of the data from one side of the session to the other. Reception is the reverse of the transmission process. Since the data link layer is the first (or last) layer in the chain of successive encapsulations and manipulations of data, you should first understand how this layer functions and how it is related to the physical layer.

Assume that a user or user process at HELOISE, on the network illustrated in Figure 1-1, finds it necessary to send data to ABBEY. To transmit this data, the user at HELOISE sends two objects to the Ethernet controller board: a *header block* and a *data block.*

The header block contains three pieces of information that allow the receiving host to decode, interpret, and respond to the transmitting host's data:

- Destination Address — This indicates the host, in this case ABBEY (whose Ethernet address is #xEFEFEFEF), to which the data or message is to be transmitted.
- Source Address — This is optionally passed by the user to the Ethernet controller board, depending upon the particular higher-level software being used. The Ethernet controller board knows its own Ethernet address, which is stored in ROM. HELOISE includes the Ethernet number #xABABABAB.
- Type — This is a number that indicates to the receiving host (ABBEY) the type of data being transmitted. The user's software uses this number to determine which higher-level protocol to follow to interpret the data.

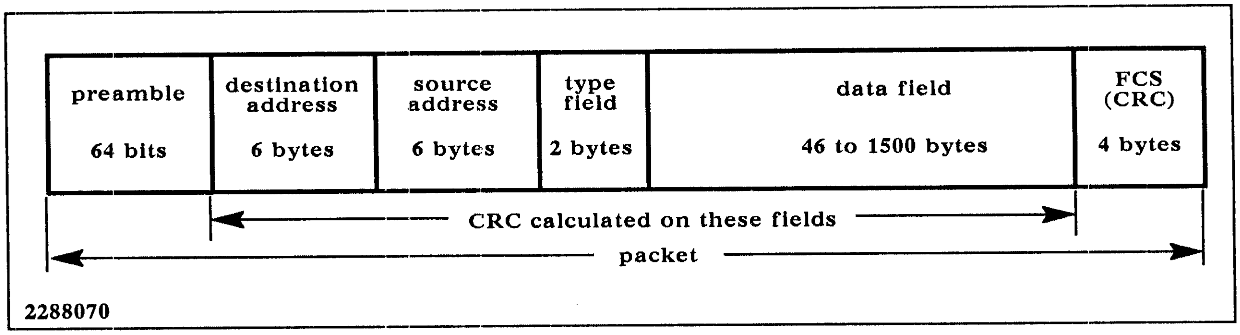
The data block contains the actual data or message the user wishes to transmit to the receiving host.

Building the Ethernet Packet

1.4.1 Before HELOISE can transmit these two blocks of information over the Ethernet network illustrated in Figure 1-1, they must first be put into an appropriate format. When HELOISE sends the header block and data block to the Ethernet controller board for transmission, the controller board first encapsulates them in a *packet,* the step indicated in the transmission procedure diagrammed in Figure 1-3. The data encapsulation process constructs the packet, performs a cyclic redundancy calculation, and appends the result of this calculation to the frame check field of the packet.

Figure 1-4 shows the format of an Ethernet packet. A packet must be at least 72 bytes long and at most 1526 bytes long, where a byte is eight bits. All of the fields in a packet are fixed-length except for the *data field,* which can vary in length from 46 to 1500 bytes. The data field contains the user's original data.

Figure 1-4 Ethernet Packet Format



Destination Address

1.4.1.1 The first item appended to the packet by the encapsulation process is the 48-bit station address of the host to which the packet is being transmitted. The Ethernet controller board gets this number from the block of header data.

At higher layers in the ISO/OSI model, the destination address is usually invisible to the user. It is either hidden behind the host name(s) or behind the names of servers that carry out functions the receiving host makes available to other Explorer systems on the network. In FLEET, the sample site in Figure 1-1, the host name HELOISE is associated with the Ethernet address #xABABABAB, and ABBEY is associated with the Ethernet address #xEFEFEFEF. The Ethernet address of ASTROLABE is #xCBCBCBCB. ASTROLABE provides printing services to the network. A request to print a document by any Explorer system in this network (including ASTROLABE) might result in an automatic passing of ASTROLABE's Ethernet address as the destination address to the Ethernet controller board, depending on the higher-level software in use, since all machines on the network could know the address of the machine providing the printer service.

The *Address Resolution Protocol* (ARP) passes the Ethernet address of the target host, along with the type flag identifying the network-layer protocol currently employed, to the Ethernet controller board. The operation of this protocol is discussed in detail in paragraph 1.6.2, Translating Higher-Level Protocol Addresses.

HELOISE might find it necessary to send a broadcast message to both ASTROLABE and ABBEY. The need to broadcast a message to more than one host simultaneously arises often in certain mail systems in which a user at one host wishes to send a bulletin or announcement to everyone else at the site. Multicast addresses are also important in the functioning of ARP discussed later and in the distribution of general networking information throughout a network by the Network Namespace server (discussed in Section 4, Getting On the Network). Each Explorer system in a network may have one or more multicast addresses. Multicast addresses usually correspond to groups of logically related hosts in a network. Multicast addresses are not a concern now, because they are essentially treated in the same manner as the station destination and source addresses.

Source Address

1.4.1.2 The source address is the address of the Explorer system that is sending the data. It indicates the sender's address to the receiver. The user does not have to pass this number to the Ethernet controller; this address is stored in ROM on each host's Ethernet controller board. The controller gets this address from ROM and appends it to the packet.

Type Field 1.4.1.3 The type field is a two-byte field that identifies the way the receiving host is to interpret incoming data. This number must be passed by the sender, along with the destination address and the data, to the Ethernet Controller board. Often the type field is used to indicate the higher-level protocol used in the data field. The data field for Internet Protocol (IP) is always #x0800, that of Chaosnet is #x0804, and that of ARP is always #x0806. Thus, the type field is a link to the next higher-level protocol (OSI level 3) above the Ethernet data link level. In most higher-level protocols, the type is specified automatically and is completely invisible to the user.

Data Field 1.4.1.4 After the information from the user's block of header information has been included in the packet, the user's data block can be added. Often, higher-level software protocols break the user's data into a group of data blocks to be passed to the Ethernet controller board. Each block is then passed to the receiving station in a series of packets.

Frame Check Sequence Field 1.4.1.5 At this point, the destination address, source address, and type of packet have been appended to the user's data. The Ethernet controller board now does a cyclic redundancy calculation on the destination address, source address type, and data fields as one data unit. The result of this calculation is appended to the end of the packet. The receiving Explorer system uses this frame check sequence for error detection and correction. The packet is now passed on to the sender's Ethernet *link management* level.

Link Management

1.4.2 The link management sublayer monitors the activity of the network to determine when to send the packet. An Ethernet network is a distributed network that has no central master; all hosts on the network are equal. All of these hosts need time on the Ethernet, so there must be a mechanism for allocating access time.

CSMA/CD 1.4.2.1 The Ethernet network uses a probabilistic access scheme called *carrier sense multiple-access with collision detection* (CSMA/CD). At any given time, there is a probability that the Ethernet is free. This probability is a function of variables, such as overall network length, packet length, and level of overall bus activity.

Traffic Detection 1.4.2.2 A host must listen before it transmits data. If the sending host detects activity on the network, it defers to the passing traffic and waits a brief time before attempting to retransmit the packet. After at least 9.6 microseconds (the minimum time allowed between packet transmissions), the host can attempt to transmit again. When it cannot detect a signal on the network, the packet is passed to the *data-encoding* level of the Ethernet controller.

Data Encoding

1.4.3 If no collision is detected, the data link mechanisms of the Ethernet controller board send the packet to be encoded. Encoding is a function of the ISO/OSI physical layer and involves two processes:

- Construction and transmission of the packet preamble
- Conversion of the binary-encoded packet into a *Manchester* phase-encoded format

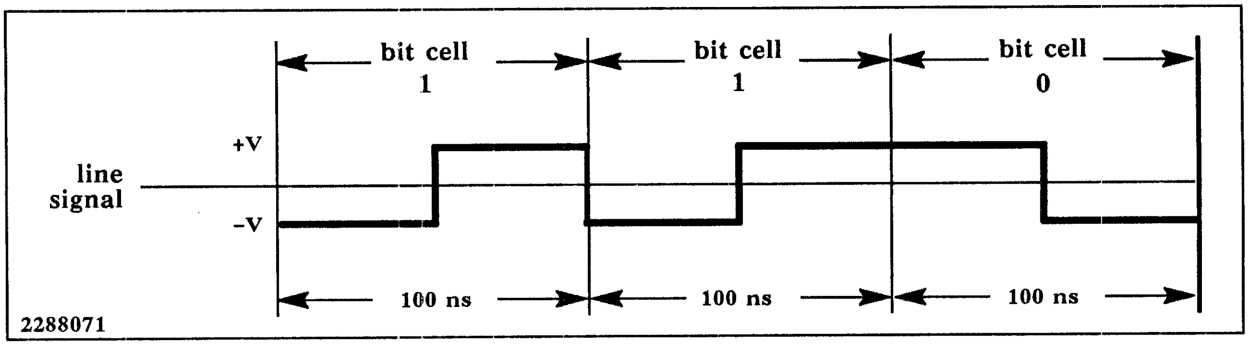
Preamble

1.4.3.1 The *preamble* is a 62-bit sequence of 10s, terminated by the 2-bit sequence 11. The first 62 bits allow all of the Explorer systems on the network to set their timing for reception of the transmitted packet. The last 2 bits mark the end of the preamble and the beginning of the data.

Manchester Encoding

1.4.3.2 To transmit a binary-encoded packet, the data must first be translated into a phase-encoded form. The Manchester-encoding format used by Ethernet is illustrated in Figure 1-5. In this type of encoding, a transition occurs in the center of each bit cell. The first half of the bit cell contains the complement of the value of the bit; the second half of the bit cell contains the true value of the bit. In Figure 1-5, the first half of the first (leftmost) bit cell has a negative, or 0, voltage level. In the middle of the bit cell, the voltage changes to a positive, or 1, level, which represents the true value of the bit cell. A 0 bit cell (the rightmost bit cell in Figure 1-5) begins with a positive, or 1, component that shifts to a negative, or 0, component halfway through the bit cell. Each bit cell is 100 nanoseconds(ns) long, resulting in a burst data rate of 10 million bits per second. For various reasons to be discussed shortly, the actual data transfer rate is slower than this theoretically attainable rate.

Figure 1-5 Manchester-Encoded Data Format



Packet Transmission

1.4.4 After the *preamble* has been constructed and sent, the Manchester-encoded packet is sent over the transceiver cable to the transceiver, which handles channel access. This step involves the actual generation of electrical signals as well as the detection of *collisions* on the Ethernet.

Collisions

1.4.4.1 Because of propagation delays and probability, two or more stations can start to transmit at the same time on an apparently free bus. The result is a collision, in which data from both stations appears on the bus. Receiving stations have no way to distinguish between the bit streams, so the data from the transmitting stations is corrupted.

Collision Fragments

1.4.4.2 A *collision fragment* is the partially transmitted packet left as a result of a halt in transmission after a collision is detected. Collision fragments are always short because they occur only when two (or more) transmitters try to access a previously free Ethernet. Receiving stations can detect and reject collision fragments because they are shorter than the minimum allowable Ethernet packet size of 64 bytes.

Jams

1.4.4.3 The transmitting transceiver detects collisions by listening to the network. The transceiver assumes that transmissions on the line that are not in exact phase with the station transmit clock are from other stations. If the transceiver detects a collision on transmission, it sends a signal back to link management, which then sends a 4-byte to 6-byte sequence called a *jam*. This sequence notifies all hosts attempting to send data that a collision has occurred. Each transmitting host then waits a random length of time before again attempting to transmit.

Data Reception

1.4.5 If no collision is detected, the preamble and packet are injected onto the Ethernet cable. Each host's transceiver (the channel access layer) senses activity on the network the moment the preamble is sent so that all of the hosts on the network begin to listen. All listening transceivers now perform three operations:

- Turn on a carrier sense signal for use by the data link layer
- Synchronize with the incoming bit stream of 10s in the preamble
- Send the incoming bits up to the next level (on the Ethernet controller board) to be decoded

Decoding

1.4.6 The decoder (on the Ethernet controller board) now carries out the following three operations:

- Translates the Manchester-encoded data back into binary data
- Discards the preamble
- Sends the data to the data link layer

Link Level

1.4.7 The link management level of the Ethernet controller board of each host on the network has already been notified that a signal is present on the Ethernet. When the signal sense detector goes off in a given host (that is, there is no more activity on the network), the link manager assumes that the packet is complete and sends the incoming data to the data decapsulator.

Decapsulation

1.4.8 The decapsulator then does a cyclic redundancy calculation on the packet; it compares the results of this check with the checksum it finds in the frame check sequence field of the received packet. If the two numbers do not match, the packet is rejected.

If the packet passes the cyclic redundancy test, the decapsulator examines the destination address field. If it matches the Ethernet address of the receiving host (or is a broadcast address), then the data is passed up to the user. If the destination address does not match the host's destination address, then the data is rejected, and the packet is not passed to the user.

Note that all of the hosts on the network carry out the processes up to the point of examination of the destination address. Only the host having a matching Ethernet address passes the data up to the user. All hosts sharing a particular multicast address will accept the packet. The transmission of data at the lowest levels of the ISO/OSI model has now been completed. The original data or message transmitted by the user of the transmitting Explorer system has been sent to the user or process in the receiving Explorer system. Note that all of the necessary functions at this level have been carried out by the Explorer Ethernet controller boards in both communicating hosts, the transceiver cables, and the transceivers of the transmitting and receiving Explorer workstations and the Ethernet coaxial cable.

Higher-Level Protocols

1.5 While the physical and link layers of the ISO/OSI model are implemented by the Explorer Ethernet hardware, the higher-level protocols of the ISO/OSI model are implemented as Explorer Lisp functions and programs. As noted previously, a frame, or packet, at each level in the ISO/OSI model is embedded within the data field of the previous lower level.

For instance, in the Explorer implementation, a *Transmission Control Protocol* (TCP) segment at the transport layer is embedded in the data field of an IP datagram at the network layer, which is encapsulated in turn within the data field of an Ethernet packet for transmission. Some protocol implementations can be quite clearly associated with one ISO/OSI layer. IP, at the network layer, is one such protocol. Protocols such as Chaosnet (which has functions at the network, transport, and session layers) and TCP (at the transport/session layer) span ISO/OSI layers. In such cases, it is not always possible to determine the boundaries between layers. The present discussion is restricted to Chaosnet. For further discussion of IP and other protocols, see the *Explorer TCP/IP User's Guide.*

Network-Layer Protocols

1.5.1 Explorer networks use the Chaosnet and IP network-layer protocols, as well as some others. Network-layer protocols have the following responsibilities:

- The establishment of connections between hosts
- The transfer of data from one host to another
- The termination of connections at the end of a communication session

Network-layer protocols control the flow of information through multi-LAN systems. They are critical in determining routing through networks that utilize bridges, gateways, and dial-up applications (bridges and gateways are discussed later in this section).

Transport-Layer Protocols

1.5.2 The transport layer of the ISO/OSI model assures that data is transferred from one party to the other with no losses or duplications. It also handles issues such as cost effectiveness and reliability of data transmission. This layer is defined to fulfill the following functions:

- Segmentation and blocking — Breaks long data segments into segments that can fit into the data field of the packet.
- Sequencing — Assures that data is reassembled in the correct order at the time of reception.
- Error detection and monitoring — Assures that data is received correctly.

The Chaosnet functions that are used for reading and writing to Chaosnet streams belong to the transport layer of the ISO/OSI model. These functions allow user processes to write indefinitely large amounts of data to a stream that is read by a foreign process at the other end of the connection. The details of breaking the data into packet-sized chunks for encapsulation into packets is invisible to the user. For details on the use of Chaosnet streams, see Section 5, Chaosnet Applications Programming and Networking.

Session-Layer Protocols

1.5.3 The session layer has also not been completely specified. Its purpose is generally considered to fulfill the following functions:

- Quarantine service — Allows the sender to request that data transmission be delayed until an acceptable quantity of data has accumulated.
- Interaction management — Determines whether the interaction between hosts is one-way, two-way simultaneous, or two-way alternate.
- Expedited and normal data exchange — Provides for methods of establishing data traffic priorities.
- Error recovery and exception reporting — Assures that certain transactions are never aborted halfway through the transmission.

Presentation-Layer Protocols

1.5.4 Presentation-layer protocols provide the means to establish, manage, and terminate a connection between communicating processes. Among the functions fulfilled are the following:

- Code and character set translation
- Formatting and data compression
- Syntax resolution
- Data encryption

The presentation layer is principally responsible for assuring that all parties interpret data in the same way.

Application-Layer Protocols

1.5.5 Application-layer protocols have also been only partially specified. This layer provides the sole means for a user or the user's application to access the lower layers of the ISO/OSI model. Its purpose is generally considered to fulfill the following functions:

- Identification of intended communications partners
- Determination of the current availability of the intended communications partners
- Establishment of the authority to communicate
- Agreement on responsibility for error recovery
- Agreement on the procedures to be used for the control of data integrity

The protocols at all layers below the application layer are completely transparent to the user.

Examples of application-layer utilities in the Explorer environment include Mail, VT100 emulation, the TIME protocol, and Converse, as well as various file transfer services.

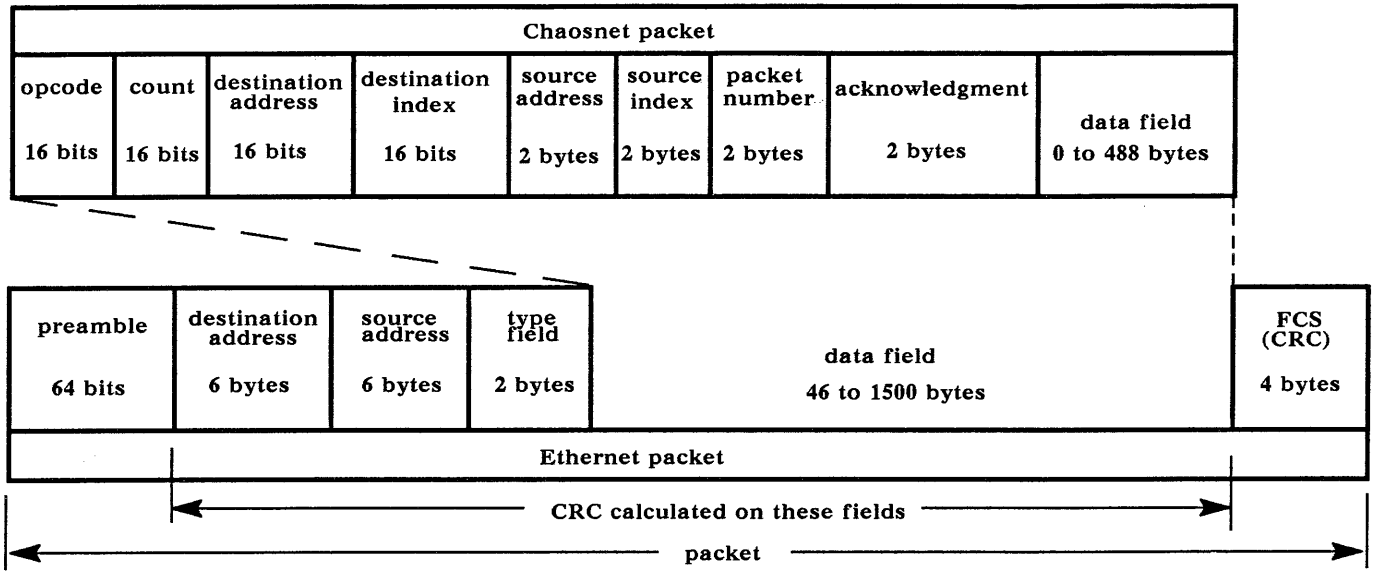
Data Transfer in Higher-Level Protocols

1.6 The Chaosnet protocol can serve as an example of a network-layer protocol. For specific information on TCP/IP and DECnet, refer to the *Explorer TCP/IP User's Guide* and the *Explorer DECnet User's Guide*.

Building the Chaosnet Packet

1.6.1 Like lower-level protocols, Chaosnet, at the network (and higher) layers, bundles data into packets for transmission. A Chaosnet packet includes header information and data, as well as a checksum for the verification of packet transmission reliability. Since Chaosnet is (basically) a network-layer protocol, it relies upon the protocol at the next lower layer, the data link layer, for actual transmission of the packet over the network. Ethernet is the next lower level. When Chaosnet presents a packet, Ethernet encloses it within an Ethernet packet for transmission. A Chaosnet packet is always entirely embedded within the *data field* of the Ethernet packet within which it is transmitted. Figure 1-6 shows the Chaosnet packet format and the way it is related to the Ethernet packet that carries it.

Figure 1-6 Chaosnet Packet



2288072

Packet Headers

1.6.1.1 At every level of the ISO/OSI model (except the physical layer), a frame, or packet, consists of a header and a data field. Header information is used to establish peer-to-peer communications sessions. The header may also contain information used to activate services and functions provided by lower levels. Remember, at the data link layer, an Ethernet packet header contains the destination address, source address, and type field number. At the network level, a Chaosnet packet has a header that provides eight types of information necessary for the routing of packets through the network.

- Operation code (*opcode*) — A 16-bit number that tells how to interpret the packet. For example, opcode #x01 indicates that the packet is a request for connection.
- Count — A 16-bit number indicating not only the number of times a message has been forwarded across bridges in the network but also the length of the data field of the packet.
- Destination subnetwork address — A 16-bit number indicating the network on which the destination host is found.
- Destination host address — A 16-bit number giving the address of the host to whom the data is directed.
- Source subnet address — A 16-bit number indicating the subnetwork on which the host sending the data is found.
- Source host address — A 16-bit number identifying the host that is sending the data.
- Packet number — A 16-bit number indicating the order in which a controlled packet has been transmitted. Controlled packets undergo various error-checking and error-correcting procedures.

Chaosnet Data

1.6.1.2 It is possible for the data field of a Chaosnet datagram to be empty or to be very small. This is always the case for certain types of packets, as indicated by the packet's opcode. An EOF packet, for example, always has an empty data field. Such packets are only 16 bytes long. This *runt* packet is too small: the Ethernet data field must be at least 46 bytes long. The lowest level Chaosnet software automatically *pads* the data field of a runt packet so that it is at least 46 bytes in length. This padded data (which is found to be in excess of the length of the data field provided by the count field) is stripped off the Chaosnet packet after the Ethernet controller board passes the packet to the receiver's network level process.

Translating Higher-Level Protocol Addresses

1.6.2 When a host passes a Chaosnet or other higher-level packet to Ethernet for encapsulation and transmission, there is no information directly available by which Ethernet's data link layer can determine either the type field of the Ethernet packet, which tells the destination host which higher level protocol is being used, or the Ethernet address of the destination host.

ARP is used to translate higher-level protocol addresses (such as 16-bit Chaosnet addresses or 32-bit Internet addresses) into 48-bit Ethernet hardware addresses. ARP also determines the Ethernet type field of the packet, which indicates to the receiving host how to interpret the packet. ARP also maintains and updates a table of known high-level addresses and their corresponding Ethernet addresses.

ARP works roughly as follows. A user at CERBERUS boots the system. Subsequently, a user process at CERBERUS, which has the Chaosnet address #xFABA, builds an Ethernet packet that it wishes to transmit to a user process at CHARON, which is at Chaosnet address #xFACA. As usual, the packet's opcode and count fields are filled in, as are the destination address and index, the source address and index, and so on. This packet now passes to the ARP.

Address Resolution Table

1.6.2.1 Now, ARP must determine the Ethernet address of the host, CHARON, which is to receive the message. ARP looks in its table of known hosts for a Chaosnet address matching that specified in the packet's destination address and index fields. Since the host has just booted, however, there are no entries in the table. Every time an Explorer system is powered down or rebooted, the entries in this table go away. Therefore, ARP must get the Ethernet address of the target host by other means.

Address Resolution Packets

1.6.2.2 At this point, CHARON discards the Chaosnet packet passed to ARP by the user process and creates an *address resolution* packet. This packet has the following information in its data field:

- Hardware type — Always specified as Ethernet
- Ethernet type — The type of the higher-level protocol, in this case Chaosnet
- Length of hardware address — The length in bytes of the hardware address
- Length of protocol address — The length of the Chaosnet address (in this case)
- Address resolution opcode — Either ***ar-request*** or ***ar-reply***
- Hardware (Ethernet address) of the sender — Self-explanatory
- Protocol address of sender — The Chaosnet address of the sender
- Hardware (Ethernet) address of target host — If known
- Protocol address of the target host — The Chaosnet address of the target host

There are two types of address resolution packets:

- Request — This type of address resolution packet is sent when there is not an entry for the target host in the address resolution table.
- Reply — This type of address resolution packet is sent in response to a request packet.

Request Packets Since CHARON cannot find any reference to the Chaosnet address of the target host CERBERUS, CHARON broadcasts a request packet to every host on the network. A broadcast packet has the broadcast address as the destination address of the Ethernet packet. Therefore, the exact address of the destination address of the original packet is unnecessary. Each item in the data field in the packet is specified as above, except for CERBERUS' Ethernet address. The address resolution opcode is now set to ***ar-request***.

Reply Packets When CERBERUS receives the broadcast request packet, it decapsulates the packet and passes the data field of the Ethernet packet to ARP. ARP first determines that CERBERUS has the same hardware type as that specified in the request packet and that it knows the network-level protocol being used (Chaosnet in this case). It then enters CHARON's protocol type (Chaosnet), Chaosnet address, and corresponding Ethernet address in CERBERUS' address resolution table. If this information is already on the table, it is overwritten by the new table entries.

Next, CERBERUS sees the address resolution opcode and notes that it is dealing with a request packet. Now, it puts its own Ethernet address in the sender's Ethernet address field (in the data field of the packet), sets the address resolution opcode of the packet to ***ar-reply***, and sends the packet directly back to CHARON. The packet is not broadcasted. Note that at this point CERBERUS knows how to access CHARON, but CHARON still does not know how to reach CERBERUS directly.

Now, CHARON receives the reply packet from CERBERUS. CHARON enters CERBERUS' protocol type (Chaosnet) and Chaosnet address and their mapping to CERBERUS' Ethernet address. CHARON notes that the packet is an address resolution reply and throws it away. Since the original Chaosnet packet still has not been transmitted, the higher-level protocol can now attempt retransmission. The Chaosnet protocol now passes a copy of the original Chaosnet packet to ARP. ARP then finds CERBERUS' addressing information on the address resolution table and passes the protocol type (Chaosnet) and target Ethernet address to the data link level for encapsulation and subsequent transmission.

From now on, CERBERUS and CHARON have all of the information about each other that is necessary to transfer packets back and forth. The address resolution table at each host on the network is built up incrementally as a host attempts to transmit a Chaosnet packet to a remote host. Once address resolution information about a host is available, the local host does not have to broadcast a request packet on subsequent attempts to reach that host.

Advantages of ARP This incremental method of address resolution has several advantages:

- A host needs to keep information only about those hosts with which it communicates frequently.

- An address resolution packet can be reused for sending a reply back to the originating host.

- Maintaining a (relatively) small database of protocol-type/address to Ethernet address mappings saves the overhead that would result from a large number of hosts constantly broadcasting request packets over the network.

Servers

1.7 At the network layer and higher, processes on different hosts pass data back and forth between themselves. On the Explorer system, these processes consists of two sorts of Lisp functions:

- *Server Functions*
- *Client Functions*

Server Function

1.7.1 A server function is a Lisp function that provides a service to other hosts on the network.

Client Function

1.7.2 The client function invokes a server on a remote host and interprets the data received from the server.

Network Configurations

1.8 The preceding discussion has focused on the simple network of three Explorer systems shown in Figure 1-1. Often, however, the network is far more complicated, involving multiple Ethernet cables with connections of various types to non-Explorer computer systems. A simple example of such an extended network is shown in Figure 1-7.

Ethernet Segments

1.8.1 The simplest Ethernet networks are single-segment networks such as the three Explorer networks shown in Figure 1-1. An Ethernet segment is usually a single length of Ethernet coaxial cable. A segment can consist of several shorter cables linked with coaxial cable connectors and barrel adapters or transceiver connectors. In any case, the Ethernet cable segment cannot exceed 500 meters in length. This restriction theoretically allows as many as 100 transceivers to be installed on a single segment.

Multiport boxes allow additional Explorer workstations to be attached to a single transceiver. In Figure 1-7, TEOTL, OMETOCHTLI, and TLALOC are connected to the Ethernet via a multiport box.

NOTE: Ethernet networks can branch, but they cannot form loops. This is in direct contrast to token-passing networks, which must form loops.

Several single-segment networks such as that in Figure 1-1 can be connected together to form a single large network. This is done in several different ways, depending upon the distance between hosts, the architectural layout of the site, and so on.

When two Ethernet segments are located within 100 meters of each other, they can be connected by a *local repeater*. The repeater attaches to the Ethernet coaxial cable with a transceiver and transceiver cable. In Figure 1-7, a repeater connects two segments in Tsuris Hall. Note that *both* of these segments are on the *same* network, #xFA.

When longer distances, such as may occur between buildings at a site, are required between Ethernet segments, two *half-repeaters* must be used. Each half-repeater is connected to its Ethernet cable by a transceiver and transceiver cable. The connection between the two half-repeaters is usually a duplex fiber-optic cable. This arrangement allows two subnets to be as far as 1000 meters apart. Two half-repeaters connect the #xFA subnetwork segments in Tsuris Hall with the #xFA segment in Toad Hall. Note that the #xFA segment in Toad Hall is part of the *same* network as the #xFA segments in Tsuris Hall, even though they are in different buildings, possibly separated by a long distance.

Connecting Ethernet Networks

1.8.2 It is sometimes necessary or desirable to interconnect two or more Ethernet networks. This is normally done in one of two ways: an Explorer system or another computer on one network serves either as a *bridge* or as a *gateway* to the other network.

Bridges

1.8.2.1 A *bridge* is any host that is connected to two subnetworks, both of which operate under the same set of protocols. The purpose of the bridge is to *forward* packets from a host on one subnetwork to a host on another subnetwork. The bridge connects to its two subnetworks via *two* Ethernet controller boards and transceiver cables. In Figure 1-7, FRIAR-LARRY is a bridge between network #xFA and network #xBE; ARION is the bridge between network #xBE and #xCB. Other bridges connect various other networks in Figure 1-7.

Gateways

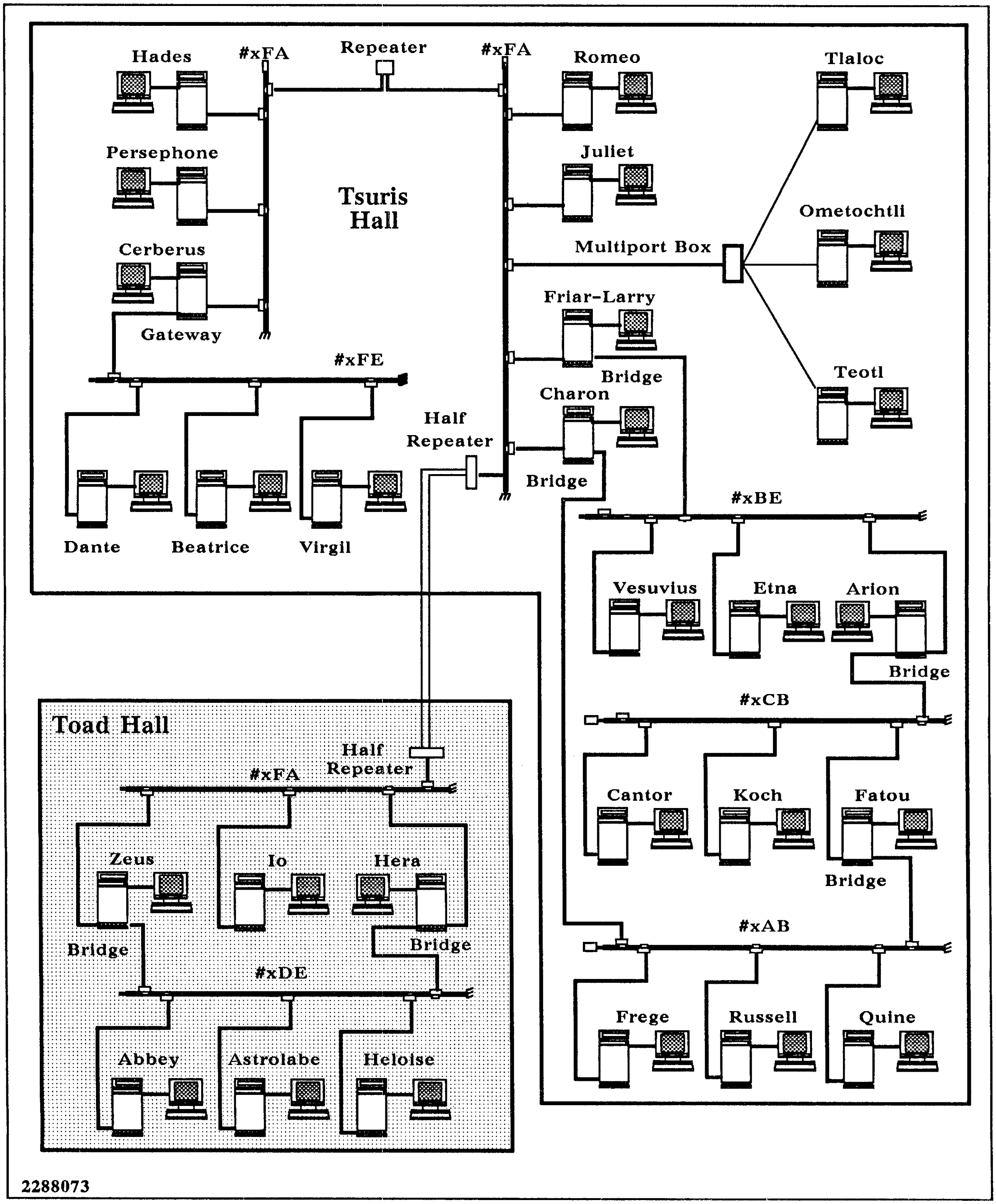
1.8.2.2 A *gateway* interconnects two networks that obey different protocols. It must understand and translate the information passing through it. Since a gateway may interconnect networks operating at different speeds, gateways must also handle data flow and error control. In Figure 1-7, CERBERUS on Ethernet #xFA serves as a gateway to the alien network #xFE, which operates under different hardware and software protocols. CERBERUS serves as a translator between these two subnetworks.

Packet Routing

1.8.3 *Routing* determines the specific way that a packet is delivered from one subnetwork to another subnetwork as specified by the packet's destination address. In the simplest case, routing is trivial. One host sends a packet to another host on the same network. The destination address is the same as the source address.

For instance, in Figure 1-7, PERSEPHONE and HADES are both on subnetwork #xFA. However, it is more complicated for PERSEPHONE on network #xFA to send a packet to QUINE, on network #xAB. The packet must pass through at least one bridge (CHARON, between network #xFA and network #xAB) and at most three bridges (FRIAR-LARRY, between #xFA and #xBE; ARION, between #xBE and #xCB; and FATOU, between #xCB and #xAB). Between PERSEPHONE and KOCH, there are also two possible routes, both of which require passing packets through two bridges. The network-layer and transport-layer protocols of the ISO/OSI model determine which route to take in complicated situations such as these. Chaosnet routing is similar in many respects to that implemented in the TCP/IP protocols and exemplifies many of the principles and problems involved.

Figure 1-7 A More Complicated Network



Host Routing Table

1.8.3.1 Each host in a network has a *routing table*. This table provides the following information:

- The subnetwork on which each host resides
- The type of connection between a host and a subnetwork (direct or bridge)
- The Chaosnet address of each host on the same network
- The *cost* of sending a packet through a given route

RUT Packet

1.8.3.2 Bridges send *routing* (RUT) packets to each subnet to which they are directly connected at 15-second intervals. The route packet contains the address of each subnetwork to which it is connected, either directly or indirectly, and the cost of sending data to that subnetwork. Each host on the subnetwork uses the RUT packet to update the information in its routing table.

Packet Transmission

1.8.3.3 Now, assume that a user at PERSEPHONE wishes to transmit a packet to a user or application at QUINE. PERSEPHONE first looks on her routing table to find the best route by which to pass the data through the network. PERSEPHONE finds two routes:

- Route A — PERSEPHONE via network #xFA to bridge CHARON,
  CHARON via network #xAB to QUINE.
- Route B — PERSEPHONE via network #xFA to bridge FRIAR-LARRY,
  FRIAR-LARRY via network #xBE to bridge ARION,
  ARION via network #xCB to bridge FATOU,
  FATOU via network #xAB to QUINE.

Cost PERSEPHONE chooses the route which *costs* least. The cost of a route is generally determined by *adding* the cost of each network gateway through which the packet must pass. This results in fewer hops across bridges and eliminates loops. PERSEPHONE chooses Route A. The cost of a given subnetwork is increased by one unit every four seconds, until the cost reaches a maximum value. This maximum cost is retained until the next RUT packet forces it to be reset to a lower value.

Transmission Once PERSEPHONE has found an efficient route to QUINE, she sends the packet with the Ethernet (hardware) destination address of the first bridge through which the message must pass. In this case, that bridge is CHARON. CHARON accepts the message, looks at the software packet header information, and sees that the final destination address is #xAB. It now retransmits the entire packet that was sent from PERSEPHONE, header and all. CHARON uses its own Ethernet address as the source address and QUINE's Ethernet address as the destination address. Thus, the software source and destination addresses are used to indicate the *final* destination of the data to the various bridges. Each bridge in turn decides the best way to forward the packet through the network.

NETWORKING PROTOCOLS 2

Introduction

2.1 This section introduces the networking protocols standardly available, or as options on the Explorer system. These include the following:

- Chaosnet
- Transmission Control Protocol/Internet Protocol (TCP/IP)
- DECnet

Chaosnet

2.2 Chaosnet is a family of protocols that provide the basic networking services for Explorer systems. The main purpose of Chaosnet is to allow high speed communication between processes on different machines, and to ensure that transmission error do not go undetected.

Chaosnet actually spans the network and transport layers of the ISO/OSI network reference model. In so doing, it provides simple datagram services as well as transmission services that provide extensive error-checking and resolution.

For information about how to program applications in the Chaosnet environment, see Section 5, Chaosnet Applications Programming and Networking, of this manual.

For specifications on Chaosnet, see David A. Moon's *Chaosnet*, A.I. Memo No. 628, M.I.T. Artificial Intelligence Laboratory, June 1981.

TCP/IP

2.3 TCP/IP is a family of protocols that provide various services in the networking environment.

TCP resides in the fourth ISO/OSI layer (the transport layer) and provides reliable transmission service between user processes—not just between machines.

IP provides the datagram service residing in the ISO/OSI network layer of the implementation. Note that IP is used not only for local area networking, but also as a means of connecting Explorer hosts to gateways that connect, in turn, to wide area networks such as ARPANET. Explorer systems cannot be connected directly to wide area networks; they must attach to a gateway.

The User Datagram Protocol (UDP) is another protocol provided in the TCP/IP family. UDP provides quick and easy data transfers. In the transport layer, UDP provides a datagram service from one process to another with minimal connection procedures. It is transaction-oriented and appropriate for request-response interactions rather than for streams of data.

Two application level programs are included with the TCP/IP software that interface to TCP. These are Telnet (including the VT100™ emulator) and the File Transfer Protocol (FTP). The Trivial File Transfer Protocol (TFTP) is also available for interfacing to UDP.

On an Explorer system, you can interface to TCP/IP in two different ways. The first is through explicit menus. Menus are used both by FTP and TFTP for transferring files between your local host and a remote host that also has a TCP/IP implementation. Most implementations of TCP/IP allow this first method of interfacing. However, Explorer TCP/IP provides the additional capability of interfacing directly with the Explorer file system, via Dired and other file system utilities.

For more information on TCP/IP, see the *Explorer TCP/IP User's Guide*.

DECnet

2.4 DECnet is a set of programs and protocols for use on Digital Equipment Corporation's (DEC™) computer systems. DECnet's architecture is called Digital Network Architecture (DNA).

The intention of DECnet is to provide network communications between DEC equipment. There are many isolated DECnets in the world. DECnet also differs from ARPANET in that there is no distinction between hosts and IMPs. A DECnet is just a collection of machines (nodes) some of which may run user programs, some of this may do packet switching, and some of which may do both. The functions performed by a given machine may even change with time.

DECnet has five layers:

- Application — A mixture of the ISO/OSI presentation and application layers
- Network services — Corresponds to the ISO/OSI transport layer
- Transport — Corresponds to the ISO/OSI network layer
- Data link control — Corresponds to the ISO/OSI data link layer
- Physical — Corresponds to the ISO/OSI physical layer

DECnet's physical layer can handle most kinds of lines available.

For more information on DECnet, see the *Explorer DECnet User's Guide*.

VT100 is a trademark of Digital Equipment Corporation.

DEC is a trademark of Digital Equipment Corporation.

3 NETWORK APPLICATIONS

Introduction

3.1 This section describes several of the main network applications available on the Explorer system.

- File Servers
- Telnet
- VT100 emulator
- Converse
- Name
- Time
- Eval-Serving
- Finger
- Remote disk server and band transfers
- Sending and printing notifications

File Servers

3.2 Files on remote hosts are accessed using file servers over a network. Normally, connections to servers are established automatically when you try to use them, but there are a few ways you can interact with them specifically.

When characters are written to a file server computer that normally uses the ASCII character set to store text, Explorer characters are mapped into an encoding that is reasonably close to an ASCII transliteration of the text. When a file is written, the characters are converted into this encoding; the inverse transformation is performed when a file is read back. No information is lost. Note that the length of a file, in characters, is not the same measured in original Explorer characters as it is measured in the encoded ASCII characters. In the currently implemented ASCII file servers, the following encoding is used (see Table 3-1). All printing characters and any characters not mentioned explicitly here are represented as themselves.

Table 3-1 **ASCII File Server Encoding**

Explorer Code	ASCII Transliteration
#o010 (Lambda)	This group of code is preceded by a #o177; that is, #o177 is used as a quoting character.
#o011 (Gamma)	
#o012 (Delta)	
#o014 (Plus-minus)	
#o015 (Circle-plus)	
#o177 (Integral)	
#o200-207 inclusive	
#o213 (Delete)	
#o216 and above	
#o210 (Backspace)	Converted to #o010 (backspace)
#o211 (Tab)	Converted to #o011 (horizontal tab)
#o212 (Line)	Converted to #o012 (line feed)
#o214 (Page)	Converted to #o014 (form feed)
#o215 (Tab)	Converted to #o015 (carriage return) followed by #o012 (line feed)
#o377	Ignored completely — cannot be stored in files

When a file server is first created for you on a particular non-Explorer host, you must tell the server how to log in on that host. This procedure involves specifying a *user name* and, if required by that file server, a password. The Explorer system prompts you for these on the terminal when they are needed.

Logging in on a non-Explorer file server is not the same as logging in on an Explorer. The latter identifies you as a user in general and involves specifying one host, your login host. (For more information about logging in, see the documentation for that host's system.) The former identifies you to a particular file server host and must be performed for each host on which you access files. However, logging in on the Explorer system does specify the user name for *your* login host, and logs in on a file server there.

The Explorer system uses your user name as a first guess for your password (this takes no extra time). If that does not work, you are asked to type a password, or else a user name and a password, on the keyboard. You do not have to give the same user name with which you logged in, because you may have or use different user names on different machines.

Once a password is recorded for one host, the system uses that password as the guess if you connect to a file server on another host.

Chaosnet File Server Variables

3.2.1 The following variables are associated with the Chaosnet implementation of a file server.

fs:user-unames Variable

This variable represents an association list that matches host names with the user names that you have specified on these hosts. Each element is the cons of a host object and the user name, expressed as a string.

fs:user-host-password-alist Variable

Once you have specified a password for a given user name and host, that password is remembered for the duration of the session in this variable. The value is a list of elements, each of the following form:

```
((user-name hostname) password)
```

All three elements are strings.

The remembered passwords are used if more than one file server is needed on the same host, or if the connection is broken and a new file server needs to be created. If you do not want your password known, turn off the recording by setting the following variable.

fs:record-passwords-flag Variable

If this variable is non-**nil**, passwords are recorded in the password alist when you type them in.

You should set **fs:user-host-password-alist** at the front of your initialization file and also set **fs:record-passwords-flag** to **nil**, because it already recorded your password when you logged in.

fs:host-unit-lifetime Variable

If you do not use a file server for an extended period of time, it is killed in order to save resources on the server host.

The **fs:host-unit-lifetime** variable indicates the length of time after which an idle file server connection should be closed (expressed in 60ths of a second. The default is 20 minutes.

Chaosnet File Server Functions

3.2.2 Some hosts have a caste system in which all users are not equal. It is sometimes necessary to enable your privileges in order to exercise them. This is done with the following functions.

fs:enable-capabilities *host* &rest *capabilities* Function

The **fs:enable-capabilities** function enables the named capabilities on file servers for the specified host. The *capabilities* argument is a series of strings whose meanings depend on the particular file system that is available on *host*. If *capabilities* is **nil**, a default list of capabilities is enabled; the default is also dependent on the operating system type.

fs:disable-capabilities *host* &rest *capabilities* Function

The **fs:disable-capabilities** function disables the named capabilities on file servers for the specified host. The *capabilities* argument is a series of strings whose meanings depend on the particular file system that is available on *host*. If *capabilities* is **nil**, the default list of capabilities is disabled; the default is also dependent on the operating system type.

Chaosnet File Server Conditions

3.2.3 The following conditions are signaled when there are errors in communication with file servers.

fs:file-request-failure (fs:file-error) Condition

This condition name categorizes errors that prevent the file system from processing the request made by a program.

The following condition names are always built on the more general classification of **fs:file-request-failure, fs:file-error**, and **error**.

fs:data-error Condition

This condition signifies inconsistent data found in the file system, indicating a failure in the file system software or hardware.

fs:host-not-available Condition

This condition signifies that the file server host is up, but is refusing connections for file servers.

fs:network-lossage Condition

This condition signifies certain problems in the use of the Chaosnet by a file server, such as a failure to open a data connection when it is expected.

fs:not-enough-resources Condition

This condition signifies a shortage of resources needed to consider processing a request, as opposed to resources used up by the request itself. This shortage may include running out of network connections or job slots on the file server host. It does not include running out of space in a directory or running out of disk space, because these are resources whose requirements come from processing the request.

fs:unknown-operation Condition

This condition signifies that the particular file system fails to implement a standardly defined operation, such as expunging or undeleting on a host file system that does not support these capabilities.

Telnet

3.3 The Telnet window allows you to use the Explorer screen as a terminal to another host. When you are in a Telnet window, characters typed on the keyboard are passed to the remote host's Telnet server.

You can enter Telnet in any of the following ways:

- Press SYSTEM T.
- Type `(telnet)` in a Lisp Listener.
- Click on the Telnet item in the main System menu.

The following describes the **telnet** function.

telnet &optional *path mode* — Function

This function makes a Telnet connection to a host specified by *path*, which can be either the name of a valid host or **nil**.

For those sites that have hosts serving as gateways (bridges) between Chaosnet and Arpanet subnetworks, *path* can also be a string that contains information about how to get to a valid host. The following are example formats:

gateway-name ESCAPE *internet-host-name*

internet-host-name / *socket-number*

gateway-name ESCAPE *internet-host-name* / *socket-number*

In the previous examples, ESCAPE means to press the ESCAPE key at that point.

The default for *mode* is **t**. If *mode* is set to **t** and *path* is specified, a non-connected Telnet window is selected and a connection to *path* is made. If *mode* is set to **t** and *path* is **nil**, a connected Telnet window is selected, if there is one available; if there is not one available, a connection is made to the host named by the variable **telnet:telnet-default-path**.

When *mode* is **nil**, a Telnet window is selected. If *mode* is set to **nil** and *path* is specified, a new Telnet window is selected and a connection to *path* is attempted. If *mode* is set to **nil** and *path* is **nil**, a connection is made to the host whose name is the current value of the variable **telnet:telnet-default-path**.

Once you have established a Telnet connection, you can transmit key combinations, or key sequences, for generating all 128 USASCII codes. For more information, refer to J. Postel and J. Reynolds' *Telnet Protocol Specification*, RFC 854, USC/Information Sciences Institute, May 1983.

Entering a Telnet Window

3.3.1 When you enter a Telnet window, you are prompted for a name that specifies the host to which you want to connect. If a connection is already established when you enter the window, Telnet continues to use that connection and you are not prompted for a host name.

The Telnet General Help window shows the various ways of identifying this host. One way is to specify the host as a string which the remote host recognizes as its host name or alias.

You can also specify a string formed from the name of the host followed by a slash and the *connect name* (for example, "HOST/TELNET"). The connect name is used by a server to recognize a service, and must be formed of uppercase ASCII letters, numbers, and/or punctuation. After a connection is established, the connect name is discarded. Telnet's connect name is "TELNET".

For those sites that have hosts serving as gateways (bridges) between Chaosnet and ARPANET subnetworks, the host name can be substituted by a string that contains information about how to get to a valid host. The following are example formats:

gateway-name ESCAPE *internet-host-name*

internet-host-name / socket-number

gateway-name ESCAPE *internet-host-name / socket-number*

Once you are connected to a host, most of the keys on the terminal keyboard lose their normal function and their characters pass to the remote host. Particular keys are affected as follows:

- The SYSTEM and TERM keys retain their normal function (and are therefore not forwarded).
- The ABORT, BREAK, and RESUME keys retain their normal function and they are not forwarded to the remote host.
- The CLEAR INPUT key sends the Erase Line (EL) Telnet command to the remote host.
- The STATUS key sends the Are You There (AYT) Telnet command to the remote host.
- The NETWORK key is the first key of a two-key sequence that sends a Telnet command to the remote host.
- The END key exposes the previously selected window and leaves the Telnet connection open.

Because Telnet is implemented with the Universal Command Loop (UCL), Telnet command descriptions and online help are available from the HELP key.

Telnet Commands

3.3.2 To enter a Telnet command, use a two-key sequence, where NETWORK is the first key, followed by one of the keys listed in Table 3-2.

Table 3-2

Telnet Commands

Keystroke	Description
A	Send the Abort Output (AO) Telnet command to the remote host.
CLEAR INPUT	Send the Erase Line (EL) Telnet command to the remote host.
D	Disconnect from the current remote host and ask for the name of another remote host to which you want to connect.
END	Expose the previously selected window and leave the Telnet connection open.
HELP	Describes the Telnet commands. Also, you can obtain this information by pressing HELP and clicking on Command Display.
M	Toggle the ***more*** processing variable on the Telnet window.
O	Toggle between Insert mode (the default mode) and Overwrite mode.
P	Send the Interrupt Process (IP) command to the remote host.
Q	Expose the previously selected window and disconnect from the remote host.
STATUS	Send the Are You There (AYT) Telnet command to the remote host.

Telnet Server

3.3.3 When a connection is first established, Telnet sends the remote host the print herald of the local machine. Telnet then uses the read-eval-print loop to set up a communications loop, reading characters from the remote host until a complete Lisp expression arrives. Telnet then evaluates the Lisp expression and returns the resulting value(s) to the remote host.

Use the following function on your system to enable or disable the Telnet server.

telnet:telnet-server-on (*mode* **:notify**) Function

This function enables and disables the Telnet server. *mode* can take on the following values: **t** for on; **nil** for off; **:notify** (the default) for on, with the condition that the user is notified when a connection is made; and **:not-logged-in** for on, if no one is logged in.

NOTE: Note that this function affects what happens when a remote host attempts to use Telnet on your system. It does not affect a remote host on which you may later want to use Telnet.

VT100 Emulator

3.4 The VT100 emulator runs as an application on top of Telnet. With the VT100 emulator, you can use the Explorer monitor and keyboard as a VT100 terminal. The VT100 emulator is resident on the Explorer system and has the following features:

- All alphanumeric keys and cursor-control keys transmit the proper VT100 codes.
- All function keys transmit VT100 codes, except the BREAK key and its variants.
- The keys on the Explorer keypad transmit VT100 auxiliary keypad codes.
- Most of the control commands used by a VT100 terminal are emulated on the Explorer.
- There are no Explorer keys that correspond to the VT100 SETUP and NO SCROLL keys.
- CTRL-H corresponds to the VT100 BACKSPACE key.

The VT100 emulator is implemented using the UCL; command descriptions and online help are available from the HELP key.

For a description of the VT100 escape and control sequences, refer to the *VT100 User's Guide*, published by Digital Equipment Corporation.

To enter an Explorer window that emulates the VT100 terminal, press SYSTEM V. The VT100 emulator frame has an automatic-scroll window and a light-emitting diode (LED) window. If a connection has not already been established before you enter the VT100 frame, you are prompted in the automatic-scroll window for the name of a host to which you want to connect. The LED window displays four LEDs that simulate the LEDs on a VT100 keyboard. The LEDs are turned on and off by the appropriate key sequences.

In addition to a default character font, the font map for a VT100 window includes the following:

- Graphics font
- Top and bottom fonts
- Double-wide font

The graphics font matches the VT100 special graphics character set. It is selected after you enter the proper VT100 command sequence. The VT100 emulator uses the top and bottom fonts when you enter the double-height, top-bottom command sequence. It uses the double-wide font after you enter the double-width command sequence.

The following VT100 control sequences are currently not implemented on the Explorer system:

- Underscore on
- Bold on (used with graphics font)
- Invoke confidence test
- Cursor Key mode
- ANSI/VT52 mode
- Scrolling mode
- Origin mode
- Autorepeat
- Interlace

The Explorer builds the VT100 emulator frame on top of Telnet. Just as in the Telnet window, most of the keys pressed on the keyboard are passed to the remote host. Particular keys are affected as follows:

- The SYSTEM and TERM keys retain their normal function (and are therefore not forwarded).
- The ABORT, BREAK, and RESUME keys retain their normal function and are not forwarded to the remote host.
- The CLEAR INPUT key sends the Telnet EL command to the remote host.
- The STATUS key sends the Telnet AYT command to the remote host.
- NETWORK is the first key of a two-key sequence that sends a Telnet command to the remote host. Most of the Telnet commands are also valid in the VT100 emulation frame. See the Telnet section for a description of these commands.

Table 3-3 describes the VT100 commands displayed in the command menu. Be sure to press the NETWORK key before each VT100 command.

Table 3-3 VT100 Commands

Name	**Keystroke**	**Description**
Answerback	B	Send the Answerback message string in ***vt100-answerback-message***
80/132 Columns	C	VT100 Setup-A 80/132 Columns
Set Lines	L	Set the number of lines for the VT100 screen and reconfigure screen
Reset	R	VT100 Setup-A Reset
VT100 Switch	S	Enable/Disable VT100 escape sequence processing
Truncate	T	Toggle truncating of VT100 screen pane
Reverse Video	V	Complement black-on-white state of VT100 screen

Converse

3.5 Converse is an interactive message editor that displays all the messages that you have sent or received. You can enter Converse in any of the following ways:

- Press SYSTEM C.

- Type the form `(qsend)` in a Lisp Listener.

- Click on the Converse item in the System menu.

On the screen, Converse groups into a *conversation* all the messages that you send to or receive from a particular user. Conversations from different users are separated by thick dividing lines. Do not delete these dividing lines because this will cause messages to be lost. Messages are grouped according to user name. Messages are put into separate conversations in the following cases:

- The same user name is employed on different hosts

- Different user names are employed

Within a conversation, the name of the other party to the conversation appears at the top of a chronologically ordered group of messages. Converse creates a new conversation when you begin communicating with someone for whom there is no existing conversation.

When Converse asks you to enter a user name with the `To:` prompt, use the following syntax conventions:

- When you want to deal with one user, specify the form *user@host*, where *user* is a valid user name and *host* is a valid host name.

- When you want to deal with multiple users, specify the form *user@host*, *user-1@host-1*, *user-2@host-2*,... *user-n@host-n*. A separate copy of your message will go into the conversation for each of the recipients.

- If you just enter *user* instead of *user@host*, Converse will try to find an Explorer that *user* is logged in to and forward the message to that host. If you give the variable **zwei:*converse-extra-hosts-to-check*** a list of hosts to check, Converse tries to determine where *user* is logged in among these hosts. If *user* is not logged in under any of the hosts in that list, Converse gives you a menu of hosts from which to choose in order to continue your search for *user*.

- You can omit the user name and specify only the host as *@host*. Whoever is logged in on that host receives your message.

To send a message, perform the following:

1. Move the cursor to the right of the `To:` prompt and type the user name.

2. Press the RETURN key and enter your message.

3. Press the END key or use the CTRL-END sequence in order to transmit.

Zmacs Editor Commands With Converse

3.5.1 Converse allows you to use most of the Zmacs editor commands to edit your messages and move them around to different conversations. Table 3-4 lists the additional Converse commands that are available.

Table 3-4 Converse Commands

Key Sequence	Explanation of Command
END	Send the current message without exiting from Converse.
CTRL-END	Send the current message and exit from Converse.
ABORT	Eliminate the current Converse window.
CTRL-M	Mail the current message instead of sending it with Converse.
META-{	Move to the previous To: line.
META-}	Move to the next To: line.
META-X Delete Conversation	Delete the current conversation.
META-X Write Buffer	Write all of the conversations into a file.
META-X Write Conversation	Write only the current conversation into a file.
META-X Append Conversation	Append the current conversation to the end of a file.
META-X Regenerate Buffer	Rebuild the buffer structure. This command is useful if you edit in or across the thick dividing lines that separate conversations, which damages the buffer structure. Some error messages may suggest that you execute this command before retrying an operation. Note that this command deletes anything you have inserted in the buffer but have not yet sent.
META-X Gag Converse	Toggle the value of the **zwei:*converse-gagged*** variable. If set to **t**, the variable tells Converse to reject incoming messages; if set to **nil**, it tells Converse to accept incoming messages.

Converse Functions

3.5.2 You can use the following Converse functions to send or reply to a message:

qsends-off &optional *gag-message* — Function

If the value of *gag-message* is set to **t**, which is the default, Converse rejects all incoming messages; if it is set to **nil**, Converse accepts all incoming messages. This command is useful to specify whether you want to be interrupted with any interactive messages. *gag-message* can be a string that is automatically forwarded as a reply to the sender of a message.

qsends-on Function

This function specifies that all incoming messages are to be accepted. This function is the complement to the **qsends-off** function.

qsend &optional *destination message mail-p wait-p* Function

This function sends *message*, which should be a string, to the user name(s) specified in *destination*. If *message* is empty, you are prompted for its contents. *destination* can be one of the following:

- When you want to deal with one user, specify *user@host*, where *user* is a valid user name and *host* is a valid host name.
- When you want to deal with multiple users, specify a list in the form (*user@host user-1@host-1 user-2@host-2* ... *user-n@host-n*).
- If you just enter *user* instead of *user@host*, Converse will try to find an Explorer that *user* is logged in to and forward the message to that host. If you give the variable **zwei:*converse-extra-hosts-to-check*** a list of hosts to check, Converse tries to determine where *user* is logged in among these hosts. If *user* is not logged in under any of the hosts in that list, Converse gives you a menu of hosts from which to choose in order to continue your search for *user*.
- You can omit the user name and specify only the host as *@host*. Whoever is logged in on that host receives your message.

If *destination* is empty, the user is put into the Converse window and asked to specify the destination.

If *mail-p* is set to **nil**, which is the default value, the message is sent interactively; if it is set to **t**, the message is mailed instead.

If *wait-p* is set to **nil**, the **qsend** function immediately returns **nil** as its value and sends the message in background mode. If *wait-p* is set to **t**, the **qsend** function monitors the status of the message it sends and returns a list of the recipients that received the message. The default value for *wait-p* is set to the value of the **zwei:*converse-wait-p*** variable.

```
(qsend '(john@zebra jane@giraffe) "The ark is ready"))
```

zwei:reply &optional *message destination mail-p wait-p* Function

This function sends *message*, a string, to the last user who sent you a message. If *message* is empty, the command prompts you for the contents of the message.

Because the default value of *destination* is set to the value of **zwei:*last-converse-sender***, the message goes to the host from which the last user sent a message, unless you specify another host as *destination*. If you specify *destination*, provide a user name.

If *mail-p* is set to **nil**, which is the default value, the message is sent interactively; if it is set to **t**, the message is mailed instead.

If the value of *wait-p* is **nil**, the **zwei:reply** function returns the recipient of the message. If *wait-p* is set to **t**, the **zwei:reply** function monitors the status of the message it sends and returns a list of the recipients that received the message. The default value for *wait-p* is set to the value of the **zwei:*converse-wait-p*** variable.

User Options With Converse

3.5.3 You can set the following user options in your login initialization file:

zwei:*converse-receive-mode* Variable

This variable controls what occurs when you receive a new interactive message. The variable can take on one of five values:

:auto indicates that the Converse window is automatically entered when a message arrives.

:notify indicates that, whenever a message arrives, you are to be informed about both its arrival and its origin.

:notify-with-message is similar to **:notify**, except that you are given the sender's message as well as the sender's name. This is the default value for the variable.

:pop-up indicates that the receipt of a message results in the appearance of a pop-up window on the screen. The window gives you the option to reply to the message, to enter the Converse window, or to do nothing at all. This window notifies you in the same way as **:notify-with-message**.

:simple is the same as **:pop-up**.

zwei:*converse-append-p* Variable

If this variable is set to **t**, a new message is appended to the end of the sender's conversation. If this variable is set to **nil**, which is the default value, a new message is added to the beginning of the sender's conversation.

zwei:*converse-beep-count* Variable

This variable indicates the number of times the Explorer will beep when a message arrives. The default value is two.

zwei:*converse-extra-hosts-to-check* Variable

This variable indicates which hosts are checked when someone types *user* instead of *user@host* when asked for a user name. The hosts are checked to see if *user* is logged in to any of those hosts. To specify a group of hosts, enter a list of valid host names. This variable defaults to **nil**, which means that all hosts are checked.

zwei:*converse-end-exits* Variable

If the value of this variable is **t**, the following will happen after you type a message:

- If you press END, the message is sent and you exit Converse.

- If you press ABORT, the message is not sent and you exit Converse.

- If you press CTRL-END, the message is sent and you remain in Converse.

If the value of this variable is **nil**, which is the default value, the following will happen after you type a message:

- If you press END, the message is sent and you remain in Converse.
- If you press ABORT, the message is not sent and you exit Converse.
- If you press CTRL-END, the message is sent and you exit Converse.

zwei:*converse-gagged* Variable

If the value of this variable is not **nil**, Converse will reject all incoming messages. In this case, **zwei:*converse-gagged*** should be a string that contains the reason why you are not receiving messages.

If the value of this variable is **nil**, which is the default value, you will receive all incoming messages.

Although this variable is available for your convenience, the **qsends-on** and **qsends-off** functions are recommended instead.

zwei:*converse-wait-p* Variable

If this variable is set to **t**, which is the default value, Converse waits to determine the status of any message that you send. If set to **nil**, Converse does not monitor the status of messages.

Name

3.6 This is the ARPANET Name protocol. The Name protocol establishes a full connection—with retransmission—to get the names of user(s) currently logged in to a host. When used with a Lisp machine, this protocol is similar to the Finger protocol. There is no Explorer high-level interface to this protocol.

Time

3.7 The Time protocol allows a host to ask the time of day. It is used by the system software to get the time of day. There is no Explorer high-level interface to this protocol.

Eval Serving

3.8 The Eval server allows you to set up a read-eval-print loop on a remote host. Before you can use the Eval server, it must be enabled on the remote host. Use the following functions to turn on the Eval server at the host site.

chaos:eval-server-on (*mode* **t**) Function

This function enables and disables the Eval server. The *mode* argument can take on the following values: **t** for on (the default), **nil** for off, **:notify** for on, with the condition that the user is notified when a connection is made, or **:not-logged-in** for on, in the case that no one is logged in.

After you enable the server at the remote host, start the Eval session by issuing the **chaos:remote-eval** function at your host.

chaos:remote-eval *host* Function

This function initiates an Eval server session to *host*, where the read-eval-print loop is handled. Results are returned to your terminal until you terminate the session.

Each time a complete symbolic expression arrives, the Eval server reads it, evaluates it, and sends back the result. Terminate the Eval session by pressing ABORT.

Fingering Hosts

3.9 The **finger** function and the TERM F key sequence display information about users logged in at various machines in your network. The **chaos:find-hosts-or-lispms-logged-in-as-user** function returns a list of hosts on which a user is logged in.

The **finger** function and the TERM F key sequence display the following information about a user logged in at a machine in your network:

- Login name
- Full name
- Process now running
- Idle time (if any)
- Location

Both the **finger** function and the TERM F key sequences are interfaces to the Finger protocol. The Finger protocol is a Lisp machine version of the ARPANET Name protocol that uses a simple transaction instead of a stream connection.

The Finger Function

3.9.1 The following paragraphs describe the **finger** function.

finger &optional *spec* (*stream standard-output*) *hack-brackets-p* Function

Prints brief information about a user as specified by *spec*. The *spec* argument can be *user@host* or *@host*. The *stream* argument specifies where to print the information. If *hack-brackets-p* is **t**, the first line shows what host you are fingering.

If you enter the **finger** function with no arguments, your own machine is fingered.

You can also obtain finger information by pressing TERM 0 F. You are prompted for *user@host* or *@host*.

Pressing TERM F displays information about all users logged in at the various machines in your network.

Finger digits can be assigned via the Namespace Editor. These assignments specify which machines to finger. For example, pressing TERM 3 F might display information about the users logged on the machines ROMEO, SIERRA, and TANGO.

Example:

```
(finger "@young")
```

The following is displayed:

```
LISA  -                ZMACS     12       TI-AUSTIN
```

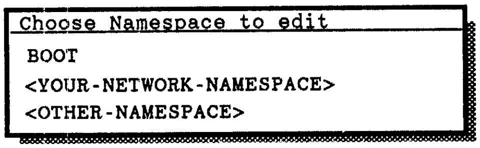
chaos:find-hosts-or-lispms-logged-in-as-user *user* &optional *hosts no-lispms-p* Function

Returns a list of hosts on which *user* is logged in. The *hosts* argument is the list of hosts to check (in addition to all Lisp machines). If *no-lispms-p* is **t**, the function does not return any Lisp machines.

Making Finger Assignments

3.9.2 To make finger assignments, you must first have a valid network namespace, or you must create one, adding your finger assignments as you do so. Creating a network namespace is discussed in detail in Section 4, titled Getting on a Network. If you are unfamiliar with the Namespace Editor, see the *Explorer Tools and Utilities* manual. The following paragraphs describe how to make specific finger assignments in a pre-existing network namespace.

1. Enter the Namespace Editor by selecting it from the system menu or by using the Edit Namespace Zmacs command. A pop-up menu will appear similar to the one following:



```
Choose Namespace to edit
BOOT
<YOUR-NETWORK-NAMESPACE>
<OTHER-NAMESPACE>
```

2. Click on the name of your network namespace. The Namespace Editor now brings your network namespace into an editing buffer on your screen.

3. Place the mouse cursor box around the class name :SITE and click middle on the mouse. A menu of commands appears.

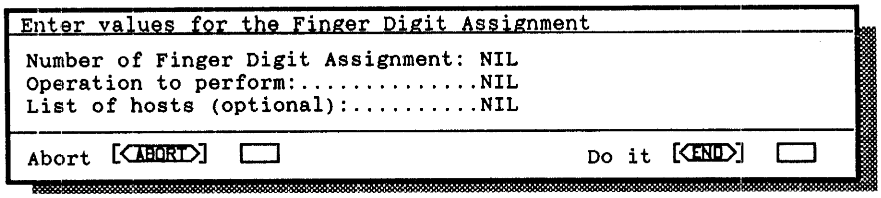
4. Click on the Expand/Unexpand Class command. The :SITE class name expands, listing all of the site objects for your network namespace.

5. Place the mouse cursor box around the object that represents your network namespace's site and click middle on the mouse. A menu of commands appears.

6. Click on the Expand/Unexpand Object command. The site listing expands showing all of the attributes (and their values) for your site. Notice that the :terminal-f-arguments attribute already has the default values described earlier in this paragraph.

7. Place the mouse cursor box around the :terminal-f-arguments attribute and click middle on the mouse. A menu of commands appears.

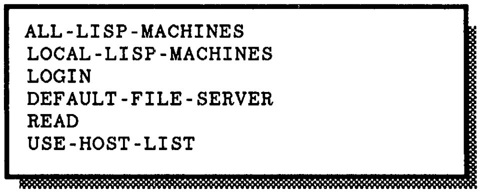
8. Click on the Add Group Member command. A pop-up menu appears similar to the one following:



```
Enter values for the Finger Digit Assignment
Number of Finger Digit Assignment: NIL
Operation to perform:..............NIL
List of hosts (optional):..........NIL

Abort [<ABORT>] □                    Do it [<END>] □
```

9. The `Number of Finger Digit Assignment` prompt requires the actual finger digit itself. A finger digit is a numeric value to be used as the *n* value when you press TERM-*n*-F. Click on the `NIL` next to the prompt, enter the new finger digit, and press RETURN.

10. The `Operation to perform` prompt allows you select the type of operation you want the fingering to perform. Click on the `NIL` value for this prompt. The following pop-up menu appears:



    The entries are as follows:

    - `ALL-LISP-MACHINES` — Fingers all machines with a **:system-type** attribute of **:explorer**, **:lispm**, or **:symbolics**. For information on system type attributes, see paragraph 4.2.2, titled Network Namespace Attributes. This entry will update your local cache from the network.

    - `LOCAL-LISP-MACHINES` — This selection performs the same job as `ALL-LISP-MACHINES`, except that it only looks in your local cache. Because of this, `LOCAL-LISP-MACHINES` is normally faster than `ALL-LISP-MACHINES`.

    - `LOGIN` — Fingers the host to which you are logged into. This host, which is identified by the **net:user-login-machine** variable, may or may not be the local host.

    - `DEFAULT-FILE-SERVER` — Fingers the host which is the default file server for the host you are logged into. This host is identified by the **net:associated-machine** variable.

    - `READ` — Prompts you for the name of a host whenever you invoke the Finger utility.

    - `USE-HOST-LIST` — Allows you to specify which hosts you want for this finger digit assignment. You specify the list of hosts in the `List of hosts (optional)` prompt.

11. Click on the `USE-HOST-LIST` value. The value of `USE-HOST-LIST` replaces `NIL` in the original pop-up menu.

12. Click on the `NIL` next to the `List of hosts (optional)` prompt, enter the names of any hosts you want associated with the finger digit and press RETURN. Your new host list replaces `NIL`. The list should use the following syntax:

    ```
    ("Slocum" "Chichester" "Magellan" "LM2-Alpha")
    ```

13. Press the END key or click on the `Do it` at the bottom of the pop-up menu. You are then returned to the Namespace Editor buffer.

14. To put the changes into effect for the current session only, you must update the network namespace locally. To make your changes permanent for the network, you must update them globally. Both options are available to you if you click middle on the mouse and select the appropriate menu command.

Remote Disk Server and Band Transfers

3.10 The remote disk server is the process by which you can read from and write to disks on remote hosts. These operations are transparent to higher-level functions. Therefore, functions such as **print-disk-label** and **load-mcr-file** can be executed on a remote host.

To address absolute disk locations, pass to the sys:**disk-read** and sys:**disk-write** functions a unit identifier that is a closure representing the target disk on the remote host.

The closure is created by the following function.

sys:**decode-unit-argument** *unit use* — Function

This function returns a closure that identifies the remote unit.

Arguments:

unit can be a string that contains the name of a remote host. In this case, the target disk will be the disk that is defined as the default disk on the remote host. If you want to specify another disk instead, *unit* should be a string that contains the host name, followed by a colon, followed by a unit number or pack name. For example, "explorer" specifies the default disk on a host named Explorer; "explorer:2" specifies disk unit 2 on the same host.

use is a string that describes the reason for accessing a remote disk. The string is displayed on the remote host's terminal when the network connection is created.

This function returns two values. The first is a closure that represents the network connection to a remote host. If the second value is **nil**, the function did in fact decode *unit*. **nil** indicates that, when the remote operation has completed, you need to call the sys:**dispose-of-unit** function to release the closure. If the second value is **t**, *unit* was already a decoded unit. A **t** indicates that it is up to someone else to call the sys:**dispose-of-unit** function.

Most disk utilities already use the **sys:decode-unit-argument** function so that access to remote disks is accomplished by specifying the host name and unit number in the *unit* parameter. You can use the following functions with remote disks as well as local disks:

- **sys:print-disk-label**
- **sys:copy-disk-label**
- **sys:edit-disk-label**
- **sys:copy-disk-partition**
- **sys:compare-disk-partition**
- **sys:set-pack-name**
- **sys:get-pack-name**
- **sys:current-band**
- **sys:set-current-band**
- **sys:describe-partition**
- **sys:describe-partitions**
- **sys:get-ucode-version-of-band**
- **sys:measured-size-of-partition**
- **sys:print-available-bands**
- **sys:load-mcr-file**

Full descriptions of these utilities can be found in the *Explorer Input/Output Reference* manual.

The previous functions work well for printing and editing disk labels. For comparing or copying partitions, the following functions are more efficient for machine-to-machine transfers.

sys:**receive-band** *from-machine from-part to-unit to-part* &optional *subset-start subset-n-blocks* — Function

This function copies a partition from a remote machine.

from-machine is a string that contains the name of the remote host, followed by a colon, followed by a unit number or pack name. If you do not specify the unit number, the default unit on the remote host is used.

from-part is a four-character string that contains the name of the target partition on the remote disk.

to-unit is the unit number or pack name that contains the name of the destination disk on the local host.

to-part is a four-character string that contains the name of the destination partition on the local disk.

subset-start is a number that specifies an offset from the beginning of the partition, measured in hundreds of blocks. This offset is used to specify the point to restart the operation if the connection is broken for any reason.

subset-n-blocks is the number of blocks you want to transfer in case only a partial transfer is required, measured in hundreds of blocks.

As the sys:**receive-band** function is executing, the number of blocks—in hundreds—that have been transferred is displayed. For example, 4 indicates 400 blocks have been transferred. Knowing the number of blocks is useful if the network connection is broken for any reason. In that case, you could restart the function at the last hundredth block transferred. When you restart the function, enter the last displayed number as the value for the *subset-start* parameter. The copy will begin from there.

sys:**transmit-band** *from-part from-unit to-machine to-part* &optional *subset-start subset-n-blocks* — Function

This function transmits a partition from a local disk to a target partition on a remote host.

from-part is a four-character string that contains the name of the target partition on the local disk.

from-unit is the unit number or pack name of the target disk on the local host.

to-machine is a string that contains the name of the remote host, followed by a colon, followed by a unit number or pack name. If you do not specify the unit number, the default unit on the remote host is used.

to-part is a four-character string that contains the name of the destination partition on the remote disk.

The *subset-start* and *subset-n-blocks* options are identical to those of the sys:**receive-band** function.

As the **sys:transmit-band** function is executing, the number of blocks—in hundreds—that have been transferred is displayed. For example, 4 indicates 400 blocks have been transferred. Knowing the number of blocks is useful if the network connection is broken for any reason. In that case, you could restart the function at the last hundredth block transferred. When you restart the function, enter the last displayed number as the value for the *subset-start* parameter. The copy will begin from there.

sys:**compare-band** *from-machine from-part to-part* &optional *to-unit subset-start subset-n-blocks* Function

This function compares the contents of two partitions on different machines. It is useful for verifying that a remote copy completed without error. If any differences are found, an error message is displayed that specifies the location of the difference according to the block number and half-word offset within the block. To keep error messages manageable, a maximum of three error messages per block are printed.

All arguments are identical to those of the **sys:receive-band** function. The *to-unit* option defaults to **sys:*default-disk-unit***.

As the **sys:compare-band** function is executing, the number of blocks—in hundreds—that have been transferred is displayed. For example, 4 indicates 400 blocks have been transferred. Knowing the number of blocks is useful if the network connection is broken for any reason. In that case, you could restart the function at the last hundredth block transferred. When you restart the function, enter the last displayed number as the value for the *subset-start* parameter. The copy will begin from there.

Sending and Printing Notifications

3.11 The **chaos:shout**, **chaos:notify-all-lms**, and **chaos:notify** functions send notifications to Lisp machines. The **print-notifications** function reprints any notifications that have been received.

In addition to the functions that send notifications, a notification can come from utilities such as Mail and Converse. Also, a notification is not restricted to the network. It can be an asynchronous message from the Explorer system itself.

chaos:shout Function

Sends a message to all Lisp machines. The message is read from the terminal. The message should be brief; otherwise, you should use mail.

chaos:notify-all-lms &optional (*message* (**notify-get-message**)) Function

Sends a *message* to all Lisp machines. The message is printed as a notification. If you omit *message*, the message is read from the terminal. The message should be brief; otherwise, you should use mail.

chaos:notify *host* &optional (*message* (**notify-get-message**)) Function

Sends a *message* to *host*. The message is printed as a notification. If you omit *message*, the message is read from the terminal. The message should be brief; otherwise, you should use mail.

print-notifications Function

Reprints any notifications that have been received. The difference between a notification and a send is that a send comes from other users, while a notification is usually an asynchronous message from the Explorer system itself. However, the default way for the system to inform you about a send is to make a notification. So **print-notifications** *normally* includes all sends as well as notifications.

GETTING ON THE NETWORK 4

Introduction

4.1 Release 3 of the Explorer networking software changes the entire aspect of network configuration. Several situations now exist for which this section provides support. These include the following:

- You are updating an existing Explorer network from Release 2 software to Release 3 software. In so doing, you must convert your existing network configuration to a *network namespace*.
- You are creating an Explorer network from scratch. In so doing, you must create a new network namespace.
- You are modifying an existing Explorer network (such as adding hosts or printers). You must modify the existing network namespace.

NOTE: If you are not familiar with namespace concepts, you can refer to the *Explorer Tools and Utilities* manual. This section assumes that you are familiar with the basic namespace concepts.

The first part of this section introduces the concepts involved with network namespaces, including classes, objects, attributes, servers, caching, and so on.

Next, several paragraphs give explicit information about how to update your network from Release 2 to Release 3, and how to create a network namespace for a group of Explorer systems just up and running from their shipping crates.

The section closes with a group of reference paragraphs that detail the various options outlined in the network initialization menu.

The Network Namespace

4.2 With Release 3, Explorer network configuration information is stored in a *network namespace*. The hosts in a network refer to a network namespace for all the information they need concerning network activities.

All network namespaces should be public namespaces. Public namespaces are much faster at recording updates, and they can be loaded automatically when a server is booted.

To create or modify a network namespace, you use the Namespace Editor, rather than the network configuration menu-driven interface used in previous releases.

The information in a network namespace is arranged hierarchically. A network namespace contains six classes, which in turn contain objects that contain various attributes, which in turn have certain value assignments. The following two paragraphs discuss network namespace classes and their associated attributes.

Network Namespace Classes

4.2.1 A network namespace contains six classes of objects that relate to a network. Although the following list only describes the system-defined classes, you can use the Namespace Editor to add your own. The six network namespace classes are as follows:

- **:host** Class — The objects in the **:host** class represent the hosts on a network, either logical or physical.

- **:mailing-list** Class — The objects in the **:mailing-list** class represent actual mailing lists, with the name of the object being the same as that of the mailing list itself.

- **:namespace** Class — The objects in the **:namespace** class represent actual namespaces. At the very minimum, one object should exist in this class; that object represents the current namespace. You can also add namespace objects for other namespaces in order to tell how these namespaces are to be accessed.

- **:printer** Class — The objects in the **:printer** class are the printers on your network.

- **:site** Class — The object names in the **:site** class are strings that represent the name of network sites. Objects in this class contain information about the resources available at the site; that is, most of the hosts for this site have common attributes.

- **:user** Class — The objects in the **:user** class represent the users on your network. Each object is the same as the user ID (or login name) for that user.

Each object in these classes has attributes and attribute values. While objects in different classes can duplicate an attribute name, the value assigned to these duplicated attributes can differ from object to object. One feature that the namespace system uses to its advantage is that the value of one duplicated attribute can override the value of another.

In the network namespace system, the **:user**, **:host**, and **:site** classes (in that order) take advantage of overriding attribute values. When duplication occurs in all three of these classes, the actual value used by the network namespace is that of the **:user** attribute. If duplication occurs only between the **:host** and the **:site** attribute, the network namespace takes the value of the **:host** attribute. Accordingly, the value of the **:user** attribute is taken over that of the **:site** attribute.

Network Namespace Attributes

4.2.2 Each network namespace class has certain attributes associated with it, and values are assigned to those attributes. Some attributes are defined by the system; however, you can use the Namespace Editor to add your own or to modify the defaults supplied by the Explorer system.

The attributes available for the network namespace system are alphabetically arranged in the bulleted list that follows. Following each of the attribute names is a letter (or letters) that tells the class (or classes) for which the attribute is used. The letters are:

H — **:host** class
M— **:mailing-list** class
N — **:namespace** class
P — **:printer** class
S — **:site** class
U — **:user** class

The network namespace attributes are as follows:

- **:address-list** (M) — Mail addresses of recipients for mail that is being sent to a mailing list.

- **:addresses** (H) — This attribute is a list of network addresses for a host. The syntax for this attribute is as follows:

  ```
  '((net-type addr) (net-type addr) ...)
  ```

  For example:

  ```
  '((:chaos 1464) (:IP 1090550504) (:IP 1090550504) ... )
  ```

  NOTE: Even though a host may have multiple IP addresses, it will not act as an IP gateway unless you specifically add the following as a group member to the **:services** attribute associated with that host:

  ```
  (:gateway :ip :ip-gateway)
  ```

- **:*alias-of*** (H S U) — The name of an object for which this object is an alias.

- **:aliases** (H) — This attribute identifies a list of aliases associated with a particular host. Alias objects are added and/or deleted automatically after this attribute is edited.

- **:auditing-enabled** (N) — If **t**, all changes made to the namespace are written into the lm:name-service;<*namespace-name*>-audit.text#> file, thereby providing an easy-to-read summary of the changes made. This attribute is not applicable to Symbolics™ or Personal namespaces.

- **:auto-save-enabled** (N) — If **t**, all changes made to the namespace are automatically written to the lm:name-service;<*namespace-name*>.xld#> file as soon as the number of changes reaches the value set by the **:changes-before-save** attribute. The **:auto-save-enabled** attribute applies only to co-servers. Clients cannot write to the XLD file.

  If the value of **:auto-save-enabled** is **nil**, changes are written only into the log file (lm:name-service;<*namespace-name*>-log.text). However, if the value of the attribute is **nil**, you can force changes to be written to the XLD file by invoking the **name:distribute-namespace** function (which is discussed at the end of this section).

- **:baud** (P) — This attribute identifies the baud rate for the printer on the serial interface. The acceptable values for this attribute include the following: 300., 600., 1200., 1800., 2000., 2400., 3600., 4800., 7200., 9600., or 19200. The default is 4800., which is the default value for the **printer:*default-baud-rate*** variable. Ignore this if the interface is not via the serial port.

- **:bitmap-printer** (H S U) — The value of this attribute specifies the default printer to use when printing screen dumps or other graphics images.

- **:boot-init-file** (H) — The name of a file to be loaded automatically after booting. On a warm boot, the **:boot-init-file** will be loaded only if the file has changed since you last booted.

- **:cache-entry-timeout** (N) — This attribute specifies the amount of time until a locally cached copy of an object times out. (The copy is deleted, and a new copy is obtained from a namespace server.) You can specify this value as a number of seconds or as a list in the following form:

```
(n :second or :seconds
   :minute or :minutes
   :hour   or :hours
   :day    or :days
   :week   or :weeks)
```

  An example is `(30 :minutes)`. The value is stored in the **name:*default-cache-timeout*** variable. The default is `(24 :hours)`. This attribute is not applicable to Personal Namespaces.

- **:caching-control** (N) — This attribute identifies whether or not to store objects received from a co-server in the local cache. For a brief description of caching, see paragraph 4.2.5, titled Network Namespaces, Servers, and Caches. The acceptable values for this attribute include the following:

  - **t** or **:always** — Caches all objects received from a co-server.

  - **:selective** — Caches all objects received from a co-server, unless that object has the **:*non-cacheable*** attribute set to **t**.

Symbolics is a trademark of Symbolics, Incorporated.

- **nil** — Does not cache any objects.

  This attribute is not applicable to Personal Namespaces.

- **:changes-before-save** (N) — If the value of the **:auto-save-enabled** attribute is **t**, the value of the **:changes-before-save** attribute specifies how many changes can be made to the namespace before they are written to an XLD file. This attribute is not applicable to Symbolics namespaces.

- **:character-printer-p** (P) — If the value of this attribute is non-**nil**, the printer can print streams of ASCII characters. The default is **t**.

- **:current-version** (N) — This attribute is used internally by the Namespace system. Do *not* edit or delete this attribute.

- **:data-bits** (P) — This attribute identifies the number of data bits per character used by the serial interface. Acceptable values for the **:data-bits** attribute include the following: 5, 6, 7, or 8. You must use 8 if you want to print screen images on a TI855, TI880, TI2015, or TI2115 printer. The default is 8, which is the default value for the **printer:*default-data-bits*** variable. Ignore this if the interface is not via the serial port.

- **:default-device** (H) — Name of device that should be used as a default in pathnames for a particular host.

- **:default-file-server** (H S U) — This attribute is the name of the default file server associated with the local host. Normally, all of the hosts in a site act as their own default file servers.

- **:default-login-name** (H S U) — User-ID used when getting files from sys-host (such as an error table).

- **:default-login-password** (H S U) — Password used when getting files from sys-host (such as an error table).

- **:default-mail-host** (H S U) — The host to use as the default in mail addresses that have an unspecified host. This applies only to new messages entered into the mail system and does not override the options relating to mail servers and gateways.

- **:directory-translations** (H) — This attribute specifies the translations to be used for a logical host. (For information about logical hosts, see the *Explorer Input/Output Reference* manual.) Pathname translations for logical hosts (such as "SYS"), can be specified in one of two ways:

  - From a translations file identified by the **:translations-file** attribute (which is discussed later in this paragraph); however, this practice is discouraged.

  - From the values of the two attributes **:directory-translations** and **:host-translation** (used together). If these last two attributes are given values on a logical host, they override any translations file specified for that logical host.

    The **:host-translation** attribute specifies the name of the host to which your logical host points.

The **:directory-translations** attribute specifies the translations to be used for that host. The format is a list of sublists that can take one of the three following forms:

- `(logical-directory physical-device-and-directory)`
- `(logical-directory physical-device physical-directory)`
- `(logical-directory physical-device (subdir subdir ...))`

Using a translations file is much slower than specifying the **:host-translation** and **:directory-translations** attributes in the namespace. When you access a logical host that uses a translations file, the translations file is loaded on your local machine (a time-consuming procedure) if either of these two conditions are true: This is the first time you are trying to access the logical host, or you try to access the logical host after the cached version of the host has been marked for refresh.

If you use the **:directory-translations** and **:host-translation** attributes to specify a logical host (the recommended procedure), there is no file to load, so access to logical hosts is generally faster. On the other hand, if the size of your network namespace is a problem, the use of translations files instead of the **:directory-translations** and **:host-translation** attributes would probably make your namespace smaller.

To transfer information from a translations file to the **:directory-translations** attribute, perform the following steps:

1. Bring the translations file into a Zmacs buffer.
2. Find the desired directory translations list, and copy it to the kill history.
3. Find the **:directory-translations** attribute on the desired logical host in the Namespace Editor. If it is a group attribute (marked with a "G"), press G to toggle the group status.
4. Press E to edit the scalar attribute. A pop-up window appears.
5. Press CTRL-Y to yank the translations from the kill history into the pop-up window.
6. Press the END key.
7. Update globally by pressing CTRL-W, or only update locally (for testing) by pressing W.
8. If you have changed it, toggle the group status of the attribute back to its original value by pressing G.
9. Update globally again by pressing CTRL-W.

- **:file-control-lifetime** (H) — Length of time (in 60ths of seconds) to leave idle connections to a file server open.
- **:file-server-type** (H) — Refers to the mode or protocol used by the file server.

- **:ftp-implementation-type** (H) — The name of the implementation of the FTP server running on a particular host.

- **:ftp-prompt-for-account** (H) — If **t**, FTP will ask for account information when prompting for remote login information.

- **:home-host** (U) — The value of this attribute is the name of the host to which the user is normally logged-in to.

- **:home-phone** (U) — The value of this attribute is the home phone number at which a particular user can normally be reached.

- **:host** (P) — The value for this attribute specifies the host on which a particular printer is located. The value for **:host** can be the name of any valid host; for example, "BOLIVAR".

- **:host-for-bug-reports** (H S U) — The value of this attribute is the name of a host to which bug reports submitted via MAIL should be sent.

- **:host-translation** (H) — The name of a host (physical or logical) to which a logical host points. Pathname translations for logical hosts (such as "SYS"), can be specified in one of two ways:

  - From a translations file identified by the **:translations-file** attribute (which is discussed later in this paragraph); however, this practice is discouraged.

  - From the values of the two attributes **:directory-translations** and **:host-translation** (used together). If these last two attributes are given values on a logical host, they override any translations file specified for that logical host. For more information, see the **:directory-translations** attribute.

- **:image-printer-p** (P) — If the value of this attribute is non-**nil**, the particular printer can print bit-mapped graphics (pixel arrays) such as screen images. The default is **t**.

- **:ip-addr-subnet-bits** (H) — The implementation of subnetting requires some part of the local IP address to be dedicated to the network address providing a unique network address. The number of bits you enter is the number that will be appended to the host address. The **:ip-addr-subnet-bits** attribute is only used for gateways. Further information can be obtained in RFC950 by J. Mogul and J. Postel entitled *Internet Standard Subnetting Procedure*.

NOTE: Even though a host may have multiple IP addresses, it will not act as an IP gateway unless you specifically add the following as a group member to the **:services** attribute associated with that host:

```
(:gateway :ip :ip-gateway)
```

- **:known-classes** (N) — Specifies a standard set of Symbolics classes. This value is stored in the **name:*known-symbolics-classes*** variable. This attribute is only applicable to Symbolics namespaces.

- **:local-mail-domains** (H S U) — Identifies the valid mail domains for hosts at this site. This option is needed only for hosts in mail addresses that are not already recognized in the Name Server configuration. For instance, if this site was in the Internet domain "WIDGETS.ACME.COM", this option should have the following value:

  ```
  ("WIDGETS" "WIDGETS.ACME" "WIDGETS.ACME.COM").
  ```

- **:location** (H S U) — This attribute identifies the geographical location of the site that is, where the hardware for a host is located.

- **:machine-type** (H) — This attribute identifies the type of processor used by this host. For Explorers, the normal value of this attribute is **:explorer**.

- **:mail-address** (U) — The value of this attribute is the mail address at which a particular user receives electronic mail.

- **:mail-gateway-host** (H) — This attribute is the name of another host that should receive all mail destined for this particular host.

- **:medium-desirability** (H) — This attribute is an alist in the following form:

  ```
  ((medium1 X) (medium2 X) ...)
  ```

  This alist (where 0 <= X <= 1), is used to calculate the desirability of mediums.

- **:namespace-file-pathname** (N) — Specifies the default file where the namespace is stored. The default pathname is as follows:

  lm:name-service; namespace_name-namespace_type.xld

  For example, for a public namespace called cats, the default **:namespace-file-pathname** would be:

  lm:name-service;cats-public.xld

  Of course, you may override the default by specifying a different value for this attribute.

- **:namespace-search-list** (H S U) — This attribute is a list of all namespaces that should be searched (in order) whenever information is needed from a namespace.

- **:parity** (P) — This attribute identifies the number of parity bits per character used by the serial interface. Acceptable values for the **:parity** attribute include the following: **nil**, **:even**, or **:odd**. The default is **nil**, which is the default value for the **printer:*default-parity*** variable. Ignore this if the interface is not via the serial port.

- **:personal-name** (U) — The value of this attribute is the full name associated with the user *not just a user ID*. One use for this attribute is to generate the *From:* line of the mail system.

- **:port** (P) — This attribute identifies the specific I/O hardware port to which the printer is attached. This value can be 1, 2, or 3 for a serial printer. This value is **nil** for a parallel printer. The default value is **nil**.

- **:postmaster** (H S U) — This attribute identifies a user name or mail address of a person responsible for mail-related problems and queries.
- **:primary-device** (H) — The value of this attribute is a device name to be used as a default in pathnames for a particular host.
- **:primary-mail-servers** (H S U) — The value of this attribute is a list of host names that specifies which hosts will queue and forward mail.
- **:primary-servers** (N) — This attribute specifies the primary Symbolics servers used first for any type of request. Applicable to Symbolics namespaces only.
- **:primary-time-servers** (H S U) — The value of this attribute is a list of hosts that act as time servers. Time servers provide the time of day to all hosts on the network via the Chaos TIME protocol. You will need to specify which is the primary time server and which is the secondary time server in order of importance. If the first should fail, the second will provide the time.
- **:printer** (H S U) — The value of this attribute specifies the default printer to be used for printing of character files from a host.
- **:read-only** (N) — If the value of **:read-only** is **t**, no changes are allowed to be made to the namespace. This attribute is not applicable to a Symbolics namespace.
- **:reject-mail** (H) — Determines whether a host accepts mail. A value of **:non-local-addresses** means to accept only local mail and reject all mail destined for other hosts.
- **:remark** (M) — This attribute explains the purpose of a particular mailing list.
- **:secondary-servers** (N) — Specifies the secondary Symbolics servers used for reads only (not updates) if no primary servers respond. This attribute is applicable to Symbolics namespaces only.
- **:servers** (N) — This attribute is a list that identifies the hosts that will be co-servers for this namespace. It is suggested that you qualify the server names. For information about qualified and unqualified object names, see paragraph 4.2.4, Namespace Search Rules. If you do not qualify the names, the host in the current namespace is searched first. Then, search rules are applied.

NOTE: To describe one namespace (X) from within another namespace (Y), you should add a **:namespace** object that looks like the one contained in the actual namespace (X). Its function is to tell you how to access namespace X by listing the servers you can contact.

- **:service-desirability** (H) — An alist in the following form:

  ```
  ((service1 X) (service2 X) ...)
  ```

  This alist (where 0 <= X <= 1), is used to calculate the desirability of services.

- **:services** (H S U) — This attribute is a list of the services supported by this host. Services are identified by triplets of the form (*service medium protocol*), as in the following example:

  ```
  ((:FILE :LOCAL :LOCAL-FILE)
   (:STATUS :CHAOS :CHAOS-STATUS)
   (:MAIL-TO-USER :CHAOS-STREAM :MAIL)
   (:LOGIN :CHAOS-STREAM :TELNET)
   (:FILE :CHAOS :QFILE)
   (:STATUS :TCP :IP-STATUS)
   (:MAIL-TO-USER :TCP-STREAM :SMTP)
   (:LOGIN :TCP-STREAM :TELNET)
   (:FILE :TCP :FTP))
  ```

  NOTE: Even though a host may have multiple IP addresses, it will not act as an IP gateway unless you specifically add the following as a group member to the **:services** attribute associated with that host:

  ```
  (:gateway :ip :ip-gateway)
  ```

  More information about services is available in Section 6, The Generic Network Interface.

- **:short-name** (H) — Short name to be given this host. An alias named with the value of **:short-name** is created automatically by the Namespace Editor after editing this attribute.

- **:site** (H) — The SITE to use for a particular host. This is useful because the **net:get-site-option** function gets the attribute values from the **:site** object if the attributes are not specified by the **:user** object or the **:host** object.

- **:site-directory** (H) — This attribute identifies the directory to be used as the site directory for a particular host. If someone references the SITE directory on this host, the directory given as the **:site-directory** will be accessed.

- **:site-device** (H) — A string that represents the name of the storage device to use when someone references the site directory on a particular host.

- **:stop-bits** (P) — This attribute identifies the number of stop bits per character used by the serial interface. Acceptable values for the **:stop-bits** attribute include the following: 1, 1.5, or 2. The default is 1, which is the default value for the **printer:*default-stop-bits*** variable. Ignore this if the interface is not via the serial port.

- **:stream** (P) — This attribute identifies the type of I/O hardware port to which the printer is attached. Normal values include **:serial** and **:parallel**. The default is **:serial**, which is the default value for the **printer:*default-stream*** variable.

- **:string-for-printing** (H) — This attribute identifies a string to be used when printing the name of a particular host (as used in pathnames and so on). An alias named with the value of **:string-for-printing** is created automatically after editing this attribute.

- **:sys-host** (H S U) — This attribute is the name of the host (logical or physical) that will function as file server for the system files.

- **:system-type** (H) — This attribute identifies the operating system currently running on a particular host. For Explorer hosts, the value of this attribute is **:explorer**. For Symbolics hosts, the value of this attribute is **:symbolics**. For all other Lisp machines, the value is **:lispm**.

- **:terminal-f-arguments** (H S U) — The value of this attribute is a list of the finger assignments for the network in general. The format for the value of this attribute is as follows:

  ```
  ((0 :read))
  ```

  The TERM-*n*-F key sequence invokes the Finger utility, which displays information about users logged in at various machines in your network. This information includes their login name, full name, and so on. The site-wide values you supply here will be the defaults for all the individual hosts on the network, unless overridden by the hosts individually.

- **:timezone** (H S U) — The time zone attribute can be either an acronym for a time zone or a decimal integer representing the difference between your local time zone and Greenwich Mean Time. The default value of this attribute is the value returned by the **time:timezone-string** function (which should be the correct time zone for your area).

- **:top-level-mail-domain-servers** (H S U) — This attribute specifies top level mail domains (such as COM, EDU, GOV, and so on), as well as which hosts can relay mail to which domains. The attribute's value is stored as an association list; each alist item is of the following form:

  ```
  ("HOST" "DOMAIN1" "DOMAIN2" ...)
  ```

- **:translations-file** (H) — Pathname translations for a logical host can be loaded from the translations file specified by the value of **:translations-file**. A translations file can be used only on a logical host. Pathname translations for logical hosts (such as "SYS") can be specified in one of two ways:

  - From a translations file identified by the **:translations-file** attribute (which is discussed in this paragraph). This practice is discouraged.

  - From the values of the two attributes **:directory-translations** and **:host-translation** (used together). If these last two attributes are given values on a logical host, they override any translations file specified for that logical host. For more information, see the **:directory-translations** attribute.

Using a translations file is much slower than specifying the **:host-translation** and **:directory-translations** attributes in the namespace. When you access a logical host that uses a translations file, the translations file is loaded on your local machine (a time-consuming procedure) if either of these two conditions are true: This is the first time you are trying to access the logical host, or you try to access the logical host after the cached version of the host has been marked for refresh.

If you use the **:directory-translations** and **:host-translation** attributes to specify a logical host (the recommended procedure), there is no file to load, so access to logical hosts is generally faster. On the other hand, if the size of your network namespace is a problem, the use of translations files instead of the **:directory-translations** and **:host-translation** attributes would probably make your namespace smaller.

- **:type** (N) — This attribute identifies the type of a namespace (such as **:public** or **:personal**). The namespace type in turn determines the attributes associated with that namespace. All network namespaces should be of type **:public**. Public namespaces provide you with a cache; they are much faster at recording updates; and they can be loaded automatically when the machine is booted.

- **:type** (P) — This attribute identifies the type of a printer. The values associated with the **:type** attribute can be **:ti855**, **:ti880**, **:ti2015**, **:ti2115**, or **:imagen-printer**.

- **:usage** (N) — The **:usage** attribute governs the way the Namespace Editor edits a particular namespace. You can specify the usage mode corresponding to a set of expert editors. For information about the Namespace Editor, see the *Explorer Tools and Utilities* manual.

- **:use-primary-mail-servers** (H S U) — The value of this attribute specifies when hosts should forward mail to a primary mail server. Acceptable values for this attribute include the following:

  - **:never** — Each host performs complete mail delivery, including address verification, queuing, and retries.

  - **:always** — Hosts never attempt to complete mail delivery or even attempt to validate mail addresses. All mail is passed to the server for processing.

  - **:unknown-address** — Similar to **:never**, except that unknown mail addresses are passed to the primary mail server.

  - **:after-first-attempt** — Hosts validate and attempt to deliver mail; however, mail for hosts that do not respond is forwarded to the primary mail server.

- **:uucp-gateway-hosts** (H S U) — The value of this attribute is a list of hosts that will forward mail to the **uucp** (UNIX™-to-UNIX copy) network. An address is considered to be for the **uucp** network if it contains no "@" and it contains at least one "!".

- **:work-phone** (U) — The value of this attribute is the work phone number at which a particular user can normally be reached.

UNIX is a registered trademark of AT&T.

- **:xon-xoff** (P) — This attribute specifies whether or not xon/xoff protocol is used. If the value of **:xon-xoff** is **t**, the protocol is used; if **nil**, it is not used. The default is **t**, which is the default value for the **printer:*default-xon-xoff*** variable.

Multiple Network Namespaces

4.2.3 Multiple network namespaces can exist. If your machine is on more than one network namespace, you can specify which network namespace that your host will boot under by setting the **name:*default-who-am-i-domain*** variable to the name of your network namespace. For example, if the host name of your machine is BOLIVAR, and you want to be booted under a network namespace called NEW-NET#4, you would set **name:*default-who-am-i-domain*** to NEW-NET#4|BOLIVAR. To make this variable-binding permanent across boot sessions, you must be sure to perform a disk save.

Another way to specify which network namespace that your host will boot under is to use the disk label editor to modify the actual name of your disk so that it reflects both the host name and the network namespace name. Note that the total length of the disk name is limited to 16 characters; therefore, you may need to abbreviate the host name portion of the host/namespace name concatenation.

For example, when you concatenate NEW-NET#4, a vertical bar character (|), and BOLIVAR, you end up with a total of 17 characters. In this situation, you could drop the final R in BOLIVAR and change your disk's name to NEW-NET#4|BOLIVA. In this way, you comply with the 16 character limit. When BOLIVA(R) is rebooted, the booting process separates the host name and the network namespace name according to the vertical bar, and boots BOLIVA(R) under network namespace NEW-NET#4.

Multiple network namespaces can also be hierarchically structured. (That is, a network namespace can exist whose name is contained in another network namespace.) Names can be explicitly qualified to indicate the containing namespace, or defaults can be assumed.

Namespace Search Rules

4.2.4 When looking up or retrieving an object either from a single namespace or from multiple namespaces, the namespace utilities follow a set of search rules, which are discussed here.

The name of an object can be either *unqualified* or *qualified*. An unqualified (or relative) object name contains no namespace name in which to search for the object. In this situation, the namespace search rules state that the namespace utilities rely upon a *search list*. The search list contains a list of available namespaces. If you look up NEIL (an unqualified name), then the namespace utilities immediately check the namespace search list for available namespaces to search, and then search those namespaces *in the order that they appear on the search list*. The lookup operation stops either when NEIL is found, or when the namespace search list is exhausted.

A qualified object name contains the namespace name in which to search for the object. Only object names that are strings can be qualified. In a qualified object name, a vertical bar character (|) separates the namespace name (also called domain name), from the object name. When used in this context, the vertical bar is called the *domain delimiter*. If you look up AUSTIN|NEIL (a fully qualified name), the namespace utilities try to find the object NEIL in the namespace AUSTIN. In this situation the namespace search list is not used.

Network Namespaces, Servers, and Caches

4.2.5 One or more hosts can act as servers for a network namespace. These servers work in conjunction with each other across the network to handle concurrent access. An individual server is called a *co-server* (even if it is the only one).

A co-server can accept changes that you make to a network namespace, write these changes to the local copy of the network namespace, and also propagate these changes to other co-servers

When you edit a network namespace from a co-server, those changes exist only in a buffer until you write them out (either locally or globally). Writing the changes out locally modifies your co-server's *cache*.

A cache contains temporary copies of some of the objects found in the network namespace. One common use for modifying a co-server's network namespace cache is to test a new network configuration before broadcasting it to all the other network namespace co-servers.

The cache is cleared whenever its host is rebooted. To be able to save changes across booting sessions, you must write out any changes globally. Writing out changes globally also changes that network namespace *on all of its co-servers.*

Clients (that is, hosts that are not co-servers) also have caches. Normally, their caches only hold copies of the network namespace brought in at boot time. However, clients can edit their copies of a network namespace, and then write out those changes locally; thereby modifying their own caches, and altering their networking environments. When a client makes a global change, that change is first made in the client's cache, and is then updated globally at a co-server.

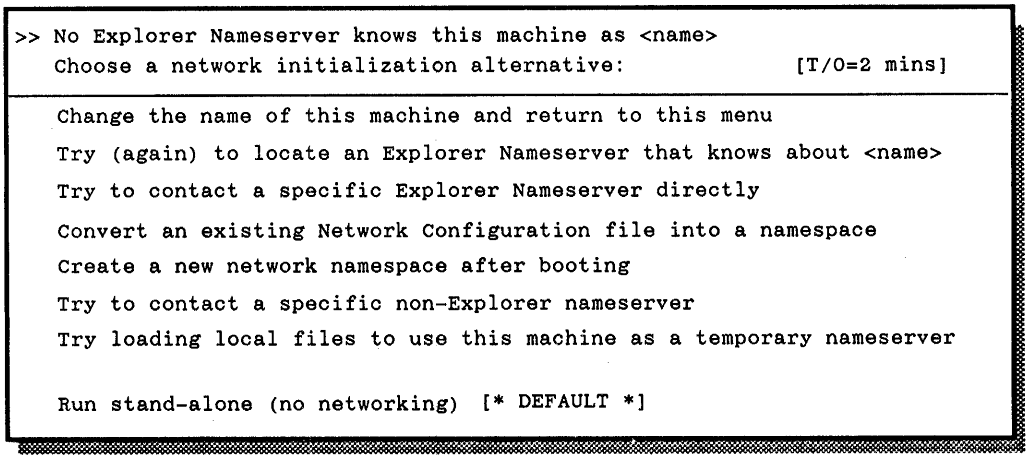
How to Update a Network From Release 2

4.3 If you have updated a site's Explorers from Release 2 to Release 3 software, you must convert the main network configuration server's configuration file (which is normally named lm:site;siteinfo.xfasl) into a network namespace.

Updating the First Host on the New Network

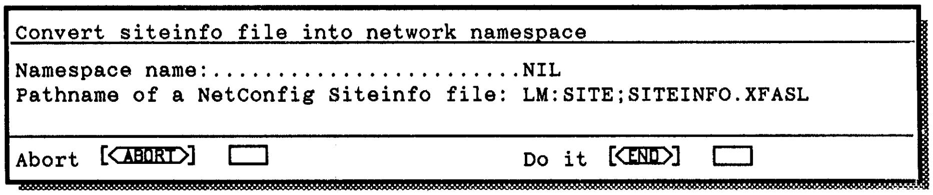
4.3.1 The first machine to receive the release software should have a current Release 2 configuration file (site;siteinfo.xfasl#>) on its file band. Although you can begin the update process from any machine on the network, unless that machine has its own copy of a Release 2 configuration file, you will need to supply the Chaosnet address of a machine that does have the needed configuration file. Normally, the machine you select should also be designated as a future network namespace co-server.

When the new co-server's software has been updated to Release 3 and it begins its initial boot under the new software, a network initialization menu appears, similar to the one following:



```
>> No Explorer Nameserver knows this machine as <name>
   Choose a network initialization alternative:           [T/O=2 mins]

   Change the name of this machine and return to this menu
   Try (again) to locate an Explorer Nameserver that knows about <name>
   Try to contact a specific Explorer Nameserver directly
   Convert an existing Network Configuration file into a namespace
   Create a new network namespace after booting
   Try to contact a specific non-Explorer nameserver
   Try loading local files to use this machine as a temporary nameserver

   Run stand-alone (no networking)  [* DEFAULT *]
```

To convert a Release 2 configuration file into a network namespace, click on the fourth item in the network initialization menu. The following window will appear:



```
Convert siteinfo file into network namespace
Namespace name:.......................NIL
Pathname of a NetConfig Siteinfo file: LM:SITE;SITEINFO.XFASL

Abort [<ABORT>] □                        Do it [<END>] □
```

Enter a name for your new network namespace (one that uniquely identifies your local site) and press RETURN. If your siteinfo file name differs from the default value of lm:site;siteinfo.xfasl, change that value and press RETURN. When both values are acceptable, click on the `Do it` prompt or press END. (If you choose the `Abort` option to the Convert Siteinfo menu, you are simply returned to the network initialization menu.)

If you specify a remote host's pathname as the source of the Release 2 configuration file, another menu will appear, requesting a network address for that host (normally a Chaosnet or IP address). You must supply those values before continuing.

After you have identified a source for the Release 2 configuration file, your local host loads that file and begins the translation process. You may see the configuration file's name appear in the lower right of the status line followed by a figure showing how much of the file has been loaded. This figure will be either a byte count or a percentage value.

Next, if you choose to save the namespace (you are prompted to do so), the translation is written into a new namespace file with the name: lm:name-service;<name>.xld#>. You may see the new file's name appear in the lower right of the status line, followed by a figure showing how much of the file has been written. This figure will be either a byte count or a percentage value.

When the new network namespace file has been written, the initial boot sequence completes. If your local host is not one of the designated network namespace co-servers, it will act as a temporary co-server until you can boot a real one.

Updating the Other Hosts on the Network

4.3.2 With a co-server (either temporary or permanent) up and running with a network namespace loaded, you can update the rest of the hosts on the network. As each of the other hosts is updated to Release 3 software and begins its initial boot sequence under that software, it attempts to load any namespace(s) for which it is a co-server.

At this time, the newly booted host will have no namespaces associated with it; therefore it issues a `who-am-i` message which is picked up by the co-server for which you just converted the Release 2 configuration file. The co-server then sends the necessary namespace information to the local host. After receiving the namespace information, the local host completes its initial boot sequence. From this point on, the local host has network access. Repeat this process for the rest of the hosts on the network.

NOTE: If the first host you booted was not designated as a network namespace co-server, it will continue to act as one (temporarily) until the first host actually designated as a co-server has acquired its own copy of the new network namespace. At that point, the actual co-server recognizes that it is a designated co-server, completes its initial boot sequence, and then sets itself up as a namespace co-server.

Optional Method for Updating Other Designated Co-Servers

4.3.3 If you followed the previous suggestion and booted a designated network namespace co-server as the first machine to run under Release 3 software, the other co-servers can get their copies of the new network namespace in one of two ways. The first way was described in the previous paragraph.

You also have the option of preparing future co-servers while they are running under Release 2 software. In this method, you first create a name-service directory on the future co-server, and then copy the following files from the host having its namespace already created:

<*host-name*>:name-service;server-boot-list.lisp#>
<*host-name*>:name-service;<*namespace-name*>-public.xld#>
<*host-name*>:name-service;<*namespace-name*>-log.text#>

With these files in place, as soon as the new co-server is booted under Release 3 software, it automatically boots the network namespace available from those files, and sets up itself as a namespace co-server.

Updating an Existing Namespace

4.4 If you already have an existing network namespace, and must modify it in some manner (adding a host or a printer for example), you need only edit that network namespace. Editing a network namespace is no different from editing any namespace. If you are unfamiliar with the operation of the Namespace Editor, see the *Explorer Tools and Utilities* manual.

After you have made the required changes to a network namespace, first write out the changes locally so that you can test them before distributing them to the rest of the network.

After you are sure that the changes are stable, write them out globally. Before the Namespace Editor will write any changes locally or globally, it performs an *incremental verification* to insure that your changes are consistent. An incremental verification checks only those objects that have been edited to see if they are valid and consistent with the rest of the namespace.

Different verification levels are available, according to the value of the **nse:*verification-level*** variable (discussed in paragraph 4.8, titled Network Namespace Functions and Variables).

Several Namespace Editor commands are available for performing verification of changes to a network namespace:

Verify Namespace Command
Keystroke: META-X Verify Namespace

Verifies objects and associated attributes within a namespace. The verification routine(s) that is called by this command is determined by the values of the **:namespace-verification-routine** keyword of the expert editors. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to the *Explorer Tools and Utilities* manual for more details.

Verify Class Incrementally Command
Keystroke: META-X Verify Class Incrementally

Verifies objects within the current class. The verification routine(s) that is called by this command is determined by the value of the **:incremental-verification-routine** keyword of the expert editor. (Verification routines have already been established for network namespaces.) Refer to the *Explorer Tools and Utilities* manual for more details.

Verify Object Incrementally Command
Keystroke: META-X Verify Object Incrementally

Verifies the current object. If verification fails, you are warned that the object content is not correct. The verification routine(s) that is called by this command is determined by the value of the **:incremental-verification-routine** keyword of the expert editor. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to the *Explorer Tools and Utilities* manual for more details.

Verify Attribute Incrementally Command
Keystroke: META-X Verify Attribute Incrementally

Verifies the value of the current attribute. If verification fails, you are warned that the attribute value is invalid. The verification routine(s) that is called by this command is determined by the value of the **:incremental-verification-routine** keyword of the expert editor. Verification routines have already been established for network namespaces. However, other namespaces do not have expert editors with verification routines unless someone has created them. Refer to the *Explorer Tools and Utilities* manual for more details.

A Namespace Editor command and a function are available for distributing a network namespace:

Distribute Namespace Command
Keystroke: META-X Distribute Namespace

Distributes the current namespace to all servers or to the local machine only. If necessary, you are asked to save the namespace before distributing. (This command calls the **name:distribute-namespace** function.)

name:distribute-namespace *namespace-name* Function
&optional &key **:local-only** (**:save-first t**) **:server-list**
(**:search-list-loc :beginning**) (**:notify t**)

Terminates configure or convert mode for a new public namespace. These modes are described in the **:explorer-server** *init-args* of the **name:add-namespace** function in the *Explorer Tools and Utilities* manual.

You can optionally dump the namespace to a binary file. You should do this if you are in convert mode because no changes have been written to the log file.

Propagating to servers of subsequent changes is enabled.

A local client instance is created, and all further namespace accesses at this machine now go through the client instance rather than directly to the server instance.

Optionally, you can start other servers, which obtain their information from this host. This works only if those servers are already on the network.

:local-only — When set to **t**, distributes only to this host. The default is **nil**.

:save-first — When set to **t** (the default), dumps the namespace to an xld file first.

:server-list — Specifies a list of servers to receive distribution. The default is all servers.

:search-list-loc — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil**.

:notify — When set to **t** (the default), notifies you if some servers did not answer.

Building a Network From Scratch

4.5 Once you have assembled all the Explorers and their related hardware for a new network, you must place all the network configuration information into a network namespace. The network namespace provides information about the relationships between the names and addresses of hosts and servers (printer servers, time servers, and so on) that are available on a network. Every host on the network uses such information to find paths to other hosts and to the services provided over the network.

You will follow five steps in creating a new network.

- Plan the network.
- Create the network namespace, using the Namespace Editor utility.
- Verify the network namespace.
- Distribute the network namespace.
- Reboot all the hosts on the network.

Plan the Network Namespace

4.5.1 The first step in getting on a network requires planning its namespace. You should have all the information about the future network *written down* before attempting create your network namespace. This information concerns the characteristics of the following components of the network:

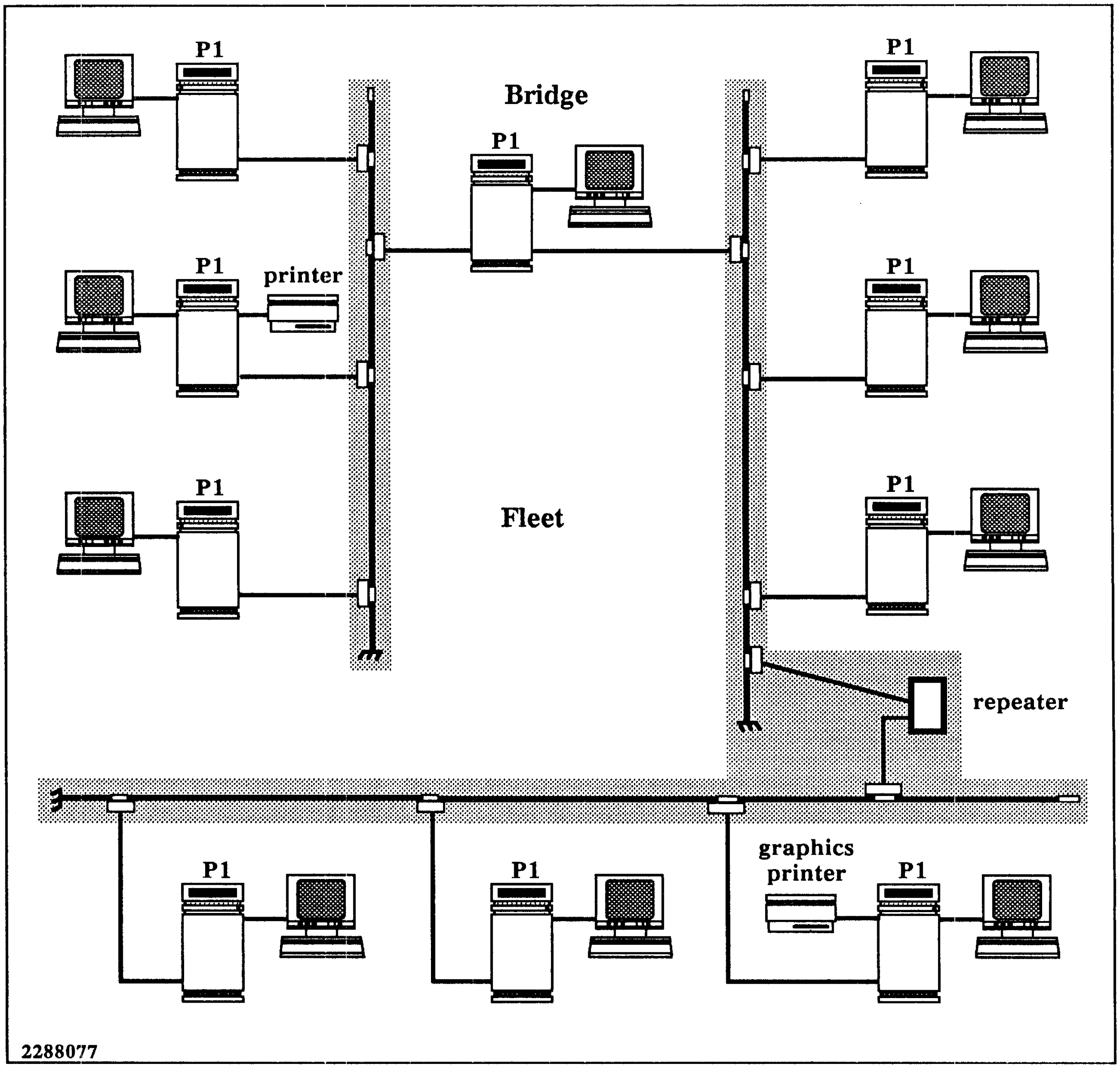
- Site — The site is the entire network. Site characteristics affect all components of the network.
- Hosts — Hosts are the individual machines on the network. They inherit some characteristics from the site (such as location), but they also have individual characteristics (such as their network addresses).
- Printers — Printers are also individual components of a network.
- Namespace — The namespace characteristics affect how the network maintains its information about itself.

The network namespace is a hierarchical arrangement of network information. For example, it treats each of the preceding network components as a *class*. Within each class are specific *objects*. The objects represent actual entities on the network, such as a specific host or printer. Within each object are *attributes* that are associated with that object. Some of the host attributes include address and machine type. Finally, attributes have associated values; for instance, a machine type attribute may have the value **:explorer**.

Based on the attributes and values discussed in paragraph 4.2, titled The Network Namespace, gather all the specific information about your network's component classes. Be sure to write all of this information in one place for easy access later. When you have collected all this information, you can proceed to paragraph 4.5.2, titled Create the Network Namespace.

In the example that runs throughout these paragraphs, you will deal with a site named FLEET which is located in Galveston. FLEET's initial configuration is seen in Figure 4-1. All machines have the default factory-shipped name (P1) at this point, and FLEET uses the Chaosnet protocol. You know nothing more about the site.

Figure 4-1 Site: FLEET

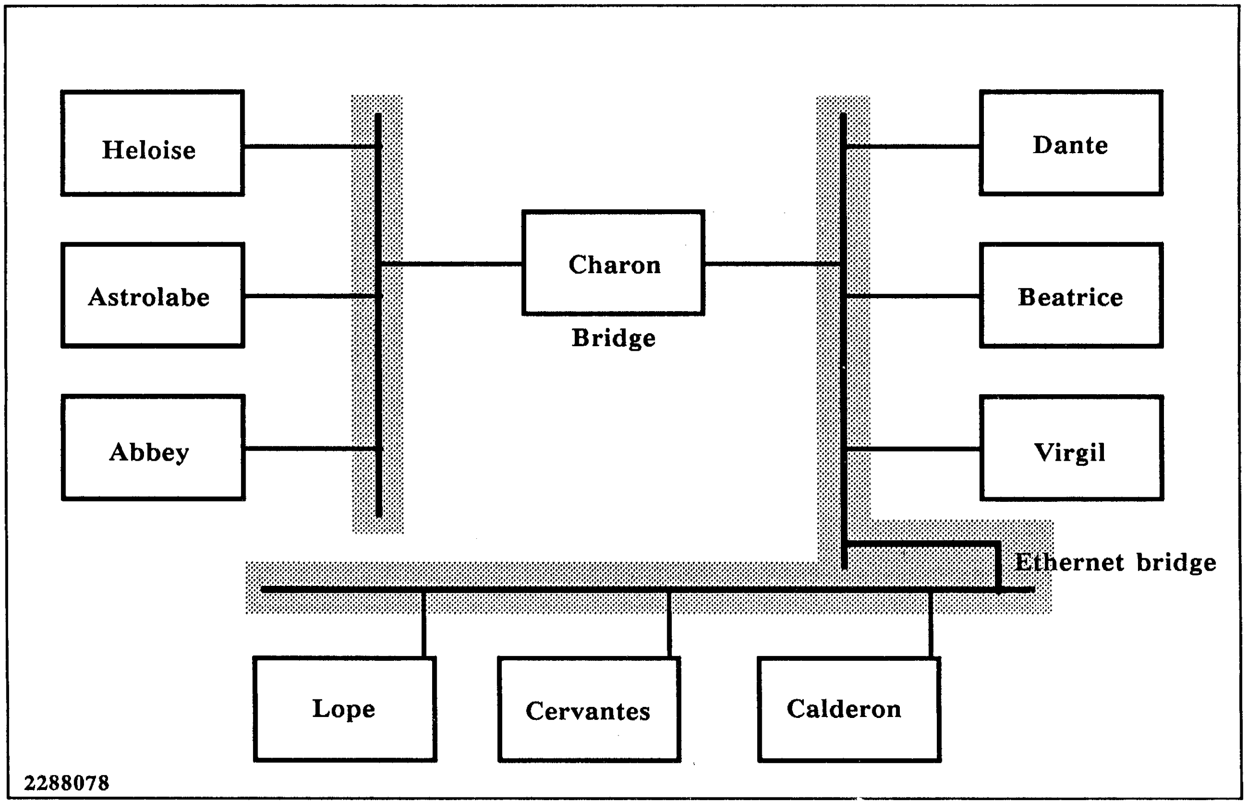


Site Diagram and Site Characteristics

4.5.1.1 Taking the top-down approach, you should first plan the details about the site (FLEET in this example). Site planning requires some pencil work.

Make a diagram of your network, and label each Explorer with its future name. With the future names assigned, each host is easily distinguishable (unlike when dealing with a network full of P1's). Figure 4-2 shows FLEET with arbitrarily chosen names for each of its hosts.

Figure 4-2 Site FLEET (With Future Host Names)



Now that you have a sketch of the future network site, you must collect information about the site as a whole. As stated earlier, you need a table of information about the site characteristics. Eventually, you will enter your site's characteristics as attributes for a site object in your network namespace.

Remember that site attributes are described in paragraph 4.2.2, titled Network Namespace Attributes. Table 4-1 , which follows, shows how FLEET's network administrator gathered the information about FLEET.

Table 4-1 **Known Site Characteristics of Site FLEET**

Characteristic or Attribute	Value
Location	Galveston
Time Zone	Central Daylight Savings Time
Default File Server	lm
System Host	Cervantes
Host for Bug Reports	Abbey
Character Printer	Astrolabe
Graphics Printer	Calderon
Primary Time Servers	Heloise, Dante
Primary Mail Server	Beatrice
Terminal F Args	Accept defaults
Namespace Search List	Accept defaults

Host Characteristics

4.5.1.2 Now that you have all the characteristics of your site tabulated, you must do the same thing for each host that will reside on the network. A little more work is required to get all the host attributes.

Host attributes are described in paragraph 4.2.2, titled Network Namespace Attributes. The five host attributes you must gather for each host include the host's aliases, network address(es), the services provided by that host, and the host's machine type and system type.

Aliases In the FLEET example, the network administrator assigned the following aliases for the hosts in FLEET.

Heloise	He
Astrolabe	Astro
Abbey	Ab
Charon	Ch
Dante	Dan
Beatrice	Bea
Virgil	V1
Lope	Lp
Cervantes	CV
Calderon	Cal

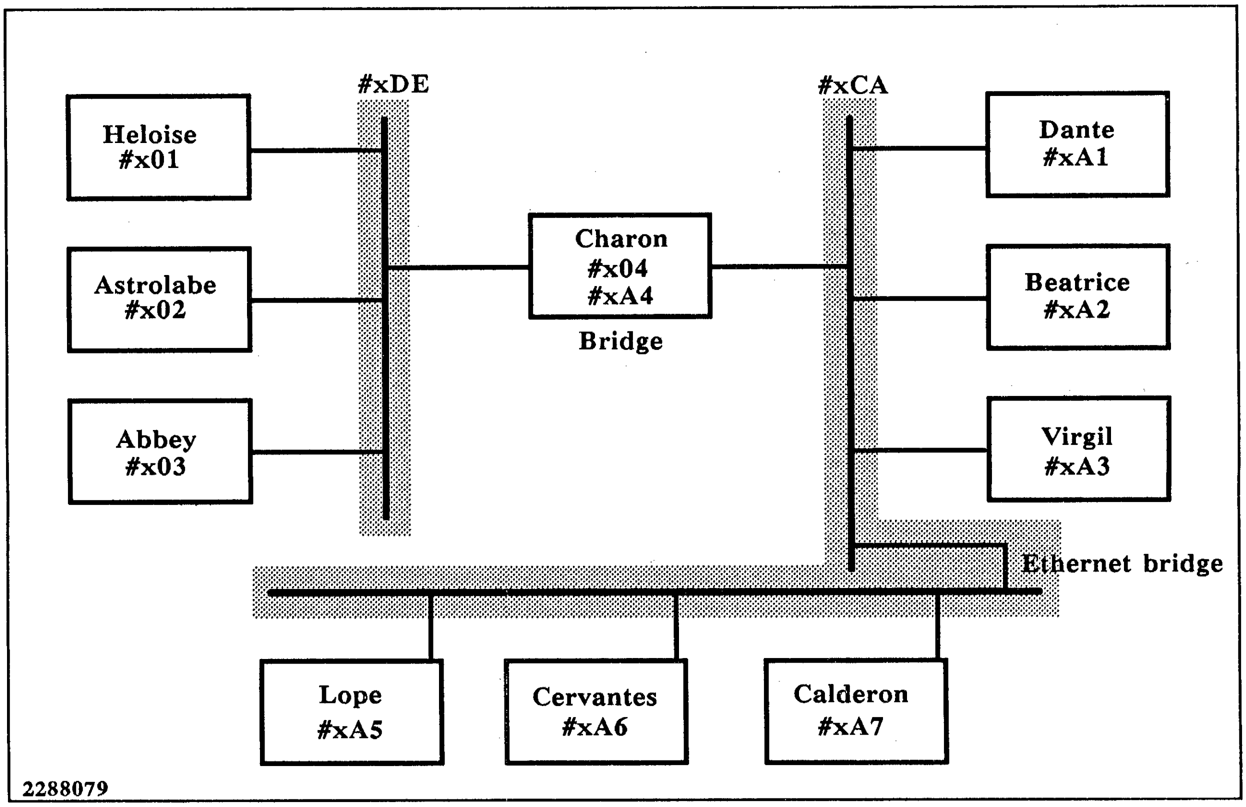
Addresses To assign addresses for each host on a network, you must identify any subnetworks and assign numbers to them. For example, FLEET consists of two subnets. In Figure 4-1 and Figure 4-2, FLEET's subnetwork topography is distinguished by shading. Notice that one host (CHARON) acts as a bridge between the two subnets, passing data from one subnet to the other. As a bridge, that host is a member of *both* subnets.

Each subnet must be assigned a *Chaosnet subnetwork number*. This number is an 8-bit hexadecimal number in the range #x01 through #xFE. The network administrator chooses Chaosnet subnetwork addresses arbitrarily. In the FLEET example, address #xDE is assigned to the top left subnet, and #xCA to the right subnet. Note that the subsegment at the bottom also has the Chaos subnetwork number #xCA, since a repeater connects the two subsegments.

Each host must also be assigned a Chaosnet *host address*. A Chaosnet host address is also an 8-bit hexadecimal number in the range #x01 through #xFE. The network administrator can choose these numbers arbitrarily. As with subnetwork numbers, a bridge must be assigned two host addresses.

Figure 4-3 shows the FLEET site with arbitrarily assigned Chaosnet subnetwork numbers and host addresses.

Figure 4-3 Site FLEET (With Chaosnet Subnetwork Numbers and Host Addresses)



Chaosnet addresses for each host on a network are constructed from the concatenation of Chaosnet subnet numbers and host addresses. For example, DANTE on subnet #xCA has the host address #xA1. Therefore, DANTE's Chaosnet address is #xCAA1. HELOISE, on the other hand, is on subnet #xDE and has the host address #x01. Her Chaosnet address is therefore #xDE01. Remember that CHARON is a bridge. One of this machine's subnetwork numbers is #xDE and the other is #xCA. We arbitrarily assign CHARON the two host addresses #x04 and #xA4. Concatenate one of these host addresses, say #x04 to subnet number #xDE, and the other to subnet number #xCA. CHARON has Chaosnet addresses #xDE04 and #xCAA4.

Services At this point you must determine which network services are to be rendered by which hosts, if you have not already done so. The hosts in site FLEET have only the default services. Services are discussed in detail in Section 6, The Generic Network System.

Machine Type and System Type All of the hosts in site FLEET are Explorers. Therefore, their machine type and system type attributes will all be **:explorer**.

After gathering all of the information about each of the hosts on your network, tabulate the information for easy reference. Table 4-2 is a table of host information for site FLEET. This table will be used as a reference during the creation of the network namespace. The present network configuration represented in this table is shown also in Figure 4-3.

Table 4-2 Known Host Characteristics of Site FLEET

Host Name	Aliases	Network Address(es)	Services	Machine Type	System Type
Heloise	He	#xDE01	default	:explorer	:explorer
Astrolabe	Ast	#xDE02	default	:explorer	:explorer
Abbey	Ab	#xDE03	default	:explorer	:explorer
Charon	Ch	#xDE04, #xCAA4	default	:explorer	:explorer
Dante	Dan	#xCAA1	default	:explorer	:explorer
Beatrice	Bea	#xCAA2	default	:explorer	:explorer
Virgil	V1	#xCAA3	default	:explorer	:explorer
Lope	LOP	#xCAA5	default	:explorer	:explorer
Cervantes	CV	#xCAA6	default	:explorer	:explorer
Calderon	Cal	#xCAA7	default	:explorer	:explorer

Printer Characteristics

4.5.1.3 Repeat the process of collecting and tabulating information; this time for each of the printers on your network. To know what information is vital about a printer, see the printer attributes discussed in paragraph 4.2.2, titled Network Namespace Attributes.

Namespace Characteristics

4.5.1.4 For planning purposes, you need to know only one characteristic for the future namespace you will be creating; that is, which machines will act as the network namespace's servers (actually called so-servers).

Network namespace co-servers maintain a usable copy of the configuration information that you have just gathered. Whenever a client boots, it sends a *who-am-i* message (containing the host address) out on the network. If a co-server is available, it will respond to the *who-am-i* message by sending the necessary network namespace information to the newly booted host, thereby enabling network services for that host.

In the FLEET example, VIRGIL will act as the only network namespace co-server for FLEET.

Create the Network Namespace

4.5.2 Now that you have planned your network configuration and have both a site diagram and a table of all the known network characteristics, you are ready to perform the initial boot sequence for the machine you have chosen to be the network namespace server (Virgil for FLEET). When that machine is first booted, you will have an option of creating a network namespace.

Each new Explorer system is shipped with a load band containing a boot network namespace that contains only enough information to boot the new system as a standalone unit. From the factory, all new Explorer units are named P1. You will change this name during the initial boot sequence.

Sequence of Events During Initial Boot

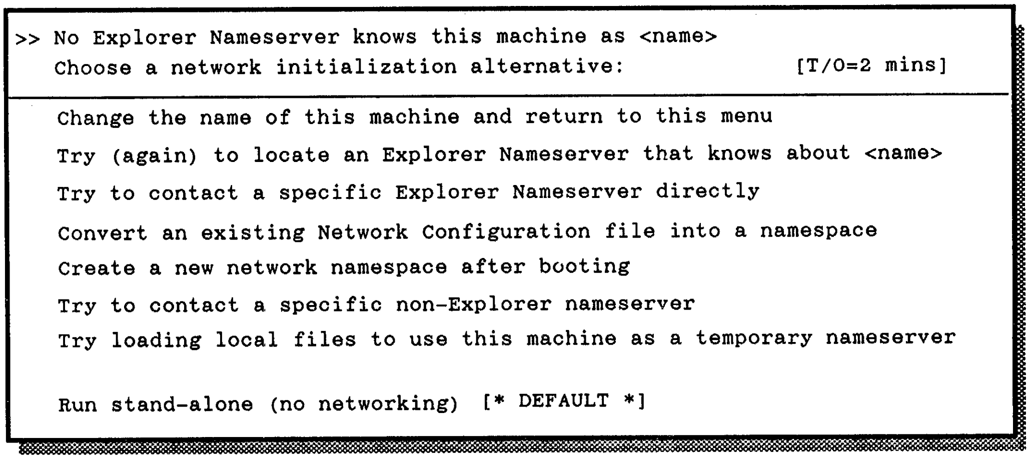
4.5.2.1 When you first boot a new Explorer, it runs a series of self-tests on its hardware. If everything passes these tests, you are prompted to specify what kind of boot sequence is to take place. You will take the defaults.

After you take the default value specifying the type of boot sequence, the machine immediately loads all of the namespaces for which it acts as a co-server. A factory-shipped machine has no such namespaces; therefore, it proceeds to the next step in the initial boot sequence.

The next step a new machine takes during the initial boot sequence is to broadcast a *who-am-i* message to the network. The who-am-i message contains the host's name (from its disk-pack), and the host's address.

Even though the physical portions of the new network are connected, no hosts on the net contain a network namespace, and therefore, no one can respond to a who-am-i message. (Besides, it is doubtful that any namespace would recognize a host called P1.) With no network namespace available from an external host, the machine proceeds to the next step of the initial boot sequence.

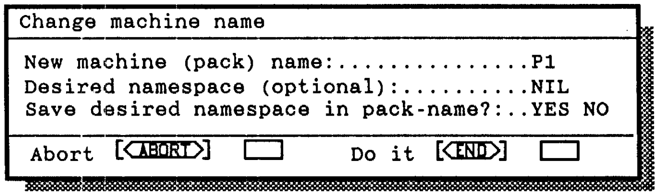
At this point, a network initialization menu appears, similar to the one following:



```
>> No Explorer Nameserver knows this machine as <name>
   Choose a network initialization alternative:            [T/O=2 mins]

   Change the name of this machine and return to this menu
   Try (again) to locate an Explorer Nameserver that knows about <name>
   Try to contact a specific Explorer Nameserver directly
   Convert an existing Network Configuration file into a namespace
   Create a new network namespace after booting
   Try to contact a specific non-Explorer nameserver
   Try loading local files to use this machine as a temporary nameserver

   Run stand-alone (no networking)  [* DEFAULT *]
```

Rename Your Network Namespace Co-Server

4.5.2.2 The first step in preparing the new network configuration for site FLEET is to rename the network namespace server host. Click on the Change the name of this machine and return to this menu prompt. The following pop-up menu appears when you click on this menu item.



Change machine name

```
New machine (pack) name:...............P1
Desired namespace (optional):..........NIL
Save desired namespace in pack-name?:..YES NO
```

Change P1 to the name of your network namespace co-server (VIRGIL for FLEET). Accept the default value of nil for the Desired namespace (optional prompt. (This prompt lets you specify a default network namespace to be used with this host.) Click on the NO value for the Save desired namespace in pack-name prompt.

Your machine has now been properly named, and the network initialization menu reappears.

Completing the Initial Boot

4.5.2.3 Now click on Create a new network configuration after booting. After you click on this option, booting completes, and the following message appears at the top of the print herald:

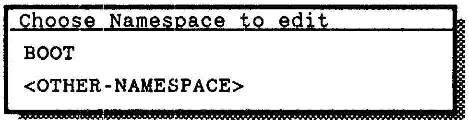
```
*** To create a new network namespace after booting has completed,
    select the Namespace Editor from the System Menu.
```

Log in to your new Explorer, specifying 'lm for the host name, and t to indicate that no login-initialization file should be loaded, as in the following example:

```
(login 'roger 'lm t)
```

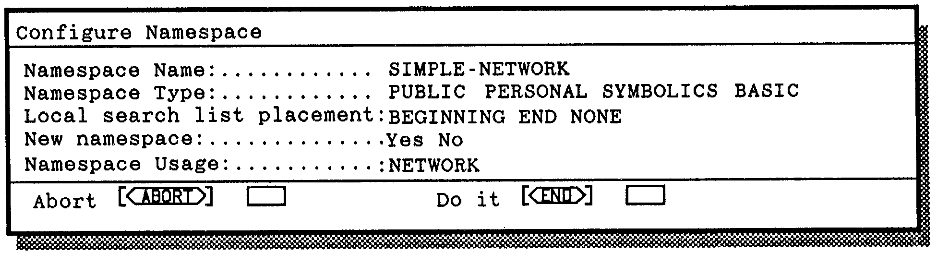
Bring Up the Namespace Editor

4.5.2.4 When you have successfully logged in, click right to bring up the system menu, then select the Namespace Editor option under the Programs column. Most of the information you need to use the editor is contained in this section. If, however, you need more information, see the *Explorer Tools and Utilities* manual. The following pop-up menu appears:



NOTE: The BOOT namespace is a memory-resident-only namespace that allows an Explorer to boot standalone. Although you can edit the BOOT namespace, you cannot save any changes that you make. The changes are in effect only until the machine is rebooted.

Click on the <OTHER-NAMESPACE> option. A prompt appears in the minibuffer, requesting the name of a namespace to edit. Enter a name and press RETURN. At this point a display appears, similar to the one following:



```
Configure Namespace
 Namespace Name:............ SIMPLE-NETWORK
 Namespace Type:............ PUBLIC PERSONAL SYMBOLICS BASIC
 Local search list placement:BEGINNING END NONE
 New namespace:..............Yes No
 Namespace Usage:...........:NETWORK
  Abort [<ABORT>] □              Do it [<END>] □
```

Namespace Name Move the cursor to the end of the dotted lines by the Namespace Name prompt, and click left. Enter the name by which your new network namespace will be known (SIMPLE-NETWORK in this example).

Namespace Type All network namespaces should be public namespaces so that the information on the network namespace is accessible to everyone on the network. Choose the PUBLIC option (which is the default).

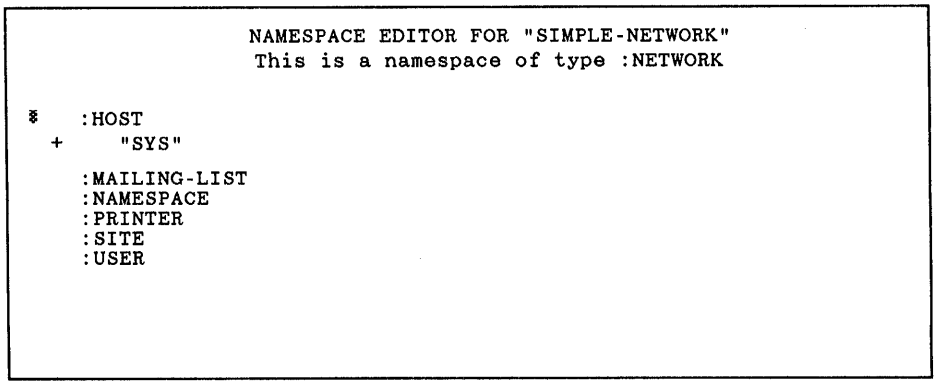
Local Search List Placement Your network namespace resides on a search list stored in the **name:*namespace-search-list*** variable. When you act as a server for several namespaces, they will all be on this list. It is possible for a host to be defined in more than one namespace.

The `Local search list placement:` prompt allows you to specify where your new network namespace is placed on the search list. The first namespace in the search list is searched first for the host. If the host is not found on that namespace, then the second namespace is searched, and so on, until the host is found. In the creation of SIMPLE-NETWORK, which is new, you choose the BEGINNING option.

New Namespace Your network is new, (and so is FLEET); therefore choose the YES option at this point. (The default is NO.)

Namespace Usage The default value is **:network**. Since you are setting up a network namespace, you will accept the default.

Click on Do It or press the END key to signal that you have completed entering information in the Configure Namespace menu. At this point, your Explorer enters the Namespace Editor, and displays the Namespace Editor buffer on the screen:



```
                 NAMESPACE EDITOR FOR "SIMPLE-NETWORK"
                  This is a namespace of type :NETWORK

 *   :HOST
   +    "SYS"

     :MAILING-LIST
     :NAMESPACE
     :PRINTER
     :SITE
     :USER
```

NOTE: The following paragraphs tell you how to create objects for each of the classes needed by the network namespace. The paragraphs also tell how to change the default attribute values for each of those objects.

If you have questions about a class, object, or attribute, you can refer to paragraph 4.2, titled The Network Namespace, or you can move the editing cursor next to the item for which you need information, and then press CTRL-SHIFT-D to see the online documentation for that item.

While this manual discusses the creation and editing of the different network namespace objects in a certain order, you are not restricted to that same order. For example, you can prepare your **:site** class object first and your **:host** class objects last, if you so desire. The only restriction to this procedure is that you create the namespace on a machine that you have designated as a network namespace co-server.

Create the Host Objects

4.5.2.5 The next step in creating a network namespace is to create host objects for every host that will reside on the new network. The hosts must be added one at a time, *beginning with the host that is being used to create the namespace*. In the FLEET example, VIRGIL is used for this purpose, and VIRGIL will remain FLEET's network namespace co-server in the future.

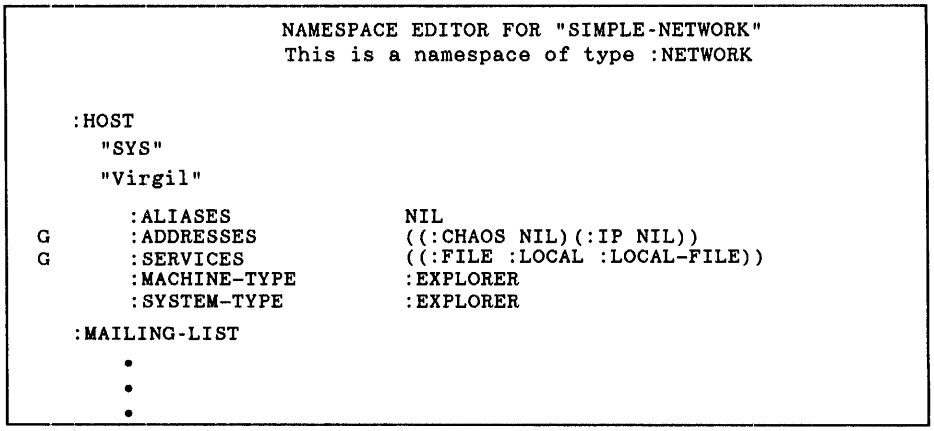
Remember, if you have questions about a class, object, or attribute, you can refer to paragraph 4.2, titled The Network Namespace, or you can move the editing cursor next to the item for which you need information, and then press CTRL-SHIFT-D to see the online documentation for that item.

1. Place the mouse cursor box around the class name :HOST and click middle on the mouse. A menu of commands appears.

2. Click on the `Add Object` command. A prompt appears in the minibuffer, requesting you to enter the class of the new object.

3. Be sure that the class for the new object is :HOST, and press RETURN. You are now prompted to enter the name of the new **:host** object.

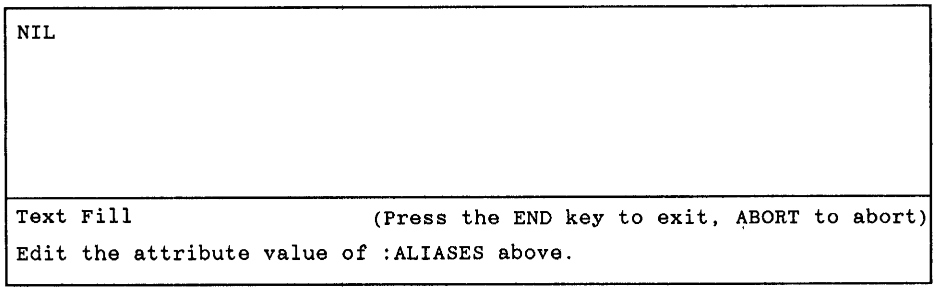
4. Enter the name of your host. For FLEET, we enter `Virgil`. A display will appear, similar to the one following, showing the default attributes (and values) of the new host.



```
                         NAMESPACE EDITOR FOR "SIMPLE-NETWORK"
                          This is a namespace of type :NETWORK


     :HOST
       "SYS"
       "Virgil"

         :ALIASES             NIL
  G      :ADDRESSES           ((:CHAOS NIL)(:IP NIL))
  G      :SERVICES            ((:FILE :LOCAL :LOCAL-FILE))
         :MACHINE-TYPE        :EXPLORER
         :SYSTEM-TYPE         :EXPLORER

     :MAILING-LIST
         •
         •
         •
```

5. Currently, the default value of the :ALIASES attribute is **nil**, meaning that the host (VIRGIL) has no aliases. To change the **nil** value so that the host has an alias, move the mouse cursor box around the :ALIASES attribute, and click middle. A menu of commands appears.

6. Click on the `Edit Attribute` command. The following edit window appears:

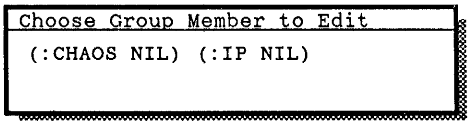


```
NIL

Text Fill                (Press the END key to exit, ABORT to abort)
Edit the attribute value of :ALIASES above.
```

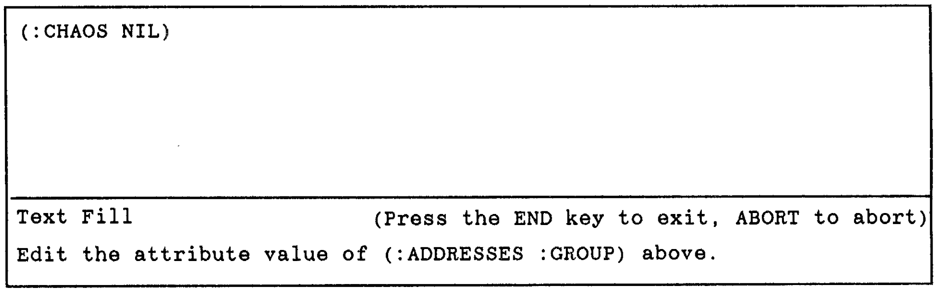
7. Delete NIL, using standard Zmacs key functions (such as CTRL-D for delete), and replace NIL with your host's alias ("V1" for VIRGIL). Press the END key (*not* RETURN) to signal completion of the new value for the **:aliases** attribute. Now "V1" replaces NIL as the value of :ALIASES, and a host object for "V1" is automatically added to your display.

8. You are now ready to enter the address(es) for this host. Place the mouse cursor box around the attribute name :ADDRESSES and click middle on the mouse. Because :ADDRESSES is a group attribute, its value is a list of `(network-type address)` pairs. Accordingly, a slightly different menu appears.

9. Click on the `Edit Group Member` command. The following pop-up menu appears:



10. Click on the appropriate entry to change an address from **nil**. For VIRGIL, only one type of address (Chaosnet) is available. Therefore we click on the (:CHAOS NIL) entry. The following edit window appears:



```
(:CHAOS NIL)

Text Fill                (Press the END key to exit, ABORT to abort)
Edit the attribute value of (:ADDRESSES :GROUP) above.
```

11. Edit the value just as you did in step 7, previously. This time add the correct Chaosnet address for your host, maintaining the list format presented as the value of the group attribute `:ADDRESSES`. After entering a Chaosnet address, the list of `:SERVICES` is automatically updated to include the default Chaos services.

NOTE: If you include non-numeric characters in the `(network-type address)` pair (such as periods to indicate dotted decimal format), you must enclose the `address` portion of the `(network-type address)` pair in quotation marks. For example: `(:IP "101.001.001.747")`

This action does not apply to the #x that indicates a hexadecimal address.

---

12. If your host has other network protocols (such as IP), you must repeat steps 8 through 11 for each protocol's address that you need to add for your host.

13. When you have entered correct values for all the possible network addresses of your first host, you are ready to add any services provided the particular host. Services are discussed in Section 6, The Generic Network System. Normally, the default services are all that are needed in a new network. If you must add a service for this host, perform the following steps:

    a. Move the mouse cursor next to the `:SERVICES` attribute and click middle. A command menu appears.

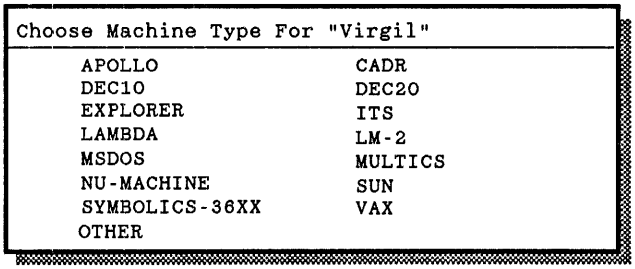
    b. Now click on the `Add a Group Member` command. A prompt appears in the minibuffer, requesting you to enter the new group member.

    c. Add the new group member and press RETURN.

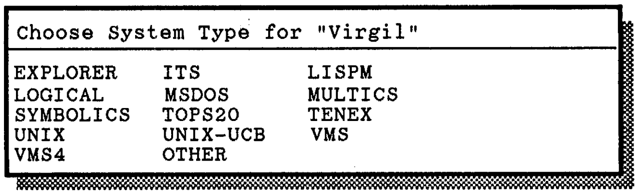
    VIRGIL provides only local file service and the default Chaos services, so FLEET skips step 13.

14. Unless you have a non-Explorer host, you can accepts the default values provided for the next 2 attributes (`:MACHINE-TYPE` and `:SYSTEM-TYPE`). Should you ever need to change the value of these attributes, you can do so by using the standard Namespace Editor commands as previously presented.

    The following pop-up menu identifies the choices you could select from if you needed to change the value of the `:MACHINE-TYPE` attribute:



The following pop-up menu identifies the choices you could select from if you needed to change the value of the :SYSTEM-TYPE attribute:



```
Choose System Type for "Virgil"

EXPLORER    ITS        LISPM
LOGICAL     MSDOS      MULTICS
SYMBOLICS   TOPS20     TENEX
UNIX        UNIX-UCB   VMS
VMS4        OTHER
```

15. Repeat steps 1 through 14 to create a **:host**-class object for every machine on your network. FLEET would need to create 10 new objects in the **:host** class.

Identify the Namespace Servers

4.5.2.6 For a new network, only one possible change needs to be made to the **:namespace** class. You must identify any machines that will be co-servers for the network namespace *other than the machine on which you are creating the network namespace*. If you have only the one co-server for your network namespace, you can skip the following 6 steps.

Remember, if you have questions about a class, object, or attribute, you can refer to paragraph 4.2, titled The Network Namespace, or you can move the editing cursor next to the item for which you need information, and then press CTRL-SHIFT-D to see the online documentation for that item.

1. Place the mouse cursor box around the class name :NAMESPACE and click middle on the mouse. A menu of commands appears.

2. Click on the `Expand All Objects in Class` command. The :NAMESPACE class then expands showing all of its default attributes. Notice that the value of the :SERVERS attribute is a list that already has the disk-pack name of the host on which you are working.

3. Place the mouse cursor box around the :SERVERS attribute, and click middle. A menu of commands appears.

4. Click on the `Add a Group` Member command. A prompt appears in the minibuffer, requesting that you enter the new group member.

5. Enter the name of a host that will be a network namespace co-server, and press RETURN.

6. Repeat steps 1 through 5 for any other hosts that will be acting as network namespace co-servers.

Create the Printer Objects

4.5.2.7 Having added the necessary values for all the **:host** and **:namespace** objects, you must now identify your printers to the network.

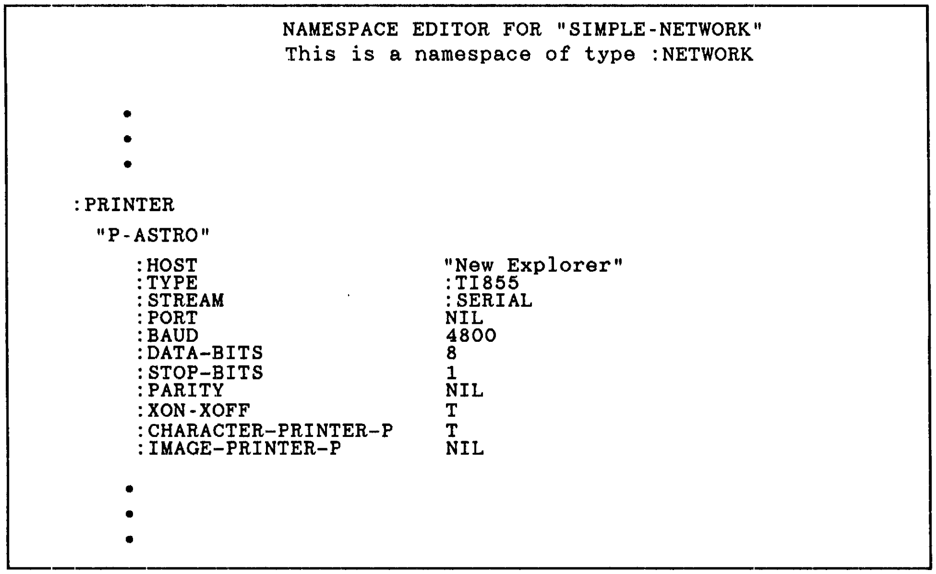
Remember, if you have questions about a class, object, or attribute, you can refer to paragraph 4.2, titled The Network Namespace, or you can move the editing cursor next to the item for which you need information, and then press CTRL-SHIFT-D to see the online documentation for that item.

1. Place the mouse cursor box around the class name :PRINTER and click middle on the mouse. A menu of commands appears.

2. Click on the `Add Object` command. A prompt appears in the minibuffer requesting you to enter the class of the new object.

3. Be sure that the class for the new object is :PRINTER, and press RETURN. You are now prompted to enter the name of the new **:printer** object.

4. Enter a name for your printer. The name can be any alphanumeric collection of characters. FLEET's character printer resides on host Astrolabe, so as a mnemonic we enter P-ASTRO. A display will appear, similar to the one following, showing the default attributes (and values) of the new printer.



```
                    NAMESPACE EDITOR FOR "SIMPLE-NETWORK"
                    This is a namespace of type :NETWORK

      •
      •
      •

 :PRINTER
   "P-ASTRO"
      :HOST                      "New Explorer"
      :TYPE                      :TI855
      :STREAM                    :SERIAL
      :PORT                      NIL
      :BAUD                      4800
      :DATA-BITS                 8
      :STOP-BITS                 1
      :PARITY                    NIL
      :XON-XOFF                  T
      :CHARACTER-PRINTER-P       T
      :IMAGE-PRINTER-P           NIL

      •
      •
      •
```

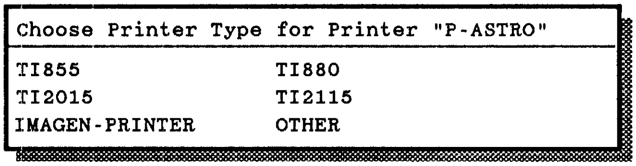
5. Place the mouse cursor box around the attribute :HOST and click middle on the mouse. A menu of commands appears.

6. Click on the Edit Attribute command. The edit window appears.

7. Delete "NEW EXPLORER", and replace it with the name of the host to which your printer is attached. Press the END key (*not* RETURN) to signal completion of the new value for the :HOST attribute.

8. If your printer is not a TI855 as shown as the default value for the :TYPE attribute, place the mouse cursor box around the attribute :TYPE and click middle on the mouse. A menu of commands appears.

9. Click on the Edit Attribute command. This time, a pop-up menu appears, similar to the one following:



```
Choose Printer Type for Printer "P-ASTRO"
TI855                 TI880
TI2015                TI2115
IMAGEN-PRINTER        OTHER
```

10. Click on the printer type that agrees with your printer.

11. Change the values as needed for any of the remaining attributes shown for your new printer. Whenever you click on the `Edit Attribute` command for the remaining attributes, an appropriate pop-up menu will appear, complete with the necessary choices from which to select.

12. Repeat steps 1 through 11 for any other printers that you need to add to your network.

Create the Site Object

4.5.2.8 Having added the necessary values for all the **:host, :namespace,** and **:printer** objects, you must now define your site attributes.

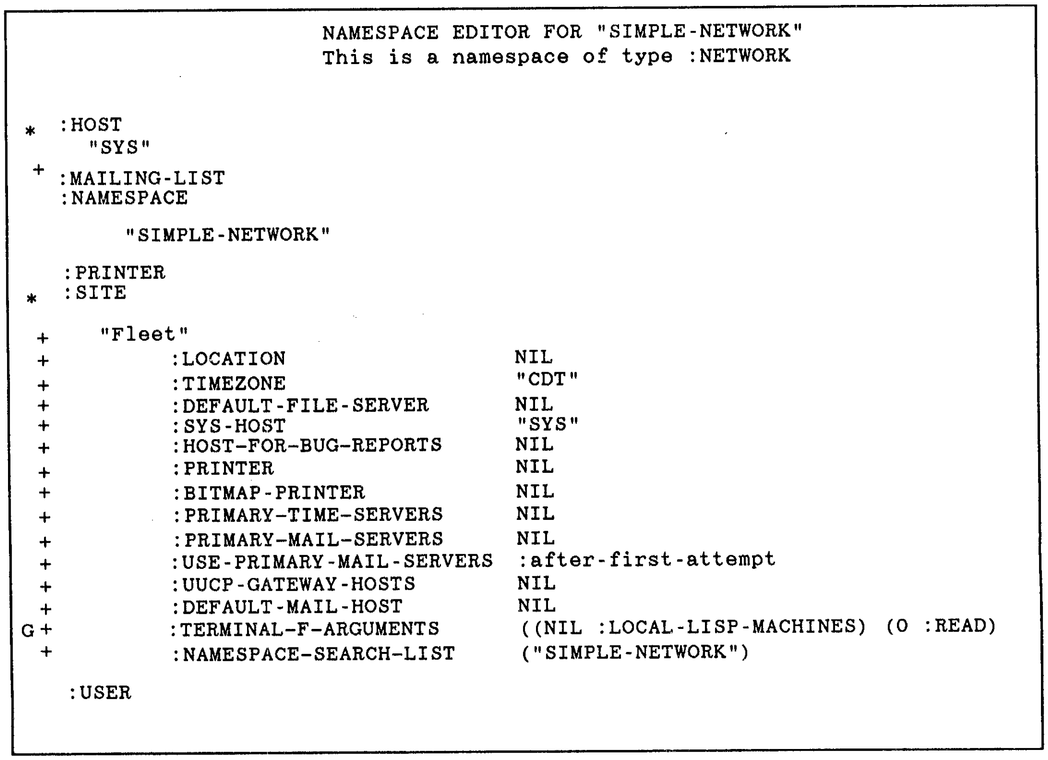
Remember, if you have questions about a class, object, or attribute, you can refer to paragraph 4.2, titled The Network Namespace, or you can move the editing cursor next to the item for which you need information, and then press CTRL-SHIFT-D to see the online documentation for that item.

1. Place the mouse cursor box around the class name `:SITE` and click middle on the mouse. A menu of commands appears.

2. Click on the `Add Object` command. A prompt appears in the minibuffer requesting that you enter the class of the new object.

3. Be sure that the class for the new object is `:SITE`, and press RETURN. You are now prompted to enter the name of the new **:site** object.

4. Enter a name for your site. The name can be any alphanumeric collection of characters. FLEET is the site name in our example. When we enter `FLEET` and press RETURN, a display will appear, similar to the one following, showing the default attributes (and values) of the new site.



```
                         NAMESPACE EDITOR FOR "SIMPLE-NETWORK"
                         This is a namespace of type :NETWORK


*  :HOST
     "SYS"
+  :MAILING-LIST
   :NAMESPACE

       "SIMPLE-NETWORK"

   :PRINTER
*  :SITE

+     "Fleet"
+          :LOCATION                   NIL
+          :TIMEZONE                   "CDT"
+          :DEFAULT-FILE-SERVER        NIL
+          :SYS-HOST                   "SYS"
+          :HOST-FOR-BUG-REPORTS       NIL
+          :PRINTER                    NIL
+          :BITMAP-PRINTER             NIL
+          :PRIMARY-TIME-SERVERS       NIL
+          :PRIMARY-MAIL-SERVERS       NIL
+          :USE-PRIMARY-MAIL-SERVERS   :after-first-attempt
+          :UUCP-GATEWAY-HOSTS         NIL
+          :DEFAULT-MAIL-HOST          NIL
G+         :TERMINAL-F-ARGUMENTS       ((NIL :LOCAL-LISP-MACHINES) (0 :READ)
+          :NAMESPACE-SEARCH-LIST      ("SIMPLE-NETWORK")

   :USER
```

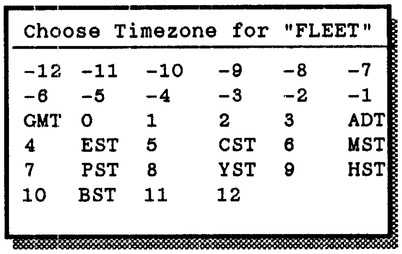
5. Place the mouse cursor box around the attribute `:LOCATION` and click middle on the mouse. A menu of commands appears.

6. Click on the `Edit Attribute` command. The edit window appears.

7. Delete `NIL`, and replace it with the name of your site. Press the END key (*not* RETURN) to signal completion of the new value. In the example, FLEET is located in Galveston, so we would enter `Galveston`.

8. If the default value shown for the `:TIMEZONE` attribute is correct, skip to step 11. Otherwise, place the mouse cursor box around the attribute `:TIMEZONE` and click middle on the mouse. A menu of commands appears.

9. Click on the `Edit Attribute` command. A pop-up menu appears, similar to the one following:



```
Choose Timezone for "FLEET"

-12 -11  -10  -9   -8   -7
-6  -5   -4   -3   -2   -1
GMT 0    1    2    3    ADT
4   EST  5    CST  6    MST
7   PST  8    YST  9    HST
10  BST  11   12
```

10. Either click on one of the three-letter timezones that matches the timezone where your site is located, or click on a number that represents the difference in hours between Greenwich Mean Time (GMT) and your local timezone. Negative numbers indicate that your timezone is earlier that GMT, and positive numbers indicate that your timezone is later than GMT.

11. Change the values as needed for any of the remaining attributes shown for your new site. Whenever you click on the `Edit Attribute` command for the remaining attributes, an appropriate edit window will appear, or a pop-up menu will appear that contains the choices from which you can select.

NOTE: If you have multiple values to add to a given attribute, for example **:terminal-f-arguments**, be sure that the attribute has been designated as a group attribute. However, when order is important, as in **:primary-time-servers**, the attribute should not be a group attribute. Group attributes have the letter G slightly to the left of the attribute.

To make an attribute a group attribute, place the mouse cursor box around that attribute and click middle. Then, click on the Toggle Group Status command.

Verify the Network Namespace

4.5.3 After you have created a network namespace, you must write out the changes. When the changes are written, they are also *verified*. Verification checks to make sure that the choices you made during the creation of the network namespace are valid and consistent.

To write out the new network namespace you have created, perform the following steps:

1. Click middle on the mouse. A menu of commands appears.

2. Click on the following command: `Verify namespace`. A prompt appears in the minibuffer asking if you want to verify the entire network namespace.

3. Because your network namespace is totally new, answer YES, and press RETURN. At this point, your new network namespace is verified, and any constraint violations are reported. If you have errors, correct them before proceeding.

After you are sure that the changes are stable, write them out globally. Before the Namespace Editor will write any changes globally, it performs an *incremental verification* to ensure that your changes are consistent. An incremental verification checks only those objects that have been edited to see if they are consistent with the rest of the namespace.

Distribute the Network Namespace

4.5.4 When you have made all your corrections, you are ready to distribute your new namespace. The act of distributing your network namespace performs several functions.

- Creates the network namespace cache on the local machine. The cache holds a partial copy of the network namespace for use during an interactive Explorer session. The cache goes away when the machine is rebooted, and after a certain time interval, the cached entries are marked for renewal, and are renewed the next time you query the namespace for that entry.

- Writes a machine-readable representation of the namespace into the following file: lm:name-service;<*namespace-name*>.xld#>

- Creates the following file: lm:name-service;<*namespace-name*>-log.lisp#> This file will be used to record any changes made to the network namespace either by your current machine or by any other host on the network.

To distribute your new network namespace, perform the following steps:

1. Click middle on the mouse. A menu of commands appears.

2. Click on the `Distribute namespace` command.

3. Press the END key to exit the Namespace Editor utility.

Also, you can press the END key after verifying the namespace changes, and you will be prompted to distribute the namespace.

Boot the Hosts on the New Network

4.5.5 As you boot each of the hosts on your new network, the network initialization menu will appear. Select the `Change the name of this machine and return to this menu` prompt, and change the name of each host according to your configuration plan made earlier.

As each new host issues its who-am-i message along with its new name, your network namespace co-server responds by sending necessary boot information to that host.

After obtaining the necessary boot information, each new host completes its initial boot sequence and is then available for use, with full network access.

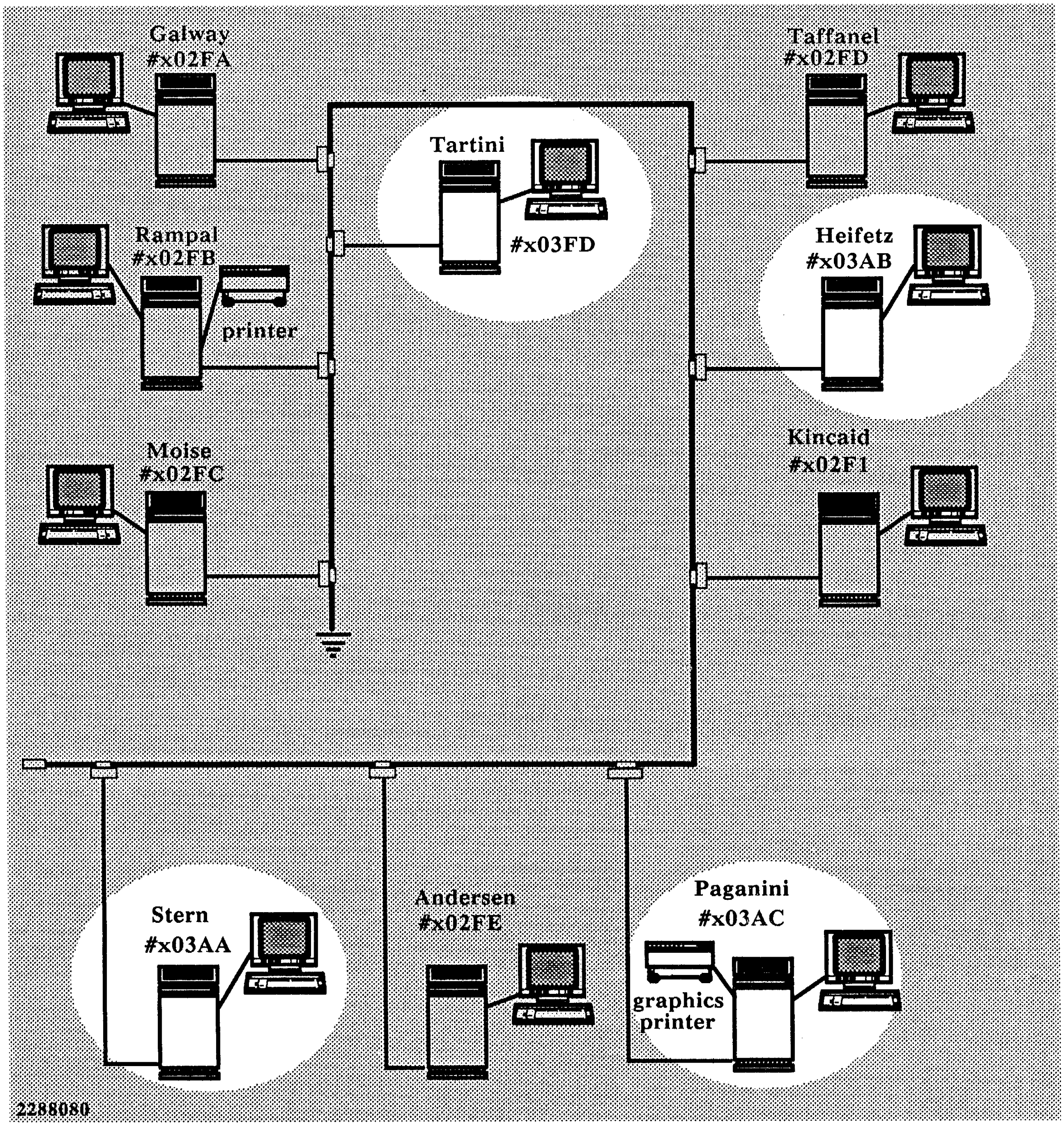
Logical Subnets

4.6 All discussion of network configuration to this point has dealt with *physical* bridges and subnets, in which there is an actual hardware separation between subnets. For example, CHARON serves as the bridge between the two physical subnets at FLEET. CHARON has two Ethernet controller boards and two Ethernet addresses, which translate into two Chaosnet addresses. It is CHARON's responsibility to forward messages from hosts on one subnet to hosts on the other subnet. It is possible, and in many cases desirable, to have multiple *logical* networks on a single physical Ethernet network. In such cases, all hosts are connected directly to a single Ethernet and networks are defined logically. This allows multiple logical networks and subnetworks to reside on a single hardware network.

Figure 4-4 shows an Ethernet on which two logical networks reside. The highlighted areas are on one network. All of the hosts on this network have a Chaosnet address #x03. All of the hosts with a Chaosnet address of #x02 are on the other subnetwork. This serves to isolate the hosts from one network from those in the other. The hosts on network #x03 cannot communicate at all with those on subnet #x02. This kind of insulation is useful where some machines need to be quarantined for one reason or another.

In cases such as these, two network namespaces must be created on separate servers on each network.

Figure 4-4 Logical Subnetworks

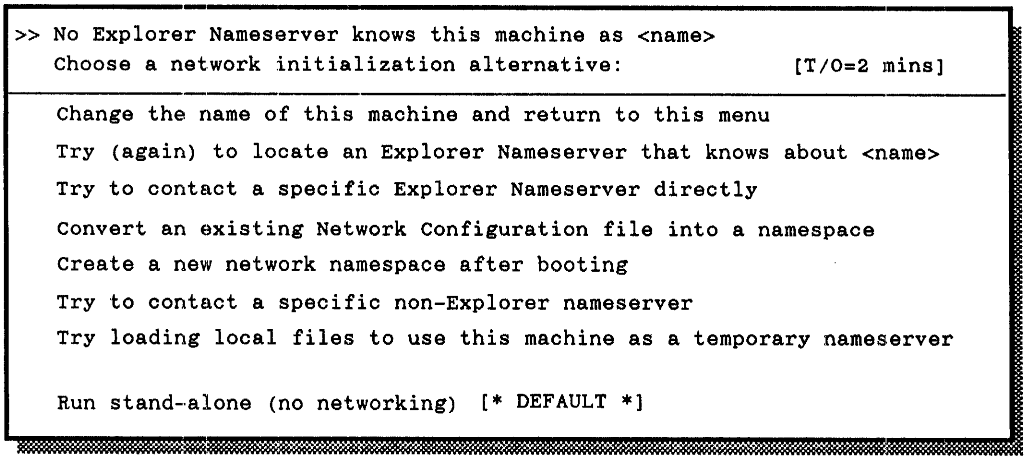


Network Initialization Menu Options

4.7 The network initialization menu appears not only when an Explorer is booted for the first time, but whenever no network namespace co-servers can be found to supply network information.

Similarly, the same menu appears if you execute the **name:initialize-name-service** function and no namespaces are listed in the local server-boot-list file (that is, your host is not a co-server), or no co-server responds to the who-am-i message.

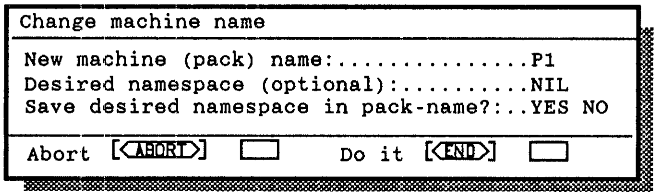
The following paragraphs describe each of the options supplied in the network initialization menu as shown here.



```
>> No Explorer Nameserver knows this machine as <name>
   Choose a network initialization alternative:          [T/O=2 mins]

    Change the name of this machine and return to this menu
    Try (again) to locate an Explorer Nameserver that knows about <name>
    Try to contact a specific Explorer Nameserver directly
    Convert an existing Network Configuration file into a namespace
    Create a new network namespace after booting
    Try to contact a specific non-Explorer nameserver
    Try loading local files to use this machine as a temporary nameserver

    Run stand-alone (no networking)  [* DEFAULT *]
```

Change Machine Name

4.7.1 The first option allows you to change the name of your Explorer. When an Explorer is first taken from the box it is named P1. If you are installing a new network of Explorers you will *have* to change the names of all the hosts but one, which can be left with the name P1. Otherwise you would create a network configuration in which all hosts on the network have the same name. You will probably want to change *all* the names, however. The following window appears if you click on this menu item.



```
Change machine name
New machine (pack) name:...............P1
Desired namespace (optional):..........NIL
Save desired namespace in pack-name?:..YES NO
Abort [<ABORT>] □      Do it [<END>] □
```

Note that the current machine name, P1, appears as the default new machine name. You can replace this name with any name you wish. Press RETURN to signify that you have completed adding a new name.

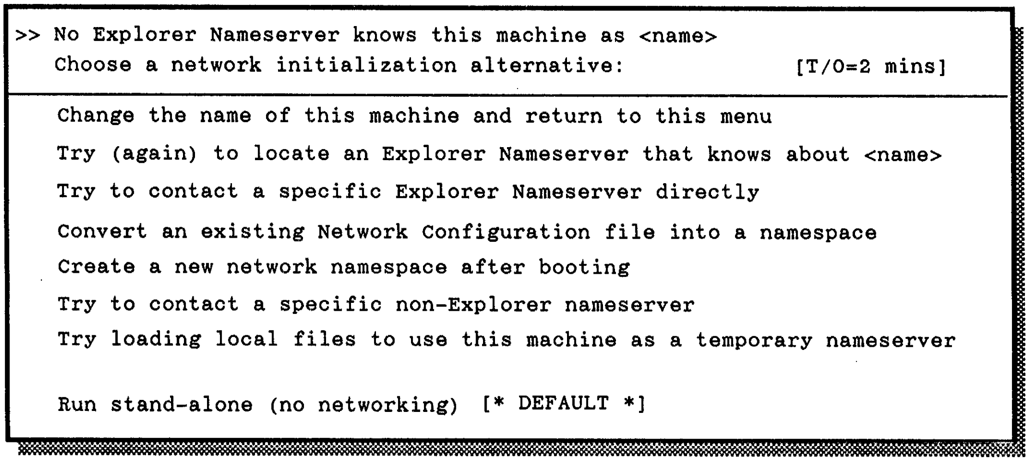
The `Desired namespace (optional)` prompt lets you specify a default network namespace to be used with this host. When you give a value for this prompt, that value is immediately bound to the **name:*default-who-am-i-domain*** variable. This variable stays bound to the new value until you reboot.

The Save desired namespace in pack-name prompt lets you store the name of the network nanespace in your disk label, so that when you boot your host, it will select that network namespace automatically.

Select the Do it option with the mouse or press END to change the machine name and return you to the network initialization menu. Choosing Abort selects the default machine name and returns you to the network initialization menu.

Locate an Explorer Nameserver

4.7.2 If you choose the second item on the network initialization menu, your Explorer will attempt to find an Explorer Nameserver on the network that knows about your machine. If there is no Explorer nameserver to be found, you will be returned to the network initialization menu, with an advisement to the effect that no nameserver could be found:

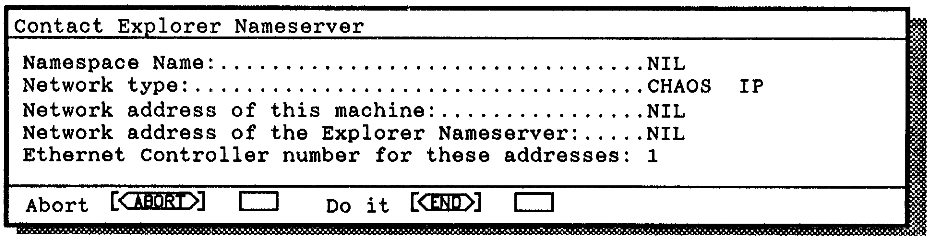


```
>> No Explorer Nameserver knows this machine as <name>
   Choose a network initialization alternative:            [T/O=2 mins]

   Change the name of this machine and return to this menu
   Try (again) to locate an Explorer Nameserver that knows about <name>
   Try to contact a specific Explorer Nameserver directly
   Convert an existing Network Configuration file into a namespace
   Create a new network namespace after booting
   Try to contact a specific non-Explorer nameserver
   Try loading local files to use this machine as a temporary nameserver

   Run stand-alone (no networking)  [* DEFAULT *]
```

Contact Specific Explorer

4.7.3 If your new Explorer has been preassigned an address in an existing network namespace configuration, and you know the address of an Explorer co-server on that network, you can select the third option in the network initialization menu to obtain namespace information from the remote co-server for your local machine.

If you choose this network initialization menu item, the following window appears:



```
Contact Explorer Nameserver
 Namespace Name:................................NIL
 Network type:..................................CHAOS  IP
 Network address of this machine:...............NIL
 Network address of the Explorer Nameserver:.....NIL
 Ethernet Controller number for these addresses: 1

 Abort [<ABORT>] □     Do it [<END>] □
```

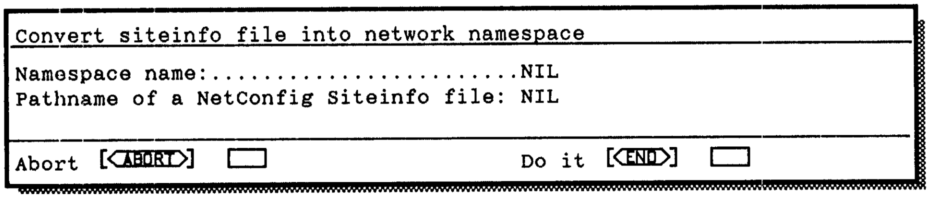
This pop-up window prompts you for a namespace name, network type, address of the local machine, the Explorer nameserver (the host that supplies the namespace), and the number of the Ethernet controller of the local machine. Network types can be Chaosnet or Internet Protocol, depending on what protocols are supported on the target network.

The Ethernet Controller number identifies which Ethernet controller board in your chassis provides control for the network address of your host. The default value of 1 identifies the first Ethernet controller board in your chassis (going from right to left, looking in the back of the Explorer chassis).

Convert a Network Configuration File

4.7.4 If you have updated a site's Explorers from Release 2 to Release 3 software, you must convert the main network configuration server's configuration file (which is normally named lm:site;siteinfo.xfasl) into a network namespace. This subject is discussed extensively earlier in this section in paragraph 4.3, titled How to Update a Network From Release 2.

If you choose the fourth option from the network initialization menu, the following window will appear, requesting that you enter a namespace name and the name of the existing configuration file (which you want to convert).



```
Convert siteinfo file into network namespace

Namespace name:.......................NIL
Pathname of a NetConfig Siteinfo file: NIL

Abort [<ABORT>] □                    Do it [<END>] □
```

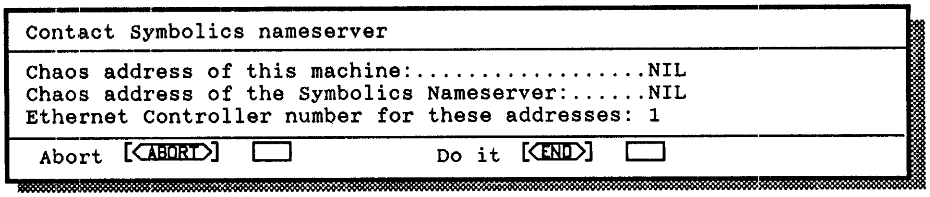
Choosing the Abort option simply returns you to the network initialization menu.

Create a New Network Configuration

4.7.5 If you choose the fifth option in the network initialization menu, the boot sequence will continue. When booting completes, you can enter the namespace editor as described in the example at the first of this section.

Contact a Specific Non-Explorer Nameserver

4.7.6 If you are installing a new Explorer on an existing network whose nameserver is not an Explorer, you can establish contact by choosing the sixth option in the network initialization menu. The following window will appear:



```
Contact Symbolics nameserver

Chaos address of this machine:..................NIL
Chaos address of the Symbolics Nameserver:......NIL
Ethernet Controller number for these addresses: 1

 Abort [<ABORT>] □               Do it [<END>] □
```

The Chaos-address prompts can be in any format.

The Ethernet Controller number identifies which Ethernet controller board in your chassis provides control for the network address of your host. The default value of 1 identifies the first Ethernet controller board in your chassis (going from right to left, looking in the back of the Explorer chassis).

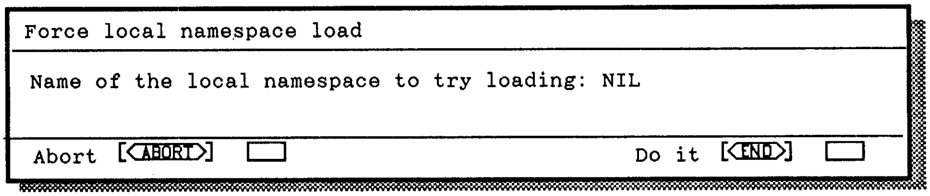
Currently, the only non-Explorer name server that is supported is Symbolics.

Load Local Files For a Temporary Nameserver

4.7.7 Even though your local host is not a network namespace co-server, you can store certain files on your host so that you can gain network access when no co-servers are available. You should copy the following files from a co-server so that you will be prepared for this contingency:

lm: name-service;server-boot-list.lisp#>
lm: name-service;<*namespace-name*>-public.xld#>
lm: name-service;<*namespace-name*>-log.lisp#>

You can store these files anywhere on your local host, and specify the namespace name in response to the pop-up menu's prompt whenever you select this option from the network initialization menu:



```
Force local namespace load

Name of the local namespace to try loading: NIL
```

Although loading the XLD file allows you to access the network and even to make local changes to the namespace, you cannot make any global changes to the network namespace.

Running Standalone

4.7.8 If you do not wish for this Explorer to participate in a network, you can choose the last option. If you do not choose one of the options in the network initialization menu within two minutes, the configuration will default to standalone operation, and the initial boot sequence will continue.

Defaults for Booting a Disk-Saved Version

4.7.9 Whenever you choose any of the non-standard boot selections from the network initialization menu (such as `Try to contact a specific Explorer nameserver directly`, `Try to contact a specific non-Explorer nameserver`, or `Try loading local files to use this machine as a temporary nameserver`, the following prompt appears as soon as that option completes its actions:

```
Should your choice be the default for booting a disk-saved version of
this band? (Yes or No)
```

If you answer YES to this query, the namespace software stores certain information inside the **name:*non-standard-boot-alternative*** variable. You can then perform a disk save of the current environment, and when you reboot, your host will automatically select the same non-standard boot alternative. In so doing, your host bypasses both the network initialization menu and any network namespace co-servers not identified as part of the booting alternative.

If, by mistake, you answer YES to this query, simply set the **name:*non-standard-boot-alternative*** variable to **nil** before performing your disk save. By doing so, you can follow the standard boot alternatives on rebooting.

Network Namespace Functions and Variables

4.8 The following functions and variables can be used to modify your network namespace environment.

name:initialize-name-service &optional *namespace* (*display* **t**) Function

Performs all name service (and network) initializations as those done during a cold boot (destroying any local namespace updates). If you specify *namespace*, this function broadcasts for a server of that *namespace* (but does not override a qualified pack host name for this machine). If *display* is non-**nil** (the default), the resulting namespace configuration is shown on ***standard-output***.

name:run-standalone Function

Removes your host from the network until you run the **name:initialize-name-service** function.

name:show-namespace-configuration &optional (*stream* ***standard-output***) Function

Displays information about the namespaces available on this machine. The following information is displayed:

- Name — The name of the namespace.
- Type — The type of namespace, such as **:personal**.
- Usage — The usage mode corresponding to a set of expert editors. (Refer to the *Explorer Tools and Utilities* manual for information on expert editors.)
- Search # — Where in the search list the namespace appears (1 for first, 2 for second, and so on).
- Server? — Whether the namespace has a server at this machine.

name:distribute-namespace *namespace-name* Function
&key **:local-only** (**:save-first t**) **:server-list**
(**:search-list-loc :beginning**) (**:notify t**)

Terminates configure or convert mode for a new public namespace. These modes are described in the **:explorer-server** *init-args* of the **name:add-namespace** function in the *Explorer Tools and Utilities* manual.

CAUTION: The name:distribute-namespace is mainly intended for internal use.

You can optionally write the namespace to a binary file. You should do this if you are in convert mode because no changes have been written to the log file.

Propagating to servers of subsequent changes is enabled.

A local client instance is created, and all further namespace accesses at this machine now go through the client instance rather than directly to the server instance.

Optionally, you can start other servers, which obtain their information from this host. This works only if those servers are already on the network.

:local-only — When non-**nil**, distributes only to this host. The default is **nil**.

:save-first — When non-**nil** (the default), writes the namespace to an xld file first.

:server-list — Specifies a list of servers (host names) to receive distribution. The default is all servers.

:search-list-loc — Specifies where to put the namespace in the search list: **:beginning**, **:end**, or **nil** (that is, do not put it on the search list).

:notify — When non-**nil** (the default), notifies you if some servers did not answer.

name:*default-who-am-i-domain* Variable

The **name:*default-who-am-i-domain*** variable allows you to specify which network namespace that your host will boot under. The value you bind to this variable should be the name of the network namespace. You must perform a disk save to make the variable-binding permanent across boot sessions.

name:*non-standard-boot-alternative* Variable

The **name:*non-standard-boot-alternative*** variable is only useful when you mistakenly answer YES to the `Should your choice be the default for booting a disk-saved version of this band? (Yes or No)` query during a non-standard boot sequence.

A YES answer to this query causes the namespace software to store certain information inside the **name:*non-standard-boot-alternative*** variable so that a disk save of the current environment will cause future reboots to automatically select the same non-standard boot alternative. In so doing, your host bypasses both the network initialization menu and any network namespace co-servers not identified as part of the booting alternative.

To correct the mistake, simply set the **name:*non-standard-boot-alternative*** variable to **nil** before performing your disk save. By doing so, you can follow the standard boot alternatives on rebooting.

nse:*verification-level* Variable

This variable specifies the default level of verification, which applies to incremental verification as well as namespace verification. The following lists the possible values for this variable:

- **:errors-only** — Warnings are not printed for incremental verification or namespace verification.
- **nil** or **:none** — Automatic incremental verification is not performed before updates.
- **:full** — Full verification is performed; warnings and error messages are printed for all verifications. This is the default value.

net:get-host-attribute *host attribute* &optional *default* Function

The **net:get-host-attribute** function returns the value of the attribute identified by the *attribute* argument from the host identified by the *host* argument.

The *host* argument is either a symbol or a namestring.

The *attribute* argument is a keyword.

The optional *default* argument, if non-**nil**, causes this function to return the default value of the attribute rather than its current value.

For example, in a particular namespace, the following code returns a value of kj:

```
(net:get-host-attribute 'ti-7|knox-johnson :short-name)
```

net:set-host-attribute *host attribute value* Function

The **net:set-host-attribute** function stores the value specified by the *value* argument into the attribute identified by the *attribute* argument. The attribute is then updated locally in the namespace.

The *host* argument identifies a particular host; or rather a network namespace host-class object. The attribute to be set belongs to this host-class object. The *host* argument can be either a symbol or a namestring.

The *attribute* argument is a keyword.

The *value* argument can be any Lisp form (symbol, string, or Lisp object).

net:get-user-attribute *key* Function

The **net:get-user-attribute** function returns the value of the attribute identified by the *key* argument (based on who is currently logged in to the local host). If no **:user** object exists for that person, **net:get-user-attribute** returns **nil**.

net:get-site-option *key* &optional *local-only* Function

The **net:get-site-option** function returns the value of the site option identified by the *key* argument (for the local site only). Site option values are specified in the site class of the network namespace.

The *key* argument is a keyword, such as **:location**, **:timezone**, and so on.

The optional *local-only* argument, if non-**nil** (the default is **nil**), always returns a value obtained from the local cache; never from a network namespace co-server.

net:set-logical-host *logical-host translated-host* Function
&key **:site-directory :site-device (:verbose t)**
(:local-only *local-only-namespace-updates*)
(:namespace *default-namespace-for-logical-hosts*)

The **net:set-logical-host** function sets the value of the *logical-host* argument so that it translates to the host (logical or physical) identified by the *translated-host* argument. For information about logical hosts, see the *Explorer Input/Output Reference* manual.

If *translated-host* is a logical host, then this function only sets the host translation; the directory translations are provided from the host identified by the *translated-host* argument. If *translated-host* is a physical host, then this function not only sets the host translation; it also tries to provide directory translations by loading the site translations file. The name of this file is based on the arguments to **net:set-logical-host:**

"*translated-host*: *site-directory*; *logical-host*.translations"

If the host has device components to its pathnames, the *site-device* argument would provide that component immediately following the *translated-host* component.

The **:site-directory** argument identifies the directory name where the translation file resides. This value can be either a string (such as "site", the default) or a symbol.

The **:site-device** argument identifies the device name where the translation file resides. On Explorer systems, this defaults to **nil**.

The **:verbose** argument (which defaults to **t**) specifies whether or not to print a message to the screen informing the user about the change.

The **:local-only** argument, if non-**nil**, causes the translations to be in effect only for your local namespace cache. The default value of this keyword is the same as the value of the ***local-only-namespace-updates*** variable.

The **:namespace** argument, if non-**nil**, identifies which network namespace contains the logical host specified as the first argument of this function. The default value of **:namespace** is the same as the value of the ***default-namespace-for-logical-hosts*** variable.

net:set-sys-host *translated-host* Function
&key **:site-directory :site-device** (verbose **t**)
(*local-only* ***local-only-namespace-updates***)
(*namespace* ***default-namespace-for-logical-hosts***)

This function sets the host called "SYS" to translate to the value specified by the *translated-host* argument.

net:translated-host *host* Function

This function returns three values: the translated host object, the logical-host object that translated to a non-logical host, and the directory translations.

CHAOSNET APPLICATIONS PROGRAMMING AND NETWORKING

5

Introduction

5.1 This section, which discusses Chaosnet applications programming and networking, assumes that your are familiar with the Chaosnet protocol. The main source of information about the Chaosnet protocol is available as a memo (AIM-628) from the following address:

Publications, Room NE43-818
M.I.T. Artificial Intelligence Laboratory
545 Technology Square
Cambridge, MA 02139 USA

Note that the memo describes a Chaosnet network built on Chaosnet hardware. The Explorer Chaosnet implementation is built on top of Ethernet hardware.

Section 5 focuses on creating Chaosnet-specific network servers, including how they are implemented on server hosts, and how you can access the service provided. Several example servers are discussed in detail.

A server is a Lisp program on a host that provides some service to other hosts on the network. The user can usually access such a program in a manner that is transparent to the operator at the server host. Every Explorer server requires two types of Lisp functions:

- User functions — The functions that the user invokes to access the server

- Server functions — The functions that process the user's request for service and return the results to the user

The functions invoked by the user are usually fairly simple in construction. The server's side can be more complicated. There are two types of functions that must be implemented at the server's side of a connection:

- Auxiliary functions — The functions the server uses to perform the data processing procedures requested by the user

- Communications functions — The functions the server uses to communicate with the user

```
((:FILE :LOCAL :LOCAL-FILE)
 (:FILE :TCP :FTP))
```

Connections

5.2 Processes on different hosts can communicate by using a simple transaction as described in the following paragraph; however, for more sustained communication to occur between the processes, Chaosnet must first establish a *connection* between the processes. A Chaosnet connection is a full-duplex channel.

A *conn* is one side of a Chaosnet connection. A conn is a named structure of type **chaos:conn** (for more information on structures, see the section titled Structures, in the *Explorer Lisp Reference* manual). The conn may have an actual connection attached to it; it may have a connection still being made; or it may record that a connection was refused, closed, or broken.

Using Simple Transactions

5.3 A *simple transaction* is one in which a user host sends out one request for connection packet to a server host, which in turn returns exactly one answer packet back to the user. A full connection is not established during a simple transaction. Therefore, if a local host performs a simple transaction and expects a reply from a remote host, it is possible that the remote host may receive the request and actually reply. If the reply is lost on the network, the local host locks up and waits for the reply even though it may never be forthcoming. For this reason, simple transactions are not normally used in situations where a client and server must exchange data.

Since a packet's data field is limited to 488 bytes, the amount of data that can be exchanged between the server and the user during a simple transaction is restricted, but simple transactions are quite useful when small amounts of data need to be passed back and forth over the net.

User Side

5.3.1 The following example illustrates the use of a very simple service and the way it is accessed from the user's side. The means by which the user accesses a particular server can vary from simple to extremely complex. Usually a server is invoked by a function provided to users for that purpose. The following function takes a minimalist approach to server access. To invoke the services of the *Witticism* server on the host called GROUCHO, the user can simply enter the following into a Lisp Listener:

```
(witticism "GROUCHO")
```

The definition of this function is as follows:

```
(defun witticism (host)
   "Return witticism."
   (let ((pkt (chaos:simple host "Witticism")))
      (format t "~&~s" (chaos:pkt-string pkt))
      (chaos:return-pkt pkt)
      nil))
```

The `chaos:simple` function carries out the user side of a simple transaction. In so doing, `chaos:simple` performs several operations. First, a connection is created and opened, and a request-for-connection packet is transmitted over the network to the host specified as the argument to `witticism`. The contact name `"witticism"` is transmitted in the data field of the request-for-connection packet. The system waits for an answer packet in response. Upon receipt of the server's packet, the connection is automatically closed.

The variable pkt is bound to the value returned by the **chaos:simple** function, which is the packet transmitted *back* to the user from the server. The function `chaos:pkt-string` extracts the string filling the data field of the Chaosnet packet. The `format` statement sends a carriage return and formats this returned string.

The next function called is `chaos:return-pkt`, which returns the packet to the system for reuse. This function must always be used after using the **chaos:simple** function.

Server Side

5.3.2 A server is a *process* on a host that provides some service to other hosts on the network. Servers listen to the network for requests and respond to a user. The definition of the `witticism-server` function is as follows:

```
(defun witticism-server ()
   (let* ((conn (chaos:listen "witticism")))
         (chaos:answer-string conn (wit))))
```

Listening

5.3.2.1 The first part of the function `witticism-server` establishes a connection `conn` that is in a **chaos:listening-state** waiting for an incoming request for connection. The function `chaos:listen` returns the connection with the contact name `"witticism"`.

Contact Name

5.3.2.2 The contact name is a string identifying the particular server. As mentioned earlier, the contact name is transmitted in the data field of the request for connection packet. When GROUCHO receives the request for connection with the contact name `"witticism"`, `chaos:listen` returns a connection that is in the **chaos:rfc-received-state**.

NOTE: The *contact name* is always the string of bytes up to the first #/Space character in the data field of the packet. This delimiter allows the contact name to be distinguished from any data being transmitted in the RFC packet.

Sending

5.3.2.3 The server now answers the user's request for a witticism using the function `chaos:answer-string`. This function sends a reply back to the user with a string in its data field. This is the heart of the simple server illustrated here.

Server Auxiliary Function

5.3.2.4 *Server auxiliary* functions are used to carry out the network-independent data processing operations associated with the server. In this example, the function `wit` is called. Its duty is to return a randomly chosen witticism from a data bank. The following is a very simple example of an implementation of this function:

```
(defun wit ()
   (nth (random (length *witticisms*)) *witticisms*))
```

The function randomly selects a witticism-string from the list identified by the `*witticisms*` variable. An example of how to establish the `*witticisms*` list follows:

```
(defvar *witticisms* '("Time is money."
                       "Money is the root of all evil."
                       "Time is the root of all evil."
                       "Without evil there can be no good."
                       "Without money you cannot have a good time."
                       "Without time you cannot spend money."))
```

Initializations

5.3.2.5 For the witticism server to function properly, the server function that is to listen for user requests must first be added to the **chaos:server-alist**. This association list (alist) maintains a list of all services that a given host provides over Chaosnet. This list tells the system to evaluate the function `witticism-server` automatically, whenever a request for connection is received. Automatic evaluation ensures that the host is always listening for a request for connection with the server. Use the **add-initialization** function to add a server to the server alist.

```
(add-initialization "witticism"
          '(process-run-function "wit" 'witticism-server)
          nil
          'chaos:server-alist)
```

For more details on initializations, see the *Explorer Lisp Reference* manual. Note that `process-run-function` is used when adding the initialization, rather than calling the server function directly. This action ensures that the server function has its own separate process in which to run (as opposed to running in the background). By running separate processes, Chaosnet is not affected if the server process fails.

Sending Messages From the User Side

5.4 Many applications that employ a simple transaction require that a message be sent from the user side to the server side. When the user wishes to send a message to another host, the message must be included in the data field of the packet. At the user side, this operation requires the concatenation of the contact name, which occupies the first *n*-bytes up to the first #\Space in the data field of the packet, and the actual message, which occupies the rest of the data. This character string then has to be parsed at the server side of the connection so that the actual message can be printed on the server's screen--minus the contact name. The following example shows the implementation of `chat`, a simple message-sending program. A user at the client host can send a message to a chat server at host YOUNG by entering the following in a Lisp Listener:

```
(chat "young" "This is intended as a message for Young")
```

The server's function is to print the message on the screen of the local host. It then sends a reply back to the originator of the transaction with the message `"Gossip sent!"`.

The User's Packet

5.4.1 At the user's side, the message is transmitted by the use of the function chat, whose definition is given here:

```
(defun chat (host message)
   (let ((pkt (chaos:simple host
                               (format nil "chat ~A " message))))
      (format t "~&~A" (chaos:pkt-string pkt))
      (chaos:return-pkt pkt)
      nil))
```

In this example, the variable pkt is bound to a packet returned from a chat server host.

The format nil sequence joins the contact name to the message (which must be entered by the user as a string). Note that the space after the last character in the contact name ("chat ") is crucial. This delimits the contact name from the rest of the data field at the server side of the connection.

Next, the string returned from the chat server is formatted on the client's screen. As before, this string is extracted from the packet with the chaos:pkt-string function. The packet pkt is then returned to the system for reuse with the function chaos:return-pkt.

The Server's Side

5.4.2 When the server receives the packet, the contact name, in this case chat is automatically parsed out of the data string. The rest of the data field is not, however. Your server function must have provisions for stripping the contact name from the data string prior to processing the remaining data in the string occupying the data field of the packet. The chat-server function is defined as follows.

```
(defun chat-server ()
   (let* ((conn (chaos:listen "CHAT"))
          (pkt (chaos:get-next-pkt conn))
          (message-string (subseq (chaos:pkt-string pkt) 5))
          (host (net:get-host-from-address
                   (chaos:foreign-address conn) :chaos)))
      (tv:notify nil "Chat from ~A: ~A" host message-string)
      (chaos:answer-string conn "Gossip sent.")
      (chaos:return-pkt pkt)))
```

In this function definition, a connection conn in the **chaos:listening-state** is created. As soon as conn goes into **chaos:rfc-received-state** (that is, it has recognized the contact name "CHAT"), pkt is returned with the chaos:get-next-pkt function. This function returns the next packet from a connection.

Next, the local variable message-string is bound to the message sent by the user at the client host. While the function **chaos:listen** can recognize a contact name that has been concatenated to a data string in the data field of the packet, it does not actually remove it from the data. To strip the message from the contact name, you can use the function subseq on the string returned by chaos:pkt-string. The numeric argument to subseq must be an integer equal to the length of the actual contact name plus 1 (to account for the #/Space after the contact name in the string). Since (length "CHAT ") returns 5, the chat-server function extracts everything in the data string after the contact name by specifying a numeric argument 5 to subseq.

The next local variable (host) returns the client's host name, using the function chaos:foreign-address. This function returns the Chaosnet address of the remote host at the other end of the connection. The function **net:get-host-from-address** returns the host address of the remote host using its Chaosnet address.

The message is then displayed on the user's screen.

The function chaos:answer-string sends a verification message back to the source host that the operation has been successful. This type of return message is the only mechanism available to a simple transaction by which the originator of the transaction can determine the success of the transmission.

Lastly, the chaos:return-pkt function is called to return the packet to the system for reuse.

As with all servers, the chat server must be added to chaos:server-alist to ensure that it listens constantly for the chat server's contact name.

```
(add-initialization "chat"
                    '(process-run-function "chat" 'chat-server)
                    nil
                    'chaos:server-alist)
```

Using Stream Input and Output

5.5 Chaos streams provide a higher level of abstraction to insulate you from many bookkeeping details, such as keeping track of the state of a connection or determining the number of bytes in the data field of the packet, that are required when using the lower-level Chaosnet functions. The functions **chaos:open-stream, chaos:make-stream,** as well as the methods **:foreign-host, :close, :force-output, :finish, ;eof,** and **:clear-eof,** are used to manipulate Chaosnet streams, a higher-level way to utilize full Chaosnet connections. The following discussion of the use of the Chaosnet stream functions and methods focuses on the development of a simple server that checks spelling for other hosts on the network.

Client Side—Spelling Server

5.5.1 A user at the client side of the speller invokes the spelling server with the function check-spelling. Input to this function is simply a list of words that the user wants to check for correct spelling. In return, the spelling-server returns a list of those words that are poorly spelled or possibly unknown to the server. The client-side function check-spelling is as follows:

```
(defun check-spelling (host text)
   (with-open-stream (stream (chaos:open-stream host "speller"))
      (write text :stream stream)
      (send stream :force-output)
      (format t
         "~&The following words are either incorrect or unknown:")
      (read stream)))
```

Opening a Stream

5.5.1.1 The function chaos:open-stream is used at the client side of a networking function to open a chaos stream. This function opens a Chaosnet connection and returns an I/O stream. The arguments to chaos:open-stream are the host to which you wish to connect and the server you wish to use. The host specified by the host argument provides the spelling server functions to the network. The contact name for the server is "SPELLER".

The Common Lisp macro with-open-stream is used to create a stream named stream. The constructor for the stream is the call to chaos:open-stream.

Writing to a Stream

5.5.1.2 The body of the `with-open-stream` macro is concerned with writing to and reading from the stream `stream`. The Common Lisp function `write`, which writes characters to a stream, is used to write the list of words to be checked to the stream `stream`.

Forcing Packet Output

5.5.1.3 The Chaosnet stream functions handle all of the details of placing user data into packets. Therefore, the user normally need not be concerned with the size of the body of data to be transmitted. If the body of data is very large, as in the case of a file transfer from client to server or vice versa, the Chaosnet stream functions break it into packet-sized pieces for transmission as a sequence of packets. If user data comes in chunks, however, they are normally held back (buffered) until there is enough data to fill a packet, which is subsequently transmitted.

There remains the possibility, however, that a small amount of data, too small to fill a packet, will fail to be transmitted. For instance, a small amount of user data at the client side of a file transfer (say, the last three characters and period, in the case of a text) may remain after the rest of the file has already been transmitted to the server. If no more data is forthcoming from the user, this data will never be transmitted. If there is a possibility that data to be transmitted will not fill the data field of the packet, then you must use the Chaosnet stream method **:force-output** to fill out the the packet's data field. Otherwise, the packet may never be transmitted. In general, any time your application deals with small blocks of data, you should send a **:force-output** message to the stream immediately after writing to the stream.

Reading From a Stream

5.5.1.4 The Chaosnet function **read** is used to read a stream of characters. Upon transmission of the list of words to be checked, the client-side function `check-spelling` waits for a reply from the server. It reads the reply on the bidirectional stream `stream`. The reply will be a list of words not found in the server's glossary.

Server Side—Spelling Server

5.5.2 The `spelling-server` function does the job of checking the spelling of the text transmitted by the user at the client-side of the session.

```
(defun spelling-server ()
   (let ((conn (chaos:listen "SPELLER")))
      (chaos:accept conn)
      (with-open-stream
         (stream
            (chaos:make-stream conn :ascii-translation t))
         (write (get-misspellings (read stream)) :stream stream)
         (send stream :force-output))))
```

The following form ensures that the server is listening for a contact name by adding an entry to **chaos:server-alist**:

```
(add-initialization "SPELLER"
                    '(process-run-function "spserver" 'spelling-server)
                    nil
                    'chaos:server-alist)
```

Server's End of Stream

5.5.2.1 In this example, once a connection has been accepted, the server makes a stream for that connection. It uses the Chaosnet `chaos:make-stream` function for this purpose. Since the data being passed to the server is in the form of a list of words, the server needs to know to translate the ASCII characters representing the data into characters that the Explorer system understands. To do this, you must pass the `:ascii-translation` argument to the function `chaos:make-stream` function with a value of `t`.

Checking Spelling

5.5.2.2 The actual process of checking the spelling is carried out by the server as follows. The Common Lisp `read` function reads from the stream `stream`. The value returned by `read` is passed as the argument to the auxiliary function `get-misspellings`, which returns a list of all of the unrecognized words in the list.

Writing Back

5.5.2.3 Now the server uses the `write` function to send the list of misspelled words back to the user. As happened at the client side, the server now sends a `:force-output` message to the stream to force the data to be sent.

Auxiliary Spelling Functions

5.5.3 The following definitions, or similar ones, are necessary for the spelling server to perform the tasks required of it. The following are the minimum necessary for the example server used here:

```
(defvar *spelling-list*
        '("This" "is" "a" "sample" "word" "list"))

(defun get-misspellings (text)
  (loop for word in text
        when (not
               (member (symbol-name word)
                       *spelling-list*
                       :test
                       #'string-equal))
        collect word))
```

Protocols

5.6 The simple servers discussed to this point involve simple interactions between the client side and the server side, in which the client requests information or a service from the server and the server performs that service or returns the needed information, terminating the information exchange immediately. When more complex interactions between the client side and the server side are required, it becomes necessary to synchronize interactions between the client and server. This is done by establishing a *protocol,* which determines the manner and order in which each of the communicating parties in the session is to interpret the data received from the other party.

To illustrate the problems posed in synchronizing complex interactions between the client and the server, a more complicated version of the spelling server is used. A hypothetical user interaction with this version of the spelling checker might proceed as follows.

You enter a request for a spelling check in a Lisp Listener:

```
(check-spelling
  '(Impudent and prolix sesquipedalians intimidate more than impress))
```

The server then checks each form against the glossary and returns a list of unrecognized forms to you at the client side of the session with the following message:

```
These words are incorrect or unknown:
(PROLIX SESQUIPEDALIAN)
```

You are then given an opportunity to add an unrecognized though correctly spelled word to the word list:

```
Do you wish to add PROLIX to the word list? (Y or N)
```

After you have responded to each prompt, the list of words to be added to the server's spelling list is sent to the server. The server then sends a message back to the client that terminates the session. The client side function then prints a `"Done"` message and returns **nil**.

Client Side

5.6.1 As always, the client side of the session invokes the server. The definition of the client server invocation function is as follows:

```
(defun check-spelling (host text)
  (with-open-stream (stream (chaos:open-stream host "SPELLER"))
    (write text :stream stream)
    (send stream :force-output)
    (let ((wrong-words (read stream)))
      (cond ((equal wrong-words "1")
             (format t "~&No incorrect spellings exist. ~%")
             (no-new-additions-to-spelling-list stream))
            (t (format
                 t
                 "~&These words are incorrect or unknown:~&~s~%~%"
                 wrong-words)
               (loop
                 for word in wrong-words
                 when (and word (y-or-n-p "Add ~S to the word list?"
                                          word))
                 collect word into new words
                 finally
                 (cond ((null new-words)
                        (no-new-additions-to-spelling-list stream))
                       (t (write new-words :stream stream)
                          (send stream :force-output))))))
    (let ((done (read stream)))
      (format t "~&~S" done))) nil)
```

The client side auxiliary communications function **no-new-additions-to-spelling-list** is defined as follows:

```
(defun no-new-additions-to-word-list (stream)
    (write "0" :stream stream)
    (send stream :force-output))
```

Server Side

5.6.2 The new `spelling-server` function is defined as follows:

```
(defun spelling-server ()
  (let ((con (chaos:listen "SPELLER")))
    (chaos:accept conn)
    (with-open-stream (stream
                         (chaos:make-stream conn :ascii-translation t))
      (let ((misspellings (get-misspellings (read stream))))
        (cond ((null misspellings)
               (write "1" :stream stream)
               (send stream :force-output))
              (t (write misspellings :stream stream)
                  (send stream :force-output)))
        (let ((temp (read stream)))
          (cond ((equal temp "0")
                 (send-done-message stream))
                (t (add-to-spelling-list temp))))
        (send-done-message stream)))))
```

The auxiliary communications function send-done-message, used by the spelling server function, is as follows:

```
(defun send-done-message (stream)
  (write "Done" :stream stream)
  (send stream :force-output))
```

For the sake of discussion the auxiliary functions used by the spelling server are kept as simple as possible.

```
(defun add-to-spelling-list (text)
  (loop for word in text
        collect (symbol-name word) into string-list
        finally (setq *spelling-list*
                      (nconc string-list *spelling-list*))))
```

The function get-misspellings has the same definition as it did earlier, as does the global variable *spelling-list*, which holds the spelling list.

```
(defun get-misspellings (text)
  (loop for word in text
        when (not (member (symbol-name word)
                          *spelling-list* :test #'string-equal))
        collect word))
```

Finally, the spelling server must to added to **chaos:server-alist** with the **add-initialization** function:

```
(add-initialization "SPELLER"
                    '(process-run-function
                       "SPELLER"
                       'spelling-server)
                    nil
                    'chaos:server-alist)
```

Writes and Reads

5.6.3 The more complicated the interaction between client and server, the more careful you must be that server and client side reads correspond to the correct server-side and client-side writes. Special care must be paid where a read at one side of a connection corresponds to two or more writes at the other side, as can occur, for example, in a conditional expression. In order to understand this concept more clearly, you can step through the client-side function, check-spelling and the server-side function spelling-server.

First Write—Client Side

5.6.3.1 The first write takes place at the client side of the connection. The function check-spelling sends the text or word list to the server to be checked. To make sure that the entire text is transmitted, a **:force-output** message is sent to the stream stream.

First Read—Server Side

5.6.3.2 At the server side of the connection, after opening the stream stream, the stream is read. The value returned by read is then passed to the function get-misspellings. The value of this list is bound to misspellings.

Second Write—Server Side

5.6.3.3 Now it is the server's turn to transmit. The first conditional clause offers two possibilities. Either it writes the arbitrarily chosen character "1" to stream, if get-misspellings has found no strange words and misspellings is bound to **nil**, or it sends the list of nonexistent or misspelled words back to the client-side function.

Second Read—Client Side

5.6.3.4 The client-side function binds the variable wrong-words either to "1" or to a list containing unrecognized words, whichever the server sends.

Flag Waving

5.6.3.5 If the spelling server does not find any incorrect words, it must notify the client-side function of this fact. Since **nil** cannot be sent back over the network, an arbitrary string is chosen to serve as a flag to the client side that no misspellings were found. In the first conditional clause in the function `spelling-server`, the string `"1"` is sent to flag the client function that no list of `wrong-words` is forthcoming.

NOTE: At this point, you might think that you could simply write the `"There are no incorrect spellings."` message and return **nil**, immediately terminating the transaction, as in the following code:

```
(cond ((equal wrong-words "1")
       (format stream "~&There are no incorrect spellings.")
       nil)))
```

This form does not work, however, because the spelling server expects to read additional input from the client side (on the server side, see the `(let ((temp (read stream)))` form). The whole communication session simply hangs after printing the message. The server will listen forever to its stream, and the function **with-open-stream** will never close the connection.

CAUTION: Every read, whether on the client side or on the server side, must have a corresponding write at the other side of the connection.

Third Write— Client Side

5.6.3.6 As noted in the previous discussion, after binding `wrong-words`, the function at the client side can respond to the server in one of two ways. It *must* respond, however.

Even if there are no new words or misspellings to be considered as additions to the `*spelling-list*`, the client side *must* return something to the server. If the client does not require further service of the server, it waves a flag during this write by sending a `"0"` back to the server with the `no-new-additions-to-spelling-list` function.

If the server finds unrecognized words in the input list, then at the client side you decide which of the words to add to the spelling list that the server maintains.

Again, there are two possibilities. If you decide not to add any new words to the lexicon, then the list bound to the variable `new-words` by the `loop` macro will be empty. As in the case of the previous option, the server is still listening to the stream, so the client *must* send something. The client side calls the function `no-new-additions-to-spelling-list`, as it did earlier, sending a `"0"` flag to the server.

The last possibility at the client side (the third write) is that you have made a selection of words to be added to the word list. These words are put into a list by the `loop` macro, which then uses the `write` function to send them to the server. To be sure that everything is sent properly, a `:force-output` message is sent to the stream.

Third Read—Server Side

5.6.3.7 The third **read** takes place at the server side of the connection. Whatever is read from the stream is bound to the variable `temp`. This variable is bound to either a list of words to be added to the `*spelling-list*` or to `"0"`.

Fourth Write

5.6.3.8 The value of the variable `temp` determines whether to add words to the `*spelling-list*` or not. If `temp` is bound to `"0"`, then `"done"` is transmitted back to the client by the fourth `write`. If `temp` is bound to a list of words, then the list is appended to the `*spelling-list*`, and a `"done"` message is sent back to the client.

Fourth Read

5.6.3.9 A communications session is always complete when the last read has been accomplished, whether by the client or by the server. In the spelling program, this final read occurs on the client side of the connection. When the client receives the `"Done"` message, `Done` is printed on the client's screen and **nil** is returned.

At this point, the functions at both sides of the connection have terminated normally. Neither side has any outstanding reads, so the Common Lisp function `with-open-stream` takes care of closing the stream at both the client side and the server side.

Frills

5.7 For clarity, the examples shown so far have defined simple servers that involve nether a great deal of user interaction nor error checking. In the following examples, a number of error-catching devices are used, as well as rejection of the connection and advice or notification to both the user at the client side and the user on the server side of the connection.

Rejecting a Connection

5.7.1 There are many conditions under which you might want to reject a request for connection. Most often the rejection occurs at the server side of the connection. The example shown below illustrates how this situation is handled.

You use the Chaosnet function **chaos:reject** to reject a request for connection. You can also include a message when you reject a connection.

```
(defun etcetera-server-function ()
  (let ((conn (chaos:listen "etc"))
        (cond ((not (member user-id '(nil " ") :test #'equal))
               (reject conn
                       (format ()
                               "This machine in use by ~A. Try later!"
                               user-id))
               (return-from etcetera-server-function ())))
               (t (chaos:accept conn)))
                        .
                        .
                        .
```

In this example, code has been added to reject a connection if anyone is logged onto the remote host. The code following the `(chaos:accept conn)` form is the same as found in the example in paragraph 5.6.2, with minor adjustments to parentheses.

Notifying the User at the Server Side

5.7.2 The `chat` server example in paragraph 5.3, Using Simple Trans actions, illustrated one way a message can be transmitted to a user at the server side of a connection. It is good practice, or at least polite, to incorporate some mechanism in your server functions to notify a user at the server host that a remote host is accessing a server, and to identify either the machine, by name or address, using the server, or the name of the user. There are several other possibilities to consider. Functions are available to allow you to get this information. When providing such notifications at the client side, your server process must be temporarily interrupted. The next examples show how this is typically done.

Getting Information on a Foreign Host

5.7.2.1 You get information on a foreign host by using the functions **chaos:conn-foreign-address**, **net:get-host-from-address**, and **host-short-name**. For instance, the following fragment of Lisp code will returns the host short name:

```
(host-short-name (chaos:conn-foreign-address conn))
```

The short name, discussed in Section 4, Getting on the Network, is the host name normally used in messages in the status line and in filenames. The function **host-short-name** takes an address as its argument. In this example, the address of the foreign host is retrieved with the function `chaos:conn-foreign-address`. This function takes a connection as argument and returns the address of the client side host.

The function **net:get-host-from-address** takes an address as argument and returns the actual host object. Information about this host can be obtained by sending specific messages.

Server Side Protection

5.7.3 Under certain circumstances, a user at the server side of a connection might find it inconvenient to allow another machine to access a server. You can provide a means to allow a user at the host to selectively disallow the establishment of a connection. First, you need to create the ***spelling-server-on*** variable as follows:

```
(defvar *spelling-server-on* nil
  "t means always allow SPELLING server requests,
   :notify means allow requests, but notify the user,
   nil means never allow them, and
   :not-logged-in means allow them when no one is logged in.")
```

Next, add a conditional statement to the **spelling-server** function, similar to the one following. The conditional statement allows the connection to be rejected and control to be returned to the calling function.

```
(defun spelling-server ()
  (let ((conn (chaos:listen "SPELLER")))

    ;; protection begins
    (cond ((null *spelling-server-on*)
           (chaos:reject conn "The spelling server is off.")
           (return-from spelling-server))
          ((eq *spelling-server-on* :not-logged-in)
           (chaos:reject conn "This machine in use by someone else.")
           (return-from spelling-server))
          ((eq *spelling-server-on* :notify)
           (tv:notify nil "Spelling server in use by ~s"
                      (net:get-host-from-address
                        (chaos:conn-foreign-address conn) :chaos))))
    ;; protection ends

    (chaos:accept conn)
    (with-open-stream (stream
                    .
                    .
                    .
```

In this example, the code following `(stream` can be taken from the example in paragraph 5.6.2, with minor adjustments to parentheses.

Conns

5.8 A *conn* is one side of a Chaosnet connection. A conn is a named structure of type **chaos:conn**. The conn may have an actual connection attached to it; it may have a connection still being made; or it may record that a connection was refused, closed, or broken.

States of a Conn

5.8.1 A conn may be in any of the following states.

chaos:inactive-state Constant

The conn is not in use.

chaos:rfc-sent-state Constant

The conn was used to request a connection to another process, but no reply has yet been received. When the reply is received, it may change the state of the conn to **chaos:answered-state**, **chaos:cls-received-state**, or **chaos:open-state**.

chaos:listening-state Constant

The conn is being used for listening. If the conn receives an RFC packet with the contact name for which it is listening, the state of the conn changes to **chaos:rfc-received-state**.

chaos:rfc-received-state Constant

An RFC packet has arrived with the exact name for which the conn was listening. You can accept, reject, forward, or answer the request. Accepting the request changes the state of the conn to **chaos:open-state**. Answering, refusing, or forwarding the request changes the state to **chaos:inactive-state**.

chaos:open-state Constant

The conn is now one end of an open connection. You can send and receive data packets, including any packets that were waiting.

chaos:answered-state Constant

The conn was used to send an RFC packet, and an ANS packet was received in response. In other words, an answer to a simple transaction arrived, and you can then read the ANS packet.

chaos:cls-received-state Constant

The conn has received a CLS packet. In other words, the connection was closed or refused. You can read any data packets that were received before the CLS packet. After reading them, you can read only the CLS packet.

chaos:los-received-state Constant

The remote host has sent an LOS packet stating that the conn does not have the connection with the remote host that the conn assumed it had.

chaos:host-down-state Constant

The host at the other end of the connection has not responded to anything sent by the conn for a significant period of time.

chaos:foreign-state Constant

The connection is being used with a foreign protocol that is enclosed in uncontrolled packets.

Accessor Functions for a Conn

5.8.2 The following accessor functions can be used to obtain information about a conn. Note that these functions are open-coded by the compiler.

chaos:conn-state *conn* Function

This accessor function returns the state of the *conn*.

chaos:conn-foreign-address *conn* Function

This accessor function returns the Chaosnet address of the remote host at the other end of the connection. To find out which host this is, use **net:get-host-from-address**.

chaos:conn-read-pkts *conn* Function

This accessor function returns an internally threaded chain of incoming packets that are available to be read from the *conn*. To read from this chain, use **chaos:get-next-pkt**.

If the result returned is not **nil**, there are incoming packets available to be read by the application program.

chaos:conn-window-available *conn* Function

This accessor function returns the maximum number of packets you can transmit before the network software forces you to wait for the receiver to read some of them. By the time you actually send that many packets, the receiver might indicate there is room for more. The **chaos:send-pkt** function waits for this number to take on a nonzero value.

chaos:conn-plist *conn* Function

This accessor function returns the properties that have been defined for *conn*.

chaos:contact-name *conn* Function

This accessor function returns the contact name with which *conn* was created. The contact name is not significant to the functioning of the connection, once the connection has been established. But the contact name is saved in case debugging is needed.

Wait Function on a Conn

5.8.3 You can use the following function with a connection to set a timeout.

chaos:wait *conn state timeout* &optional *whostate* Function

This function waits until the state of the *conn* is not equal to the symbol *state* or until a set time has elapsed. The set time is equal to the value of *timeout*, which is given in units of 1/60th of a second. For example, a value of 600 specifies a timeout of ten seconds. If a timeout occurs, **nil** is returned; otherwise, **t** is returned. As an option, you can specify *whostate*, which indicates the process state to put in the status line of the Explorer screen. If you do not specify *whostate*, the default is `Chaosnet wait`.

Opening and Closing Connections

5.9 You can use the following functions and variables to open and close a connection. This paragraph is divided into two parts: the user side and the server side.

User Side of the Connection

5.9.1 You can use the following functions and variables to open and close a connection from the user side of the connection.

chaos:connect *host contact-name* &optional *window-size timeout* Function

This function opens a connection. If successful, it returns a network connection object; if not, it signals a fatal **network-error** condition. A network connection object is important because it can be used as input to other functions that require you to supply an argument for *conn*. The *host* may be the Chaosnet address or a string containing the name of a known host. The *contact-name* argument is a string containing the contact name and any additional arguments to go in the RFC packet. If not specified, the value of *window-size* is 13 by default. If not specified, *timeout* is 600, which is equal to 10 seconds.

chaos:simple *host contact-name* &optional *timeout* Function

This function performs the user side of a simple transaction. If successful, the server returns an ANS packet; if not, a **network-error** condition is signaled. The ANS packet should be disposed of when you are done with it; use the **chaos:return-pkt** function for that purpose. The *host* argument can be the Chaosnet address or a string containing the name of a known host. The *contact-name* argument is a string containing the contact name and any additional arguments to go in the RFC packet. If not specified, *timeout* is 600, which is equal to 10 seconds.

chaos:remove-conn *conn* Function

This function makes *conn* null and void. The *conn* becomes inactive, all its buffered packets are freed, and the corresponding Chaosnet connection—if any—goes away.

chaos:close-conn *conn* &optional *reason* Function

This function closes and removes the connection. If the connection is open, a CLS packet is sent from it containing *reason*, which is a string that contains an explanation for the closure. Do not use this function to reject a request; use **chaos:reject** for that purpose.

chaos:open-foreign-connection *foreign-host foreign-index* &optional *pkt-allocation distinguished-port* Function

This function establishes the local side of a connection that can be used to transmit and receive foreign protocols that are enclosed in uncontrolled packets.

This function is typically used to set up a transmission channel between an Explorer and a non-Explorer machine, which is referred to as the foreign host. Since you are not setting up a connection but only the local side of one, you provide to this function information concerning the foreign host. The *foreign-host* argument can be the Chaosnet address of the foreign host or a valid host name that is known on the network. The *foreign-index* argument is a number in the range of 0 through 127 decimal that you use to represent the conn. Together, *foreign-host* and *foreign-index* make up the destination address for packets that are sent with **chaos:send-unc-pkt**.

The *pkt-allocation* argument indicates the maximum number of input packets that can be buffered, which correlates to the window size. If not specified, the value of *pkt-allocation* is 10 by default. If specified, the value for *distinguished-port* determines the number by which the local index is set. A value for *distinguished-port* is necessary for protocols that define the meanings of particular index numbers.

Server Side of the Connection

5.9.2 You can use the following functions and variables to open and close a connection from the server side of a connection.

chaos:listen *contact-name* &optional *window-size wait-for-rfc* — Function

This function waits for a request for the specified contact name to arrive. It then returns a network connection object that will be in **chaos:rfc-received-state**.

The *contact-name* argument is a string containing the contact name. If not specified, the value of *window-size* is 13 by default. The maximum window size is 128. If not specified, the value of *wait-for-rfc* is **t**. If the value of *wait-for-rfc* is specified as **nil**, then the conn is returned immediately without waiting for an RFC to arrive.

chaos:server-alist — Variable

This variable contains an entry for each server that always exists. An entry is a list composed of a contact name, a form to be invoked, and two items that are used for accounting by the **add-initialization** function.

When an RFC arrives for one of these servers, the specified form is evaluated in the background process. Typically, it creates a process that then performs a **chaos:listen**. Use the **add-initialization** function to add entries to this list.

add-initialization *name form* &optional *keywords* (*list-name* '**sys:warm-initialization-list**) — Function

This function adds an initialization with *name* and definition *form* to an initialization list.

The *name* argument should be a string and *form* an expression to be evaluated later.

The keywords can be one keyword or a list of them. The keywords can be in any package. Keywords can be **:head-of-list**, meaning add to the front of the list rather than to the end, **:cold**, **:warm**, **:once**, **:system**, **:before-cold**, **:login**, **:logout**, **:site**, **:site-option**, **:full-gc** or **:after-full-gc**, specifying a list, or **:now**, **:first**, **:normal** or **:redo**, indicating when to run the initialization. The **:now** argument specifies to run the initialization as well as adding to the list; **:first** means run the initialization now if it is not on the list; **:normal** means do not run the initialization now; **:redo** means do not run it now, but mark it as never having been run even if it is already on the list and has been run.

If the keywords do not specify the list, **:list-name** is used. The default for **:list-name** is **sys:warm-initialization-list**.

chaos:accept *conn* Function

This function accepts a request for a connection. Before you can use this function, the connection must be in a **chaos:rfc-received-state**. In other words, *conn* must have received a valid request. 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**—the packet is discarded. If the RFC packet contains arguments in addition to the contact name, you should read the packet before accepting the request.

chaos:reject *conn reason* Function

This function rejects a request for a connection. Before you can use this function, *conn* must be in **chaos:rfc-received-state**. This function sends a CLS packet that contains *reason*, which is a string that explains why the conn is being rejected, and the *conn* is removed.

chaos:forward-all *contact-name host* Function

This function changes the host that is associated with *contact-name*, from the present host to that designated by *host*. All future requests for connection that specify *contact-name* are forwarded to *host*.

The *contact-name* argument is a string containing the contact name and any additional arguments to go in the RFC packet. The *host* argument can be the Chaosnet address or a string containing the name of a known host.

chaos:answer-string *conn string* Function

This function sends an ANS packet that contains *string*, and *conn* is removed. Before using this function, *conn* must be in **chaos:rfc-received-state**.

chaos:answer *conn pkt* Function

This function transmits *pkt* as an ANS packet, and *conn* is removed. Before using this function, *conn* must be in **chaos:rfc-received-state**. The *pkt* argument must be a released packet, which means that either the **chaos:get-pkt** function has allocated it or that Chaosnet has given it to the application. Use this function when the answer is made up of binary data rather than a string of text.

chaos:fast-answer-string *contact-name string* Function

If a pending request exists to *contact-name*, this function sends an ANS packet that contains *string* in response to the request, and **t** is returned. Otherwise, **nil** is returned. This function involves the minimum possible overhead. No connection is created.

The *contact-name* argument is a string containing the contact name and any additional arguments to go in the RFC packet.

Stream Input and Output

5.10 The following functions, variables, and methods are available for stream input and output.

chaos:open-stream *host contact-name* &rest *options* Function

This function opens a Chaosnet connection and returns a stream that performs I/O. The host to which you want to connect is specified by *host*, and the contact name for that host is specified by *contact-name*. If the value of *host* is **nil**, the function listens for *contact-name* on the local host, calls the **chaos:accept** function, and returns a stream object that represents the packets for that connection. The arguments *host* and *contact-name* are passed along to the function **chaos:connect** or to the function **chaos:listen**.

The *host* argument can be the Chaosnet address or a string containing the name of a known host. The *contact-name* argument is a string containing the contact name and any additional arguments to go in the RFC packet.

A list of alternating keywords and values is specified by *options*, which is made up of the following:

:window-size — This numeric argument specifies the number of packets that can be sent at one time without being acknowledged. It is passed to the function **chaos:connect** or to the function **chaos:listen**. The default is 13.

:timeout — This numeric argument is passed to the function **chaos:connect** or to the function **chaos:listen**. For example, a value of 600 specifies a timeout of 10 seconds, which is the default.

:error — If the value of this function is **nil**, the argument returns a string that explains the reason for an error, in the event that one occurs. If the value of this argument is **t**, which is the default, a failure to connect generates a Lisp error and an error condition is signaled.

:direction — This argument is passed to the function **chaos:make-stream**. The value of this argument can be **:bidirectional**, **:input**, or **:output**. The default is **:bidirectional**.

:characters — This argument is passed to the function **chaos:make-stream**. Specify **t** for the argument if the stream contains characters; **nil** if it does not. The default is **t**.

:ascii-translation — This argument is passed to the function **chaos:make-stream**. Specify **t** for the argument if characters are present in the stream and need to be translated from ASCII to the Explorer character set. Otherwise, specify **nil**, which is the default.

chaos:make-stream *conn* &key **:direction :characters :ascii-translation** Function

This function creates and returns a stream that performs I/O on *conn*, which should be open as a stream connection. The **:direction** option can take on the value **:input**, **:output**, or **:bidirectional**. The default is **:bidirectional**.

If the value for **:characters** is **nil**, the stream reads and writes 16-bit bytes. If the value for **characters** is not **nil**, the stream reads and writes 8-bit bytes. The default is not **nil**.

If the value for **:ascii-translation** is not **nil**, characters written to the stream are translated from ASCII to the Explorer character set. The default is **nil**.

:foreign-host Method of *chaos streams*

This method returns the host object for the host at the other end of this stream's connection.

:close Method of *chaos streams*

This method sends a CLS packet and removes the connection. If the stream can transmit data, the method first sends the stream an **:eof** message.

:force-output Method of *chaos output streams*

This method forces any buffered data to be transmitted. Normally, output is accumulated until a packet's worth of bytes is ready to be transmitted so that a full packet is sent.

:finish Method of *chaos output streams*

This method waits until either all packets have been sent and acknowledged or until the connection ceases to be open. If the operation is successful, **t** is returned. If the connection goes into a bad state, **nil** is returned.

:eof Method of *chaos output streams*

This method forces any buffered output to be transmitted. It then sends an EOF packet and performs a **:finish** method.

:clear-eof Method of *chaos input streams*

This method allows you to read past an EOF packet during input. Until a **:clear-eof** message is sent, one of the following occurs if any **:tyi** message is sent after an EOF:

- An end-of-file error is signaled.

- The message returns **nil**.

Note that the **:tyi** method itself has an option that controls whether **nil** is returned when an end of file is reached.

Packet Input and Output

5.11 Using the functions in this paragraph, you can perform input and output on a Chaosnet connection at the packet level. A packet is represented by a **chaos:pkt** data structure. Allocation of packets is controlled by the system. Each packet that it gives you must be given back. There are functions to convert between packets and strings. A packet is an **art-16b** array containing the packet header and data. For more information on arrays, refer to the *Explorer Lisp Reference* manual. The leader of a packet contains a number of fields used by the system.

chaos:first-data-word-in-pkt Constant

The value of this constant is the offset to the first 16-bit word of user data—following the header data—in any packet.

chaos:pkt-opcode *pkt* Function

This function allows you to get or set the opcode field of *pkt*. To set the opcode, use the following statement:

```
(setf (chaos:pkt-opcode my-pkt) my-opcode)
```

The system provides names for all the standard opcodes. The names that are useful in an application program appear at the end of this numbered paragraph.

chaos:pkt-nbytes *pkt* Function

This function allows you to get or set the number-of-data-bytes field of *pkt*. This field indicates how much of the data within *pkt* is valid, measured in 8-bit bytes. The field can be also be set with the **setf** statement. The maximum number of data bytes is 488.

chaos:pkt-string *pkt* Function

This function can be used to get or set an indirect array that fills the data field of *pkt* as a string of 8-bit bytes. The length of this string is equal to the value of **chaos:pkt-nbytes**. If you wish to record the contents of *pkt* permanently, you must copy this string.

chaos:set-pkt-string *pkt* &rest *strings* Function

This function concatenates *strings*, copies them into the data field of *pkt*, and sets the number-of-data-bytes field of *pkt* accordingly.

chaos:get-pkt Function

This function allocates a packet for your use. The packet must be returned by either the **chaos:return-pkt** function or the **chaos:send-pkt** function.

chaos:return-pkt *pkt* Function

This function returns *pkt* to the system for reuse. The packets given to you by the **chaos:get-pkt**, **chaos:get-next-pkt**, and **chaos:simple** functions should be returned in this way when you are finished with those packets.

chaos:send-pkt *conn pkt* &optional *opcode* Function

This function transmits the packet specified in *pkt* on *conn*. The *pkt* must be a released packet allocated by **chaos:get-pkt**, and it must have its data field and its number-of-data-bytes field filled in. If *conn* is not open, an appropriate network-error condition is signaled.

The *opcode* argument allows you the option of specifying an opcode for the packet. If specified, *opcode* must be either EOF or a data opcode of 200 or more. Its default is equal to the value of **chaos:dat-op**.

Note that giving a packet to the **chaos:send-pkt** function is the same as giving it back to the system. You do not need to call the **chaos:return-pkt** function.

chaos:send-string *conn* &rest *strings* Function

This function obtains and sends a data packet that contains the concatenation of *strings* as its data.

chaos:send-unc-pkt *conn pkt* &optional *pktn-field ack-field* Function

This function transmits *pkt*, an uncontrolled packet, on *conn*. The opcode packet number and acknowledge fields in the packet header are filled in. The optional arguments for *pktn-field* and *ack-field* allow you to use these fields for your own purpose.

The default for *pktn-field* is the value of *pktnum*, whose default is set to *pkt*. The default for *ack-field* is the value of *pkt-ack-num*, whose default is set to *pkt*.

chaos:may-transmit *conn* Function

This function is a predicate that returns **t** if there is any space in the window for transmitting on *conn*. A **t** means you can transmit immediately. If the function returns **nil**, you may have to wait before you can transmit data.

chaos:finish-conn *conn* &optional *whostate* Function

This function waits for either all packets to be sent and acknowledged or for the connection to cease being open. If successful, the function returns **t**. If the connection ceases to remain open, the function returns **nil**. For *whostate*, specify the process state you want to display in the status line you wait for the function to complete. `Net Finish` is the default.

chaos:get-next-pkt *conn* &optional *no-hang-p whostate check-conn-state* Function

This function returns the next input packet from *conn*. Since the system releases the packet to you, you must give it back to the system with the **chaos:return-pkt** function.

If you specify the *no-hang-p* argument, giving **t** as its value, the function returns **nil** if there are no packets available or if the connection is not open. If the value of *no-hang-p* is **nil**, which is the default, and the connection is not open, an error is signaled. The following condition names indicate which conditions could be signaled:

- **chaos:host-down-state**
- **chaos:los-received-state**
- **chaos:read-on-closed-connection**

If the value of *no-hang-p* is **nil** and if no packets are available, the **chaos:get-next-pkt** function waits for packets to come in or for the state of the connection to change.

The default value for *whostate* is `Chaosnet Input`.

The *check-conn-state* argument is a Boolean argument that specifies whether the state of the conn is checked. Its default is the opposite of the *no-hang-p* argument.

chaos:data-available *conn* Function

This function is a predicate that returns **t** if there are any input packets available from *conn*.

The following are symbolic names that you need to know for the opcodes used in chaos packets. A description of the part of the packet that contains data is given as well.

chaos:rfc-op Constant

This constant has as its value an opcode that is used for requesting a connection. The data consists of the contact name, terminated by a space character. As an option, additional data can follow the space character; the interpretation of such additional data is the responsibility of the server for that contact name.

chaos:lsn-op Constant

This constant has as its value an opcode that is used when you ask to listen for a contact name. The data is simply the contact name. This packet is never actually sent over the network. Instead, the contact name is kept within the Chaosnet software, which compares it to the contact names in the RFC packets that arrive.

chaos:opn-op Constant

This constant has as its value an opcode that is used by the server process to accept the request for a connection conveyed by an RFC packet. Its data serves only internal functions.

chaos:ans-op Constant

This constant has as its value an opcode that is used to send a simple reply. The server sends a simple reply in place of opening a connection.

chaos:los-op Constant

This constant has as its value an opcode used in a packet that you receive if you try to use a connection that has been broken or that no longer exists. Its data may be a message that explains the situation and that can be printed out.

chaos:cls-op Constant

This constant has as its value an opcode used in a packet that is sent by one end of a connection to close the connection. Its data is a message that explains the reason for closure and that can be printed out. Note that the other side of a connection cannot depend on receiving a CLS packet because it is not retransmitted if it is lost. If a CLS packet is lost during its transmission, the other side—thinking that the connection is still open—receives an LOS packet the next time it tries to use the connection.

CLS packets are also used to refuse to open a connection in the first place. In this case, use the **chaos:reject** function to send the CLS packet.

chaos:eof-op Constant

This constant has as its value an opcode that is used to indicate the end of the data for transmission. When the EOF packet is acknowledged by the other process, you know that all the data was received properly. You can wait for this acknowledgment with the **chaos:finish-conn** function. An EOF packet carries no data itself.

chaos:dat-op Constant

This constant has as its value an opcode of 200 octal, which is the normal opcode for 8-bit user data. Some protocols use multiple data opcodes in the range 200 through 277 octal, but simple protocols that do not need to distinguish between different types of packets simply use opcode 200 octal.

Connection Interrupts

5.12 The following functions are available for interrupting a connection.

chaos:interrupt-function *conn* Function

This function specifies whether another function is stored as an attribute of *conn*. If a function is stored as an attribute, it is called when certain events occur on the connection. Normally, the value of **chaos:interrupt-function** is **nil**, which means not to call any function.

If you choose to store a function as an attribute, you can use the **setf** function. Since a function that is stored as an attribute is called in the Chaosnet background process, the function should not perform any methods that might have to wait for the network, because waiting for the network could permanently hang up the background process.

The value of the stored function is an object that can be called as a function. The function must accept two arguments and can optionally take a third.

The first argument of the function can be one of the following:

:input — A packet has arrived for the connection when the connection did not have any input packets queued. The **chaos:get-next-pkt** function can be invoked without waiting. There are no additional arguments.

:output — An acknowledgment has arrived for the connection, making space in a window that was formerly full. Additional output packets can now be transmitted with the stored function without waiting. There are no additional arguments.

:change-of-state — The state of the connection has changed. A third argument for the stored function is necessary, and the third argument is the symbol for the new state.

The second argument of the function is the network connection object that is associated with *conn*.

The optional third argument of the function is the symbol for the new state of the connection. It is present only if **:change-of-state** is the first argument.

In the following example, the `setf` statement stores the `pkt-historian` function as an attribute of a conn. Both `reason` and `conn` represent arguments that can be called by a function.

```
(defun pkt-historian (reason conn)
  (if (eq reason :input)
      (push (copy (read-pkts conn)) *input-history*)))

(setf (chaos:conn-interrupt-function conn) #'pkt-historian)
```

chaos:pkt-link *pkt* Function

This function returns the next packet in the list that *pkt* was part of. If no packets are available, the macro returns **nil**. To alter the next packet after *pkt*, use **setf**.

Information and Control

5.13 The following functions are available to help you gain information or control different host attributes.

chaos:print-conn *conn* &optional *short-pkt-display* — Function

This function prints everything it knows about the specified connection. If the value of *short-pkt-display* is **nil**, the function also prints everything the system knows about each queued input and output packet on the connection. The default value of *short-pkt-display* is **t**.

chaos:print-pkt *pkt* &optional *short-pkt-display* — Function

This function prints everything the system knows about the packet. If *short-pkt-display* is **t**, only the first line of the information is printed. The default value of *short-pkt-display* is **nil**. The data field is printed as a string, so binary data is not printed in any meaningful way.

chaos:print-all-pkts *pkt* &optional *short-pkt-display* — Function

This function calls the **chaos:print-pkt** function on *pkt* and on all packets on the threaded list emanating from it. If *short-pkt-display* is **t**, only the first line of the information is printed. The default value of *short-pkt-display* is **t**.

net:halt *stop?* — Function

If *stop?* is not **nil**, all network operations for this host are halted. If *stop?* is **nil**, the network is enabled for this host.

chaos:reset &optional *enable-p* — Function

This function resets the Chaosnet for this host. If the value of *enable-p* is **nil**, which is the default, the function turns off the network for this host. If *enable-p* is not **nil**, the network is enabled and turned on. This function does not override the **net:halt** function.

chaos:assure-enabled — Function

This function turns on the network for this host if it is not already on. The network is normally on unless you call the **chaos:disable** function. This function does not override the **net:halt** function.

chaos:enable — Function

This function turns on the Chaosnet network for this host.

chaos:disable — Function

This function turns off Chaosnet network access for this host.

THE GENERIC NETWORK SYSTEM

6

Introduction

6.1 This section describes the *generic network system*, a standard interface to the transport, session, and presentation layers of the networking protocols available for the Explorer system (such as Chaosnet and TCP).

The servers on a network implement services in various protocols. The generic network system allows you to access these services transparently, that is, without knowing protocol-specific information.

The generic network system provides these advantages:

- The ability to write protocol-independent applications and services
- Access to all of the services available with the installed set of network protocols
- A set of standardized error conditions

Two interface modules form the generic network system:

- The generic network interface (GNI)
- The generic services interface (GSI)

The following paragraphs discuss both these modules, providing information about the flavors, functions, and conditions available to you in GNI and GSI. Finally, this section tells how to implement generic network servers and how to define generic services.

The Generic Network Interface

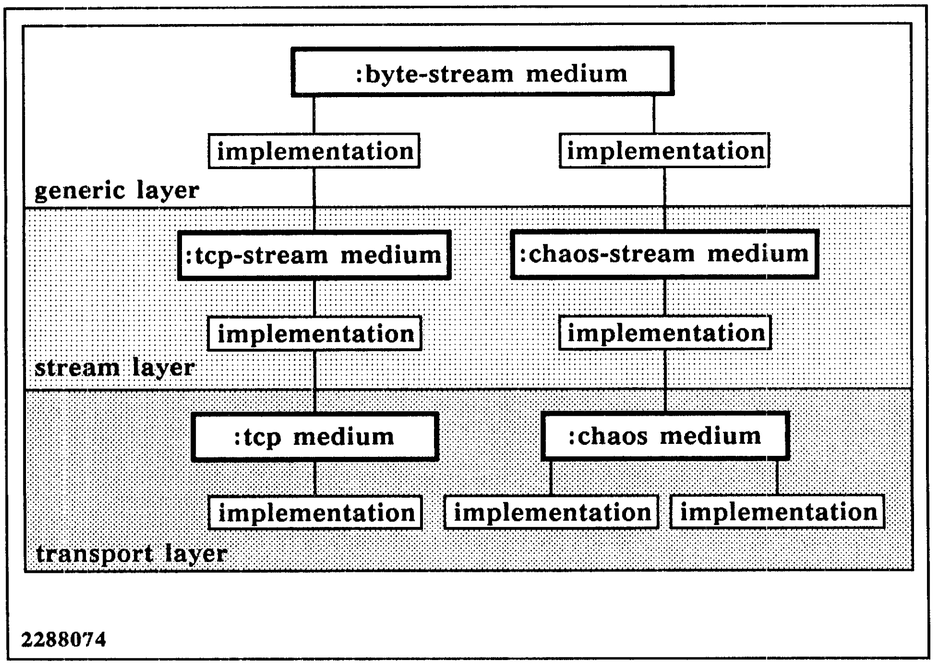
6.2 The function of GNI is to determine the most efficient way to provide services that are available on the network (such as file transfer, remote login on a virtual terminal, mail, and so on). This interface allows a user process to request a network connection to a remote process using varying degrees of specificity about the network layer protocol to be used.

For instance, if you know that a particular service is available on a Chaosnet network, then you can request that Chaosnet implementation. On the other hand, if all that your user process requires is a byte stream for input, then it can request a simple byte stream connection. GNI determines which byte stream implementation is best for making the connection between user and server processes, returning a byte stream (such as a Chaos-stream or TCP-stream) based on the protocol and implementation that it has determined to be best. Your user process need not specify the specific lower-level stream connection to be returned. GNI returns the best available stream connection to the requested service.

Mediums, Layers, and Connections

6.2.1 A *medium* is a structure that returns a network connection. Depending on the type of connection returned by the medium, that medium is said to exist in one of three possible *layers*: the generic layer, the stream layer, and the transport layer. In Figure 6-1, the **:byte-stream** medium exists in the generic layer, the **:tcp-stream** and **:chaos-stream** mediums in the stream layer, and the TCP and Chaos mediums in the transport layer.

Figure 6-1 **Mediums and Layers**



A transport-layer medium returns only transport connection objects (such as Chaosnet conns), while a stream-layer medium returns only stream connection objects (such as Chaos-stream connections). With these three medium layers, a user process has a choice about how specific a connection request needs to be.

For example, if process A (spawned by the QFILE protocol) requests a connection, it requires no stream I/O at all. The QFILE protocol takes care of all packet assembly and shipping. Therefore, process A would issue its connection request specifying that the **:chaos** medium be used.

Process B, on the other hand, may require a TCP stream for its input connection. If this is the case, and process B is actually aware that the connection must be made via TCP, it can issue its connection request by specifying the **:tcp-stream** medium. However, if process B does not know what protocol is necessary for the connection (only that a byte stream is required), it can issue its connection request by specifying the **:byte-stream** medium. In either case, process B is assured of getting the proper type of connection.

Medium Implementations

6.2.2 An *implementation* performs the actual steps necessary to return a connection for a given medium. A medium can have one or more implementations by which it establishes a connection of a particular type. For example, the generic byte stream medium shown in Figure 6-1 has two implementations that return stream objects; one returns a TCP stream object, and one returns a Chaosnet stream object.

All implementations of the medium should return the *same type* of connection. To guarantee that a medium returns a byte stream, all of the implementations of that medium must return connections on byte streams.

When a connection request specifying a particular medium is issued, the request invokes that medium's implementations one-by-one until one succeeds. If the **net:*try-all-medium-implementations*** variable is non-**nil** and none of the implementations can make the connection, a proceedable error is signaled.

When a medium's implementation is invoked, it executes its *connection steps*. These steps direct the medium to call a connect function or call another medium. Connection steps appear as a list of keyword-designated actions, described in the following paragraphs.

:network *Step*

6.2.2.1 The **:network** step calls a connect function. For example, a connection step that would call the **chaos:chaos-connect-function** or the **chaos:listen-function** would appear as follows:

```
((:network :chaos))
```

The `:chaos` argument identifies the name of the network-layer protocol from which the connect function is called.

:medium *Step*

6.2.2.2 The **:medium** step indirectly obtains a connection by requesting that connection from another (secondary) medium.

The secondary medium can have another **:medium** step (taking the level of indirection one step further), or it can have a **:network** step that calls the connection function. A **:medium** step would appear as follows:

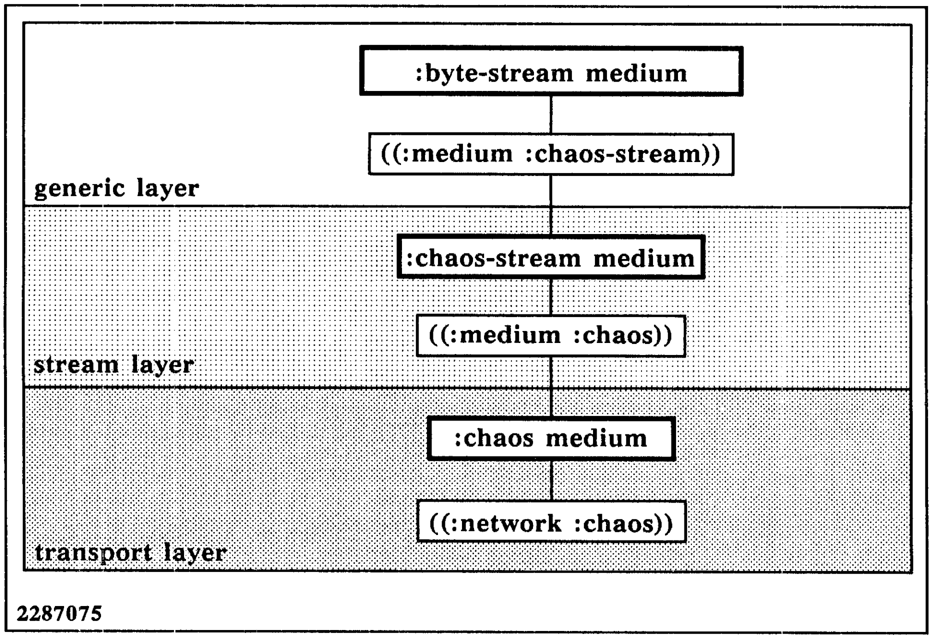
```
((:medium :chaos))
```

When a connection is finally obtained, it is passed as a returned value up through each medium in the calling sequence until it reaches the original medium making the connection request. If the connection object is passed from medium B to medium A as is, that is, without altering it in any way, then medium A is said to be the *superior* of medium B.

However, a connection can be altered by any of the mediums in the calling sequence, as is shown in the following example based on Figure 6-2.

In Figure 6-2, notice that the boxes representing each medium's implementation now contain that implementation's connection steps.

Figure 6-2 **Implementations and Connection Steps**



When the **:byte-stream** medium requests a connection, its implementation immediately executes a **:medium** connection step, telling the **:chaos-stream** medium to obtain the connection.

When the **:chaos-stream** medium requests the connection, its implementation executes another **:medium** connection step, telling the **:chaos** transport-layer medium to obtain the connection.

The **:chaos** transport-layer medium requests the connection, and its implementation executes a **:network** connection step, telling the **:chaos** network-layer (not shown) to send a connection object (conn). Now the connection object is returned to the **:chaos** medium.

The **:chaos** medium, in turn, passes the connection object back up to the **:chaos-stream** medium, which binds the connection object to a stream object, and passes the newly bound stream object up to the **:byte-stream** medium.

In this situation, the **:byte-stream** medium is the superior of the **:chaos-stream** medium, because both mediums deal with the same type of connection object. However, the **:chaos-stream** medium is *not* the superior of the **:chaos** medium. This is because the **:chaos-stream** medium binds the connection object returned from the **:chaos** medium to a stream object before passing it upward. By modifying the returned connection object, the **:chaos-stream** medium is said to have *encapsulated* that object. Encapsulation, then, is one way of modifying a returned connection object for use by a superior medium.

Medium Desirability

6.2.3 Each implementation has an associated desirability. When a medium requests a connection from its implementations, the most desirable implementation is consulted first. The medium desirability is expressed as a number between 0 and 1, with the most desirable medium implementations having a value closer to 1. If the global variable **net:*try-all-medium-implementations*** is bound to **t**, then each succeeding implementation is consulted in inverse order of its desirability until one of them succeeds in returning a connection on the particular medium.

Logical Contact Names

6.2.4 When an application requests a connection to a remote server, it addresses that server using a *logical contact name*. Each service on the network has a logical contact name that is defined using GNI. This contact name is a structure whose name is a string. It also has an association list of pairs whose keys are network types and whose values are contact names appropriate to each particular network. For instance, a fortune cookie server that uses a byte stream network connection might have the name "FORTUNE" and associated value of 599. When making a connection to the fortune cookie server, a remote host addresses the server on port 599 when using TCP, but uses the contact name "FORTUNE" when using Chaosnet.

When a generic server listens for a contact name, it does so on all medium implementations that have translations for the logical-contact-name. The **net:define-logical-contact-name** function (described later in this section) defines a generic contact name if one does not already exist and adds to it a set of translations for the mediums that are implemented. The function **net:listen-for-connection-on-medium** listens for the logical contact name on a particular medium.

Defining a Medium

6.2.5 The function **net:define-medium** is used to define a medium object and add an implementation to it, or simply to add an implementation to an existing medium object. The **net:define-medium** function is discussed in paragraph 6.4.1, titled GNI Functions; however, the following paragraphs introduce you to some of the function's major arguments.

NOTE: The code shown in paragraph 6.2.5 has already been implemented in the Explorer system, and is not required to run the examples later in this section.

The following portions of code refer to the mediums used in Figure 6-2.

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

In the preceding line of code, the `net:define-medium` function defines a medium called `:byte-stream`. Only one argument is provided for `net:define-medium`, and that is `:byte-stream`, which is the *name* argument. The *name* argument is a keyword (or a string) that represents the name of the medium object being created. Because it has no *superior-medium-list* argument, the `:byte-stream` medium has no superiors. Now consider the medium definition for the **:chaos-stream** medium:

```
(net:define-medium :chaos-stream
  :connection-steps '((:medium :chaos))
  :medium-desirability .85
  :implementation-desirability .85
  :superior-medium-list '(:byte-stream)
  :listen-function 'chaos-stream-listen-function
  :connect-function 'chaos-stream-connect-function)
```

In this code, another `net:define-medium` function defines the `:chaos-stream` medium. This medium has as its superior the `:byte-stream` medium previously created. The rest of the arguments are discussed in the definition of the **net:define-medium** function, found in paragraph 6.4.1. The final medium to be defined is the **:chaos** medium:

```
(net:define-medium :chaos
  :connection-steps '((:network :chaos))
  :medium-desirability .8
  :implementation-desirability .8
  :connect-function 'chaos-stream-connect-function
  :connection-possible-p-function 'chaos-connection-possible-p-function
  :add-server-function 'chaos-add-server-function
  :delete-server-function 'chaos-delete-server-function
  :listen-function 'chaos-listen-function)
```

Again, with no *superior-medium-list* argument, the `:chaos` medium has no superiors.

Medium Connections

6.2.6 The functions **net:open-connection-on-medium** and **net:listen-for-connection-on-medium** are used as connect functions by an implementation's connection function or listen function.

Again, these functions are discussed in detail later in this section (in paragraph 6.4.1, titled GNI Functions).

The Generic Services Interface

6.3 GSI allows an application to request a network service, such as file transfer, status, or time, without specifying the exact network implementation that is to be used. Rather, when a request for a particular service is made, the GSI looks at the network services currently available and selects the most appropriate implementation of the service to use. If the **net:*try-all-service-implementations*** variable is non-**nil**, GSI tries every implementation of a service until one of them is successful.

Services

6.3.1 A *service* is a generic utility that you can access over a network. A service refers abstractly to a specific generic function available to hosts on a network. Examples include Mail, remote login (virtual terminal), and file transfer services, each of which refers to the generic service provided, without specifying the precise nature of their implementations. These implementations can differ considerably from one network to another depending upon the type of network protocols being used.

For example, file transfer services are provided by QFILE for the Chaosnet family of protocols, and by FTP for the TCP/IP family of protocols. Nevertheless, these two file transfer utilities perform (essentially) the same task: they transfer files from one host to another. Likewise, the mail service is provided by the Mail protocol on Chaosnet and by the SMTP protocol on Chaosnet and IP.

Again, these protocols play a similar role on their respective networks. Therefore, the (abstract) generic services **:file** and **:mail** can be said to exist independently of the type of network and lower-level network-specific protocols by which users and servers interact.

A service definition must have the following information:

- How the service is accessed
- Specifications of the operations (methods) that must be supported by any particular implementation of the service
- What side effects are expected to result from a call to the service
- Any returned values

Service Implementation

6.3.1.1 A generic service may have one or more implementations. An implementation tells the generic services interface how to access a particular server and carry out the operations that it allows. Service implementations are flavor instances that implement the utility defined by the service definition. In defining the generic service **:file**, you mix in the **service-implementation-mixin** flavor with the service to be provided.

Service Operations

6.3.1.2 Each separate operation that a service offers is a different method of the service flavor. For example, if QFILE is a service implementation of the FILE service, then the **:delete** method must be defined to provide the file deletion operation. To delete the file `"romeo:mercutio;play.text#3"`, you should be able to enter the following:

```
(send (find-service-implementation :qfile)
     :delete "romeo:mercutio;play.text#3")
```

The **net:define-service** macro (discussed later in this section) allows you to specify the keyword arguments that are passed to the service implementation. You can pass anything that exists in the method's environment, such as *self* (the host) or the method's instance variables. Two useful keyword arguments you should specify are **:host** and **:medium**.

The keyword **:host** identifies the host from which the service is requested.

The keyword **:medium** identifies the medium by which the service is provided. Each service implementation verifies that the connection provided by the medium is acceptable to it. This verification is done by defining the medium that the service implementation is designed to run on and by testing the medium that is passed to see if the desired medium is one of its superiors.

Service Implementation Arguments

6.3.1.3 An implementation of a service must handle all of the arguments that could be sent to any of the other implementations of the same service, and it should ignore any arguments not understood. In the simplest case, the various implementations of a service could be written in such a way as to accept exactly the same set of arguments. For instance, you could have a `:chat` service that has IP and Chaosnet implementations. A user passes the client side function `chat` the name of the target host and a message, in that order. The server side then prints the message on the screen of the target host. In writing the IP and Chaosnet code, you could arrange for both the Chaosnet implementation and the IP implementation to take the same arguments. Then, in genericizing the service, you provide the means for the generic server to handle the target host name and the message.

A more complicated situation can arise when two existing servers take different arguments, but provide exactly the same generic service. Suppose, for instance, that you have a generic service called `:weather-report` that has two implementations. A user invokes the Chaosnet implementation of the `chaos:weather-bureau` server with a Chaosnet-specific client side function called `weather-forecast`. The fictitious `chaos:weather-forecast` function takes the two arguments *city* and *state*. The `weather-report` service also has a TCP/IP implementation called `IP:weather` that is normally invoked by the IP specific client-side function `report-on-the-weather`. The latter function takes four arguments: *time, date, city,* and *state*.

A generic weather service for these two implementations can take one of two courses.

It can accept the four arguments *city, state, time,* and *date*. In this situation, if the generic `:weather-report` service makes a connection on the **:chaos-stream** medium (thereby using the Chaosnet `weather-forecast` server), the additional *time* and *date* arguments passed to the generic server must be ignored.

It can accept two arguments, *city* and *state*. In this situation, if the generic `:weather-report` service makes a connection on the **:chaos-stream** medium (thereby using the Chaosnet `weather-forecast` server), you can have the Chaosnet `weather-forecast` server generate its own additional *time* and *date* arguments.

It may be the case that one implementation of a service does not support all of the operations defined for the service. One operation may be allowed by one implementation, but not by another. For example, QFILE supports non-destructive deletions on hosts that support this utility, but FTP does not. In this case, what *delete* means to FTP is what *delete-and-expunge* means to QFILE. FTP only handles the delete-and-expunge operation. If a user wishes to delete a file, the generic services interface chooses to do so via QFILE, the chaosnet implementation of the generic FILE protocol.

Hosts

6.3.2 A host is an object that represents a physical host on a network. A user process can call a generic service without specifying which service implementation to use by sending the host object a service message such as the following:

```
(send (net:parse-host 'thor) :file :delete "thor:jones;foo.text")
```

Most of the attributes of a host pertain to defining the host object in the network namespace, as discussed in Section 4, Getting on the Network. In order to access generic services, the most important of these attributes is the *service attribute.*

Service Attributes

6.3.2.1 Each host has a *service attribute* whose value is a list of *service-medium-protocol* triples for each service that the host provides to the network.

For instance, if the host PONY-EXPRESS provides the mail service for the network, in addition to the usual mail and Telnet (login) services, then the **:service** attribute is as follows:

```
((:mail :chaos :chaos-mail)
 (:login :chaos :telnet))
```

It is possible for a host to have more than one entry for a particular service that it provides. This situation occurs when there is more than one way to access a particular service on the given host.

For instance, a host on two networks, one of which is IP and the other Chaos, might provide mail-forwarding service on both networks. This host's **:service** attribute might include the following entries:

```
((:mail :chaos-stream :mail)
 (:mail :chaos-stream :smtp)
 (:mail :tcp-stream :smtp))
```

The first entry specifies that the mail service can be reached over the Chaos-stream medium via the Mail protocol. The second entry allows the mail service to be reached over the Chaos-stream medium again, only this time by using the SMTP protocol. The third entry specifies that the mail service can be reached on the TCP-stream medium via the SMTP protocol.

Service Implementation List

6.3.2.2 Each host object contains a list of the services and implementations that are supported by that host. The elements of this list are of the form (*service medium service-implementation*). This list indicates that this host can provide a *service* using *service-implementation* if a connection can be made on *medium*. When an application requires an operation provided by a service, the name of the service is sent a message to the host from which the service is desired. To delete the TAURUS:USER;FILE.TEXT#3 file, for example, use the following form:

```
(send (net:parse-host 'taurus) :file :delete "taurus:user;file.text#3")
```

A method created by the **net:define-service** macro handles this message. The method attempts to match the message with one of the elements of the services list. Once an element is found and a connection can be established on *medium*, then the *service-implementation* is called with the arguments that were passed in the original call.

If this call completes successfully, then the value returned by the *service-implementation* fails, indicating one of three situations:

- A connection was not possible on *medium*.
- *service-implementation* was not installed on this host.
- The call signals the **gsi-service-failure** error condition.

In any of these situations, another element of the services list is found and tried.

The Generic Programmatic Interface

6.4 There are a number of flavors, methods, functions and macros, as well as error conditions that allow you to create a new medium and to create and access generic services.

GNI Functions

6.4.1 The following functions are used to access GNI:

net:open-connection-on-medium *host medium logical-contact-name* &rest *args* &key **:timeout :timeout-after-open :window-size :stream-type** Function

This function opens a connection to the server process that is listening for *logical-contact-name* on some host on the network.

The *host* argument identifies the host on which the connection is to be made.

The *medium* argument is a keyword representing the name of the medium on which to make the connection. If a connection can be made on more than one media, then this argument is a list of keywords representing each medium.

The *logical-contact-name* argument is a string representing the contact name of the process to which the user is trying to establish a connection.

The *args* argument represents the arguments passed to the media on which you are trying to establish a connection. The value of *args* is one of the following keywords and its associated value.

:timeout — The timeout used when attempting to connect to a host.

:timeout-after-open — The timeout used when reading from a connection.

:window-size — The window size in bytes (byte size being determined by the protocol involved).

:stream-type — The type of stream to be returned (such as **:ascii-translating-character-stream**). For a complete list of stream types, see the following file: sys: chaosnet; medium.lisp#>.

net:listen-for-connection-on-medium *medium logical-contact-name* &rest *args* Function

This function listens for the *logical-contact-name* (of the server process) on each single-step implementation of the medium (identified by the *medium* argument) for which the *logical-contact-name* has a translation.

The *medium* argument is a keyword representing the name of the medium on which to listen for the contact name of the server. If there is more than one medium through which a service can be provided, then this argument is a list of keywords representing each medium.

The *logical-contact-name* argument is either a string representing the contact name for which the server process is listening, or the actual logical contact name object.

The *args* argument represents the arguments passed to the media on which you are trying to establish a connection. The value of *args* is one of the following keywords and its associated value.

:timeout — The timeout used when attempting to connect to a host.

:timeout-after-open — The timeout used when reading from a connection.

:window-size — The window size in bytes (byte size being determined by the protocol involved).

:stream-type — The type of stream to be returned (such as **:ascii-translating-character-stream**). For a complete list of stream types, see the following file: sys: chaosnet; medium.lisp#>.

net:add-server-for-medium *medium logical-contact-name function* Function

This function runs *function* when a connection request is received for *logical-contact-name* on any single-step implementation of *medium*. This function must be evaluated after calling **net:define-logical-contact-name**.

The *medium* argument can be a keyword, string, or medium object. It can also be a list of string or medium objects if more than one medium is available on which to make a connection to access the particular service.

The *logical-contact-name* argument can be either a string representing the contact name of the server process being solicited, or the actual logical-contact-name object itself.

The *function* argument represents the function to be evaluated when a connection request is received for *logical-contact-name* on any single-step implementation of *medium*.

net:delete-server-for-medium *medium logical-contact-name* Function

This function deletes the server identified by *logical-contact-name* on the medium identified by *medium*.

The *medium* argument can be a keyword, string, or medium object. This argument can also be a list of string or medium objects if more than one medium is available on which to make a connection to access the particular service.

The *logical-contact-name* argument can be either a string representing the contact name of the server process being solicited, or the actual logical-contact-name object itself.

net:find-medium *medium-name* &optional *error-p* Function

Returns the medium object identified by the *medium-name* argument. The *medium-name* argument can be a string or keyword.

The *error-p* argument, if non-**nil** and no medium is found, causes an error to be signaled.

net:define-medium *name* **:superior-medium-list :connection-steps** Function
:connect-function :connection-possible-p-function :listen-function
:add-server-function :delete-server-function :implementation-name
:implementation-desirability :medium-desirability
:implementation-desirability-function

The **net:define-medium** function creates a medium object for *name* (if such an object does not already exist) and adds one implementation to it.

An example of **net:define-medium**'s syntax would be as follows:

```
(net:define-medium :chaos
  :connection-steps '((:network :chaos))
  :connect-function 'chaos-stream-connect-function
  :connection-possible-p-function 'chaos-connection-possible-p-function
  :listen-function 'chaos-listen-function)
  :add-server-function 'chaos-add-server-function
  :delete-server-function 'chaos-delete-server-function
  :implementation-desirability .8
  :medium-desirability .8
```

The *name* argument is a keyword (or a string) that represents the name of the medium object being created.

The **:superior-medium-list** argument is a list of the media of which the current medium is an implementation; for example:

The **:connection-steps** argument is a list such as the following:

```
'((:medium :chaos))
```

The **;connect-function** argument is a function used to return a connection after the connection steps have been satisfied. The arguments to **:connect-function** are as follows: *host*, *logical-contact-name*, *connection*, and *args*. In certain cases, some of these arguments can be ignored.

The **:connection-possible-p-function** argument is a function or function name that returns **t** if it is possible to get a connection from one host to another on this medium. If in doubt, return **t**. The arguments to the **:connection-possible-p** function are *host-a* and *host-b*.

The **:listen-function** argument is a function that listens for connections to the logical contact name and returns an open connection object. The arguments to the **:listen-function** are *logical-contact-name*, *connection*, and *args*.

The **:add-server-function** argument identifies the function responsible for doing whatever is required for adding a server for the medium. Its arguments are *logical-contact-name* and *form*.

The **:delete-server-function** argument identifies the function responsible for doing whatever is required for deleting a server for the medium. It takes one argument: *logical-contact-name*.

The **:implementation-name** argument is the name of the implementation. This argument defaults to the name of the medium.

The **:implementation-desirability** argument is a number between 0 and 1, used to determine the order in which to use implementations. The default value for **:implementation-desirability** is the same as for **:medium-desirability**.

The **:medium-desirability** argument is a number between 0 and 1, used to determine the order in which to use mediums. The default value for **:medium-desirability** is the same as for **:implementation-desirability**.

The **:implementation-desirability-function** argument is a function that can be used to determine the medium desirability.

net:find-logical-contact-name *name* &optional *error-p* Function

This function returns the logical contact name object for the string *name*.

The *name* argument is a keyword (or a string) that represents the name of the service being solicited.

The *error-p* argument, if non-**nil** and *name* is not found, causes an error to be signaled.

NOTE: Although a logical contact name must have been defined prior to a call to the **net:add-server-for-medium** function, **net:find-logical-contact-name** need not be called prior to calling **net:add-server-for-medium**.

net:translate-logical-contact-name *name network-protocol* Function

This function returns the contact identifier to use on the *protocol* for *name*.

The *name* argument is a keyword (or a string) that represents the name of the service being solicited.

The *network-protocol* argument is a keyword representing the protocol to be used to access the service indicated by *name*.

net:define-logical-contact-name *name translations* Functions

This function adds *name* as a logical contact name, if it is not already defined, and adds translations to it.

The *name* argument is a keyword (or a string) that represents the name of the logical contact name to be added.

The *translations* argument is a list of logical contact name translation elements. An example is `((:chaos "MAIL")(:TCP 250))`.

net:connection-possible-p *medium host-a* &optional (*host-b* **sys:local-host**) Function

This predicate returns **t** if it is theoretically possible to create a connection from *host-b* to *host-a* on at least one implementation of *medium*.

The *medium* argument is a keyword (or string) that represents the medium or the actual medium object.

The *host-a* argument is a string representing the host name of the host to be contacted, or the actual host object itself.

The *host-b* argument is a string representing the host name of the host originating the connection, or the actual host object itself. The default binding of this parameter is **net:local-host**.

net:superior-medium-p *medium superior-medium* Function

This predicate returns **t** if *medium* is one of the implementations of *superior-medium*.

The *medium* argument is a string representing the medium, or the actual medium object.

The *superior-medium* argument identifies the name of the superior medium.

GSI Functions

6.4.2 The following functions allow you to access GSI programatically.

net:define-service *service-name service-arguments implementation-arguments* &optional *documentation* Macro

The **net:define-service** macro defines a generic service; that is, it defines a method (specified by the *service-name* argument) for a host object.

The *service-name* argument is a keyword that identifies the generic service (such as **:qfile** or **:ftp**).

The *service-arguments* argument is a list of arguments with which the method is defined.

The *implementation-arguments* argument is a list of the arguments that are actually passed to the implementation. This list can include items specified in the *service-arguments* argument, host instance variables, or special variables.

The optional *documentation* argument is a documentation string.

NOTE: The **net:define-service** macro has an additional argument called *operation*. However, do not include the *operation* argument in your service definition; the system does this automatically.

net:define-service-implementation *flavor-name* Function

The *flavor-name* argument represents the name of the service implementation flavor.

This function creates an instance of the flavor that can be found by the **net:find-service-implementation** function.

net:find-service-implementation *service* Function

This function finds an appropriate implementation of the service on the local host.

The *service* argument is a keyword specifying a service implementation.

net:service-implementation-mixin Flavor

This flavor should be mixed into any service implementation flavors that are created in order to give the object a common type and to add needed instance variables.

net:define-stream-type *stream-type flavor medium* Function

This function defines the flavor of stream to instantiate for *stream-type* on *medium*.

The *stream-type* argument is a keyword that will serve to identify the new generic service (such as **:qfile** or **:ftp**). Some of the acceptable types of streams are: **:binary-stream**, **:character-stream**, and **:ascii-translating-character-stream**. To see a list of the currently available stream types, evaluate the **net:*medium-stream-type-alist*** variable.

The *flavor* argument identifies the flavor name for the instantiation.

The *medium* argument identifies which medium the stream type operates in.

net:find-stream-type *stream-type medium* Function

This function returns the flavor of stream to instantiate for *stream-type* on *medium*.

The *stream-type* argument is a keyword that will serve to identify the new generic service (such as **:qfile** or **:ftp**). Some of the acceptable types of streams are: **:binary-stream**, **:character-stream**, and **:ascii-translating-character-stream**. To see a list of the currently available stream types, evaluate the **net:*medium-stream-type-alist*** variable.

The *medium* argument identifies which medium the stream type operates in.

Using the Generic Network Interface

6.5 The following paragraphs discuss the development of applications using GNI. The principal topic is the creation of generic network servers, their implementation on server hosts, and the ways by which you can access the service provided. Several example servers are discussed in detail.

Chat — A Simple Server

6.5.1 The first example server, the chat server, simply sends a message to a remote host. If the user at the remote host is in a Lisp Listener, the screen flashes and beeps, and the message appears in the Lisp Listener. If the remote user is in a Zmacs buffer, then the screen flashes and beeps, and the message appears in a minibuffer. The server does not provide error handling or other frills. To send a message from one host to another, a user at host Sahara enters the following into a Lisp Listener:

```
(generic-chat "foreign-legion" "Bring water!")
```

At the remote host "foreign-legion", the screen flashes and beeps, and the following message appears on the screen:

```
Chat from Sahara: Bring water!
```

The server then sends an acknowledgment message "Gossip sent!" to the user side to indicate that the message from the client was received.

Chat — Client Side

6.5.1.1 A user at the client side of the connection can access the chat server at a remote host with the following function:

```
(defun generic-chat (host message)
  (with-open-stream
    (stream (net:open-connection-on-medium
              (net:parse-host host)
              :byte-stream
              "generic-chat"
              :stream-type :ascii-translating-character-stream))
    (write-line message stream)
    (send stream :force-output)
    (format t "~&~A~%" (read-line stream)) nil))
```

The function net:open-connection-on-medium is used to create a generic bidirectional byte-stream connection stream to a remote host host. The stream created is of the :ascii-translating-character-stream type.

Forcing Output

6.5.1.2 Since stream connections normally buffer their output until enough data is available to fill an entire packet (in the case of Chaosnet) or segment (in the case of IP), it is possible for a **write-line** operation to be successful without it being sent over the network. To make sure that the packet is sent, it is sometimes necessary to send the stream a **:force-output** message. This action forwards a small amount of data (such as what you are writing in this example) to the other side of the stream connection. Sending the **:force-output** message is especially important if the client side needs to read data back from the server, as in the present case. As soon as the message appears on stream, the server's process can read it.

After the message is sent to the remote host, the chat function waits to read data from the stream connection. This reading is accomplished with the read-line function. The data sent from the server is read from the stream and printed on the screen by the format function.

Now the chat function is exited, and with-open-stream automatically closes the stream.

Chat — Server Side

6.5.1.3 The chat server function, at the remote side of the connection, is as follows:

```
(defun generic-chat-server ()
  (with-open-stream
    (stream (net:listen-for-connection-on-medium
              :byte-stream
              "generic-chat"
              :stream-type :ascii-translating-character-stream))
    (let ((message (read-line stream))
          (host (send stream :foreign-host)))
      (tv:notify nil "Chat from ~A: ~A" host message)
      (write-line "Gossip sent!" stream))))
```

Chat — Getting on the Net

6.5.1.4 After you have defined the client and server side functions, you must make three additional function calls to get the server up and running, and to access it.

The following function call defines the generic service :chat at the client's side of the connection:

```
(net:define-service :chat (host message) (host message))
```

The first argument to net:define-service is a keyword that identifies the generic service. In this case, it is :chat. The next two arguments are lists of arguments passed to the client-side function. The first list is the set of arguments required by the generic function. The second is a list of arguments expected by the particular lower-level implementation of the service.

NOTE: To avoid compiler errors, all function calls to **net:define-logical-contact-name** and **net:add-server-for-medium** must be evaluated, and therefore entered, into your .LISP file (or Zmacs buffer) in the order that they are presented in the following discussion.

---

net:define-logical-contact-name This function is called to define the server's logical contact name. The arguments include the logical contact name and a list of network name translations. The necessary form for the chat server is as follows, assuming that both the Chaosnet and TCP/IP networks are implemented:

```
(net:define-logical-contact-name
  "generic-chat"
  '((:chaos "Chat")
    (:tcp 250)))
```

net:add-server-for-medium This function adds a server for the specified medium.

```
(net:add-server-for-medium
  :byte-stream
  "generic-chat"
  '(process-run-function
     "generic-chat"
     'generic-chat-server))
```

A Witticism Server

6.5.2 The next example is slightly more complicated than the first. The means by which you access a particular server can vary from simple to extremely complex. The chat server simply sends a message to a remote host and closes the connection. The following server, a witticism server, is slightly more complex. Here, you open a connection with the server, which returns a witticism back to you. As in the case of the chat server, the following example takes a minimalist approach to networking. For simplicity, no error handling or other *frills* complicate the code. This server again illustrates the use of a generic byte stream. If your network had a witticism server called GROUCHO, you could invoke GROUCHO's witticism services by entering the following form into a Lisp Listener:

```
(generic-witticism "GROUCHO")
```

The definition of this function is as follows:

```
(defun generic-witticism (host)
  "Generically return a witticism."
  (with-open-stream
    (stream (net:open-connection-on-medium
              (net:parse-host host)
              :byte-stream
              :stream-type :ascii-translating-character-stream))
    (let ((notion (read-line stream)))
      (format t "~&~s" notion) nil)))
```

First, a stream connection is opened using net:open-connection-on-medium. The witticism server is listening for a request for connection on the foreign host identified by the net:parse-host function. The medium type is :byte-stream, and the stream type, as always, is :ascii-translating-character-stream.

After opening the connection, the user function invokes the read-line function to get the incoming witticism sent by the server on the stream connection. Note that the client-side function is now in a wait state. Nothing more is done until the read-line function returns a value.

Note that at the level of a generic byte stream this client-side function does not send anything to the server. It simply opens the connection and waits for a response.

Server Side

6.5.3 The witticism server listens for "generic-witticism" (the logical contact name) and returns the requested generic witticism back to the client side of the session. The definition of the generic-witticism-server function is as follows:

```
(defun generic-witticism-server ()
  (with-open-stream
    (stream (net:listen-for-connection-on-medium
              :byte-stream
              "generic-witticism"
              :stream-type :ascii-translating-character-stream))
    (let ((message (wit)))
      (write-line message stream))))
```

The witticism-server function listens for a connection on the generic :byte-stream medium. It listens for the "generic-witticism" logical contact name. The stream type is :ascii-translating-character-stream.

The connection is established, and the generic-witticism-server binds the message variable to a string returned by running an auxiliary function, wit, that finds and returns the appropriate generic witticism. The following function is a simple version of wit that works for demonstration purposes.

```
(defun wit ()
   (nth (random (length *witticisms*)) *witticisms*))
```

The function randomly selects a witticism-string from the list identified by the *witticisms* variable. An example of how to establish the *witticisms* list follows:

```
(setf *witticisms* '("Time is money."
                     "Money is the root of all evil."
                     "Time is the root of all evil."
                     "Without evil there can be no good."
                     "Without money you cannot have a good time."
                     "Without time you cannot spend money.")
```

Finally, the server writes the message to stream. Remember that the client-side function generic-witticism is already waiting to read the witticism from the stream at this point.

Finally, the with-open-stream macro takes care of closing the connection prior to exiting.

Getting the Server Up and Running

6.5.4 Once the client-side function and the server function have been defined, three further tasks remain to be completed. These are performed at the *client's* side. Neither the generic service nor the implementation client-side functions take any arguments. Remember that these functions must be called in the order in which they are discussed.

Logical Contact Name

6.5.4.1 A logical contact name must be defined for the server. To do this, you use the function **net:define-logical-contact-name** at the server side. This function is called with the logical contact name and a list of translations as arguments. In this case, the logical contact name is "generic-witticism"; the list of translations given here indicates that the logical contact name "generic-witticism" translates into the Chaosnet contact name "witticism" and into TCP port 250:

```
(net:define-logical-contact-name
  "generic-witticism"
  '((:chaos "witticism")
    (:tcp 250)))
```

Add Server for Medium

6.5.4.2 To add the server to the medium, you must use the GNI function **net:add-server-for-medium**. This function is called as follows for the generic witticism server. The medium is :byte-stream, the logical contact name is "generic-witticism", and the server function is generic-witticism-server:

```
(net:add-server-for-medium
  :byte-stream
  "generic-witticism"
  '(process-run-function "wit" 'generic-witticism-server))
```

Defining the Service

6.5.4.3 The last function, which must be called on the client side, is **net:define-service**. The appropriate call for the witticism server is as follows:

```
(net:define-service :witticism () ()
  "This service returns a witticism.")
```

Application Protocols

6.6 The servers discussed to this point involve simple interactions between the client side and the server side in which the client requests information or a service from the server, and the server performs that service or returns the needed information, terminating the information exchange immediately. When more complex interactions between the client side and the server side are required, it becomes necessary to synchronize interactions between the client and server.

To illustrate the problems posed in synchronizing complex interactions between the client and the server, a more complicated version of the previous spelling server is discussed. A hypothetical user interaction with this version of the spelling checker might proceed as follows.

The user enters a request for a spelling check in a Lisp Listener:

```
(check-spelling
  '(Impudent and prolix sesquipedalians intimidate more than impress))
```

The server then checks each form against the spelling list and returns a list of unrecognized forms to the user at the client side of the session with the following message:

```
These words are incorrect or unknown:
(PROLIX SESQUIPEDALIAN)
```

The user is then given an opportunity to add an unrecognized though correctly spelled word to the spelling list:

```
Add PROLIX to the spelling list? (Y or N)
```

After the user has responded to each prompt, the list of words to be added to the server's spelling list is sent to the server. The server then sends a message back to the client that terminates the session. The client-side function then prints a "Done" message to the user and returns **nil**.

The principal task here is to create the necessary client and server functions in such a way that each knows what to expect from the other at any given time.

Client Side

6.6.1 As always, the client side of the session invokes the server. The definition of the client server invocation function is as follows:

```
(defun generic-check-spelling (host text)
  (with-open-stream
    (stream (net:open-connection-on-medium
              (net:parse-host host)
              :byte-stream
              "generic-speller"
              :stream-type :ascii-translating-character-stream))
    (write text :stream stream)
    (send stream :force-output)
    (let ((wrong-words (read stream)))
      (cond ((equal wrong-words "1")
             (format t "~&No incorrect spellings in the text.~%")
             (no-new-additions-to-spelling-list stream))
            (t (format
                 t
                 "~&These words are incorrect or unknown:~&~s~%~%"
                 wrong-words)
               (loop
                 for word in wrong-words
                 when (and word
                           (y-or-n-p "Add ~S to the spelling list?"
                                     word))
                 collect word into new words
                 finally
                 (cond ((null new-words)
                        (no-new-additions-to-spelling-list stream))
                       (t (write new-words :stream stream)
                          (send stream :force-output)))))))
    (let ((done (read stream)))
      (format t "~&~S" done))) nil)
```

The client-side auxiliary communications function `no-new-additions-to-spelling-list` is defined as follows:

```
(defun no-new-additions-to-spelling-list (stream)
  (write "0" :stream stream)
  (send stream :force-output))
```

Server Side

6.6.2 The `generic-spelling-server` function is defined as follows:

```
(defun generic-spelling-server ()
  (with-open-stream
    (stream
      (net:listen-for-connection-on-medium
        :byte-stream
        "generic-speller"
        :stream-type :ascii-translating-character-stream))
    (let ((misspellings (get-misspellings (read stream))))
      (cond ((null misspellings)
             (write "1" :stream stream)
             (send stream :force-output))
            (t (write misspellings :stream stream)
               (send stream :force-output)))
      (let ((temp (read stream)))
        (cond ((equal temp "0")
               (send-done-message stream))
              (t (add-to-spelling-list temp))))
      (send-done-message stream))))
```

The auxiliary communications function send-done-message, used by the spelling server function, writes the string "done" to the stream and executes a :force-output. It is defined as follows:

```
(defun send-done-message (stream)
  (write "Done" :stream stream)
  (send stream :force-output))
```

For the sake of discussion, the auxiliary functions used by the spelling server are kept as simple as possible:

```
(defun add-to-spelling-list (text)
  (loop for word in text
        collect (symbol-name word) into string-list
        finally (setq *spelling-list*
                      (nconc string-list *spelling-list*))))
```

The function get-misspellings has the same definition as it did earlier, as does the global variable *spelling-list* which holds the spelling list:

```
(defvar *spelling-list*
        '("This" "is" "a" "sample" "word" "list"))

(defun get-misspellings (text)
  (loop for word in text
        when (not
               (member (symbol-name word)
                       *spelling-list*
                       :test
                       #'string-equal))
        collect word))
```

Contact Name

6.6.3 For the spelling server to function properly, you must define its logical contact name. The function **net:define-logical-contact-name** is used for this purpose:

```
(net:define-logical-contact-name
  "generic-server"
  '((:chaos "spelling")
  (:tcp 241)))
```

Adding the Server

6.6.4 To tell the system of the existence of the server, use the function **net:add-server-for-medium**:

```
(net:add-server-for-medium :byte-stream
                           "generic-spelling"
                           'generic-spelling-server)
```

Writes and Reads

6.6.5 The more complicated the interaction between client and server, the more carefully you have to be that server-side and client-side reads correspond to the correct server-side and client-side writes. Special care must be paid where a read at one side of a connection corresponds to two or more possible writes at the other side, as can occur, for example, inside a conditional expression. In order to understand this more clearly, you can step through the client-side function `generic-check-spelling` and the server-side function `generic-spelling-server` to check reads against writes.

First Write — Client Side

6.6.5.1 The first write takes place at the client side of the connection. The function `generic-check-spelling` sends the text or word list to the server to be checked. To make sure that the entire text is transmitted, a `:force-output` message is sent to `stream`.

First Read — Server Side

6.6.5.2 At the server side of the connection, after opening `stream`, the stream is read. The value returned by `read` is then passed to the function `get-misspellings`. The value of this list is bound to `misspellings`.

Second Write — Server Side

6.6.5.3 Now it is the server's turn to transmit. The first conditional clause offers two possibilities. Either it writes the arbitrarily chosen character `"1"` to `stream` if `get-misspellings` has not found any strange words and `misspellings` is bound to **nil**, or it sends the list of nonexistent or misspelled words back to the client-side function.

Second Read — Client Side

6.6.5.4 The client-side function binds the variable `wrong-words` either to `"1"` or to a list containing any unrecognized words, whichever the server sends.

Flag Waving

6.6.5.5 If the spelling server does not find any incorrect words, it must notify the client-side function of this fact. An arbitrary string is chosen to serve as a flag to the client side that no misspellings were found. In the first conditional clause in the function `generic-spelling-server`, the string `"1"` is sent to flag the client function that no list of `wrong-words` is forthcoming.

NOTE: At this point, you might think that you could simply write the `"There are no incorrect spellings."` message and return **nil**, immediately terminating the transaction, as in the following code:

```
(cond ((equal wrong-words "1")
       (format stream "~&There are no incorrect spellings.")
       nil)))
```

This form does not work, however, because the generic spelling server expects to read additional input from the client side (on the server side, see the `(let ((temp (read stream)))` form). The whole communication session simply hangs after printing the message. The server will listen forever to its stream, and the function **with-open-stream** will never close the connection.

CAUTION: Every read, whether on the client side or on the server side, must have a corresponding write at the other side of the connection.

Third Write—Client Side

6.6.5.6 As noted in the previous numbered paragraph, after binding wrong-words, the function at the client side can respond to the server in one of two ways. It *must* respond, however.

Even if there are no new words or misspellings to be considered as additions to *spelling-list*, the client side *must* return something to the server. If the client does not require further service of the server, it waves a flag at this write by sending a "0" back to the server with the no-new-additions-to-spelling-list function.

If the server finds unrecognized words in the input list, then the user at the client side decides which of the words to add to the spelling list that the server maintains.

Again, there are two possibilities. If the user decides not to add any new words to the lexicon, then the list bound to the variable new-words by the loop macro will be empty, as in the case of the previous option. The server is still listening to the stream, so the client *must* send something. The client side calls the function no-new-additions-to-spelling-list, as it did earlier, sending a "0" flag to the server.

The last possibility at the client side (the third write) is that the user has made a selection of words to be added to the spelling list. These words are put into a list by the loop macro, which then uses the write function to send them to the server. To be sure that everything is sent properly, a :force-output message is sent to the stream.

Third Read

6.6.5.7 The third read takes place at the server side of the connection. Whatever is read from the stream is bound to the variable temp. This variable is bound to either a list of words to be added to *spelling-list* or to "0".

Fourth Write

6.6.5.8 The value of the variable temp determines whether to add words to *spelling-list*. If temp is bound to "0", then "done" is transmitted back to the client by the fourth write (via the send-done-message function). If temp is bound to a list of words, then the list is appended to *spelling-list*, and a "done" message is sent back to the client.

Fourth Read

6.6.5.9 A communications session is always complete when the last read has been accomplished, whether by the client or the server. In the generic spelling program, this final read occurs on the client side of the connection. When the client receives the "Done" message, Done is printed on the client's screen and nil is returned.

At this point, the functions at both sides of the connection have terminated normally. Neither side has any outstanding reads, so the Common Lisp function with-open-stream takes care of closing the stream at both the client side and the server side.

Generic Access of Protocol-Specific Services

6.7 One of the principal benefits of the Generic Services Interface is the ability to use a predefined protocol-specific server without the need to modify the server code. For instance, an existing Chaosnet, TCP/IP or DECnet specific server can be used without modification. The only change that must be made is to the way that the client-side accesses the provided service. The following paragraphs discuss how this is arranged.

Server Function

6.7.1 The following example has two servers. The first of these is a Chaosnet protocol-specific server and the second is a TCP-specific server function. They are modified versions of the the chat server already discussed within the context of the generic network interface. Refer to the detailed discussion of this example in Section 5, titled Chaosnet Applications Programming and Networking; although the inner workings of the server function should be familiar in light of the preceding discussion of GNI. (See also the *Explorer TCP/IP User's Guide* for details on the TCP/IP programmatic interface.)

The following code represents the Chaosnet version of the chat server:

```
(defun chaos-chat-server ()
  (let ((conn (chaos:listen "Chat")))
    (chaos:accept conn)
    (with-open-stream
      (stream (chaos:make-stream conn :ascii-translation t))
      (let ((message (read-line stream))
            (host (sys:get-host-from-address
                    (chaos:foreign-address conn) :chaos)))
            (w:notify nil "Chat from ~A: ~A host message)
            (chaos:answer-string conn "Gossip sent!")))))
```

The following code represents the TCP version:

```
(defun tcp-chat-server ()
  (with-open-stream
    (stream
      (ip:open-stream nil
                      :local-port 241
                      :characters :ascii))
    (let ((message (read stream))
          (host (send stream :foreign-host)))
          (tv:notify nil "Chat from ~A: ~A host message)
          (write "Gossip sent!" :stream stream))))
```

To ensure that the chat server is listening for the contact name, you use the function **add-initialization**. (See the *Explorer Lisp Reference* manual for further details on initializations.) To initialize the chaos-chat-server, the function **add-initialization** is called as follows:

```
(add-initialization "chat"
                    '(chaos-chat-server)
                    nil
                    'chaos:server-alist)
```

To initialize the TCP-chat-server, use the following function call:

```
(add-initialization 241
                    '(process-run-function "TCP Chat" 'tcp-chat-server)
                    nil
                    'ip:*tcp-server-alist*)
```

Client Side

6.7.2 The server function of a service implementation is identical to any other protocol-specific server. However, the generic client-side functions used to access the servers require some attention.

Defining a Service Flavor

6.7.2.1 As discussed earlier, a service implementation is a flavor instance. The name of the generic implementation of chat is to be `:chat`. The `chat` flavor is defined as follows. Note that the name of the flavor and the name of the service implementation need not be identical, although they are in this case.

```
(defflavor chat
   ((net:name :chat))
   (net:service-implementation-mixin))
```

Service Implementation Mixin

6.7.2.2 When defining a service implementation flavor, you *must* include the mixin flavor **net:service-implementation-mixin**, as illustrated previously.

Client Side

6.7.2.3 The principal difference between the use of generic services and the use of a client-side function to invoke a generic byte-stream server or a protocol-specific (that is, a Chaosnet or IP) server is that the latter accesses the server via a client-side function call. A generic service is accessed by a method on the particular service implementation flavor. The `:send-message` method of the `chat` flavor is defined as follows.

```
(defmethod (chat :send-message) (host medium message)
  (unless (net:superior-medium-p medium :byte-stream)
    (ferror 'net:gni-service-error
            "Chat will not work on medium ~S."
            medium))
  (with-open-stream (stream
                      (net:open-connection-on-medium
                        host
                        medium
                        "new-chat"
                        :stream-type
                        :ascii-translating-character-stream))
    (write message :stream stream)
    (send stream :force-output)
    (read stream)
```

Note that this method uses the generic byte-stream medium to make the connection, even though the server with which the connection is made is Chaosnet-specific. As a result, the user can access the chat server in any of three ways:

- Use the generic **:byte-stream** medium:

```
(send "target-host" :chat :byte-stream "Using the byte stream medium.")
```

- Use the **:chaos-stream** medium:

```
(send "target-host" :chat :chaos-stream "Using chaos stream medium.")
```

- Use the **:tcp-stream** medium:

```
(send "target-host" :chat :tcp-stream "Using tcp stream medium.")
```

Compiling a Flavor

6.7.2.4 Use the **compile-flavor-methods** macro to ensure that the `:send-message` method is compiled appropriately:

```
(compile-flavor-methods chat)
```

Getting on the Network

6.7.3 Once the server function and the client-side service implementation flavor have been defined, several additional client side tasks remain.

Logical Contact Name

6.7.3.1 As is true with all generic services, the logical contact name must be defined. In the present case, you have two protocol translations, one for the the IP chat server, and the other for the Chaosnet chat server:

```
(net:define-logical-contact-name "new-chat"
                                 '((:chaos "Chat") (:ip 241)))
```

Define the Service

6.7.3.2 To define the service, you use the **net:define-service** macro. The service name is `:chat`; the generic service argument is the message sent to the foreign host:

```
(net:define-service :chat (message)
  (self net:medium message)
  "This service chats a foreign host.")
```

Service Implementation

6.7.3.3 Next, you must define the service implementation with the function **net:define-service-implementation**:

```
(net:define-service-implementation 'chat)
```

:services Attribute

6.7.4 You still cannot access the `chat` server as a generic service. To get to the `chat` server you must now enter the Namespace Editor to edit the host's **:services** attribute.

Add the following services to the **:services** attribute list:

```
(:chat :chaos-stream :chaos-chat)
```

```
(:chat :TCP-stream :TCP-chat)
```

You can now test the chat server by entering the following into a Lisp Listener:

```
(send sys:local-host :chat :send-message "Hello! This is a test.")
```

Errors

6.8 Several types of network errors can occur. First are the standard generic network errors that trap errors independently of the individual protocols implemented on the network. Additionally, each protocol has its own error monitor and trapping facilities. Thus, Chaosnet, TCP/IP, and so on, have their own error conditions. The following paragraphs discuss the standard network errors and the error conditions particular to the Chaosnet protocol. For specific information about error conditions in other network protocols, you should refer to the specific documentation for the networking option in question.

Standard Network Errors

6.8.1 The following definitions are for the standard network error condition/flavors.

net:network-error (error) Flavor

This is the base flavor for all network errors. All network errors use flavors built upon this one.

net:gni-medium-error Condition

This is used inside of any medium-related failure.

net:gni-service-error Condition

This is used inside of any service-related error.

Local Problems

6.8.2 The following Chaosnet-oriented error conditions can occur on a local host:

net:local-network-error (net:network-error error) Flavor

This flavor is used for problems that are entirely the result of activity on the local Explorer machine.

net:local-network-error (net:local-network-error net:network-error error) Condition

Some problem that is not described in the following conditions has occurred on the local network.

net:network-resources-exhausted (net:local-network-error net:network-error error) Condition

This condition is signaled when some local resource in the Network Control Programs is exhausted. There are probably too many Chaosnet connections and the connection table is full.

net:unknown-address (net:local-network-error net:network-error error) Condition

The *address* argument to **chaos:connect** or some similar function was not recognizable. The **:address** operation on the condition instance returns the address that was supplied.

Problems Involving the Actions of Other Machines

6.8.3 The following error conditions can occur as a result of the actions of remote hosts.

net:remote-network-error (net:network-error error) Flavor

This flavor is used for network problems that involve the actions—or lack of them—of other machines. This flavor is often useful for testing as a condition name.

The **:connection** and **:foreign-host** messages return the **chaos:conn** object and the host object for the foreign host.

Every instance of **net:remote-network-error** is the result of either **net:connection-error** or **net:bad-connection-state.**

net:connection-error (net:remote-network-error error) Condition

This condition name indicates failure to complete a connection.

net:bad-connection-state (net:remote-network-error error) Condition

This condition name indicates that an existing connection that was formerly valid has now become invalid. This error is not signaled until you try to use the connection.

net:no-server-up (net:connection-error net:remote-network-error net:network-error error) Condition

This condition indicates that no server was available.

net:host-not-responding-during-connection (net:connection-error net:remote-network-error error) Condition

This condition indicates that a host is not responding after it has been asked to make a connection.

net:host-stopped-responding (net:bad-connection-state net:host-not-responding net:remote-network-error error) Condition

This condition indicates that a host is not responding even though a connection to it already exists.

net:connection-refused (net:connection-error net:remote-network-error error) Condition

This condition indicates that a connection was refused.

If the CLS packet contained a reason for the refusal, the **:reason** operation on the condition instance returns that reason. Otherwise, the operation returns **nil.**

net:connection-closed (net:bad-connection-state net:remote-network-error error) Condition

This condition indicates that you have tried to transmit on a connection that has been closed by the other host.

If the CLS packet contained a reason for the refusal, the **:reason** operation on the condition instance returns that reason. Otherwise, the operation returns **nil.**

net:connection-lost (net:bad-connection-state net:remote-network-error error) Condition

This condition indicates that you have tried to use a connection on which an LOS packet was received.

If the CLS packet contained a reason for the refusal, the **:reason** operation on the condition instance returns that reason. Otherwise, the operation returns **nil**.

net:connection-no-more-data (net:bad-connection-state net:remote-network-error error) Condition

This condition indicates two things: first, you have tried to read from a connection that has been closed by the other host; second, there are no more packets left to be read. Note that it is not an error to read from a connection that has been closed if you have not yet read all the packets that arrived, including the CLS packet.

If the CLS packet contained a reason for the refusal, the **:reason** operation on the condition instance returns that reason. Otherwise, the operation returns **nil**.

7 NETWORK STATUS AND TROUBLESHOOTING

Introduction

7.1 This section discusses how to check the status of the network, and provides a brief description of the network debugging tools. Specific topics in this section include the following:

- Networking functions
- Networking-applicable portions of the Peek window
  - Network
  - File Status
  - Servers
- Networking menu interface
  - Network operations menu
  - Displays menu
  - Diagnostics menu
  - Controller configuration menu

Networking Functions

7.2 Several functions are available for checking network status and resetting the network. The utilities in Peek and the networking menu interface also employ the following functions:

net:host-status &rest *hosts* Function

The **net:host-status** function checks on the status of hosts. If the *hosts* argument is **nil**, **net:host-status** polls all hosts known by the locally-cached version of the network namespace. Otherwise, the function polls the specified *hosts*. A status message is returned for each medium supported by each host. (For related information, see the description of the **net:*poll-each-status-p*** variable which follows the **net:host-status** function definition.)

Following is an example returned by the **net:host-status** function:

```
ADDR        HOST                STATUS

1462        "Brigham-Young"     is responding on medium CHAOS.
1107337720  "Brigham-Young"     is responding on medium IP.
```

The following list explains the meaning of the headers:

ADDR The address of the listed host.
HOST The name of the host.
STATUS The status (responding or not responding) of the host, and the medium in which the particular protocol has been implemented.

net:*poll-each-status-p* Variable

The **net:*poll-each-status-p*** variable, if **nil**, causes the **net:host-status** function to return only the status of the most desirable protocol of each host. With less information to check, the **net:host-status** function operates more quickly. If **net:*poll-each-status-p*** is non-**nil**, (t is the default), the **net:host-status** function operates as previously described, returning status for each protocol on the target host.

When proper network transmission and reception breaks down, the first corrective action you should take is to reset the network. To do so, use the following function:

net:reset &optional *enable-p* Function

The **net:reset** function resets the network for *all* the protocols currently loaded. If the value of *enable-p* is **nil** (the default), the function turns off the network for this host. If *enable-p* is non-**nil**, the network is enabled and turned on.

The **net:reset** function calls the various reset functions for all the protocols that are loaded.

The following descriptions describe some of the reset functions available for Explorer-supported networking protocols.

chaos:reset &optional *enable-p* Function

The **chaos:reset** function resets the network for the Chaosnet protocol only. If the value of *enable-p* is **nil** (the default), the function turns off the network for this host. If *enable-p* is non-**nil**, the network is enabled and turned on.

chaos:enable Function

The **chaos:enable** function enables the network for the Chaosnet protocol only. This function is called by **chaos:reset**, if that function's *enable-p* argument is non-**nil**.

NOTE: The following functions may or may *not* be applicable to your Explorer's networking environment, depending on the protocols available in your environment.

---

ip:reset &optional (*enable-p* nil) (*debug-p* nil) Function

The **ip:reset** function resets the network for the Explorer TCP/IP family of protocols (IP, ICMP, UDP, TFTP, FTP, and TCP). Again, if the value of *enable-p* is **nil** (the default), the function turns off the network for this host. If *enable-p* is non-**nil**, the network is enabled and turned on.

The value you supply for the *debug-p* argument determines which of the three levels of debugging output is displayed when resetting the network. The default value of **nil** allows no debugging, **t** allows moderate debugging, and **:verbose** allows maximum debugging. The debugging output interfaces with normal system use, and is not used under normal circumstances.

ip:enable Function

The **ip:enable** function enables the network for the Explorer TCP/IP family of protocols. This function is called by **ip:reset**, if that function's *enable-p* argument is non-**nil**.

---

dna:reset &optional *enable-p* Function

The **dna:reset** function resets the network for the Explorer DECnet family of protocols. Again, if the value of *enable-p* is **nil** (the default), the function turns off the network for this host. If *enable-p* is non-**nil**, the network is enabled and turned on.

dna:enable Function

The **dna:enable** function enables the network for the Explorer DECnet family of protocols. This function is called by **dna:reset**, if that function's *enable-p* argument is non-**nil**.

The easiest way to enter Peek is to press SYSTEM-P on the keyboard. Peek is a window-oriented utility that shows you a continual update of system status. This section also highlights some of the interesting system meters that the Network and Host Status items can show. For more information about how to use Peek, refer to the *Explorer Tools and Utilities* manual.

- Fingering hosts — Discusses the **finger** function and the TERM F key sequence, which display information about users logged in at various machines in your network. Also, discusses the **chaos:find-hosts-or-lispms-logged-in-as-user** function, which returns a list of hosts on which a user is logged in.

- Sending and printing notifications — Discusses the **chaos:shout** and the **chaos:notify-all-lms** functions, which send a message to all Lisp machines. Also, discusses the **chaos:notify** function, which sends a brief message to a specified host. Finally, discusses the **print-notifications** function, which reprints any notifications that have been received.

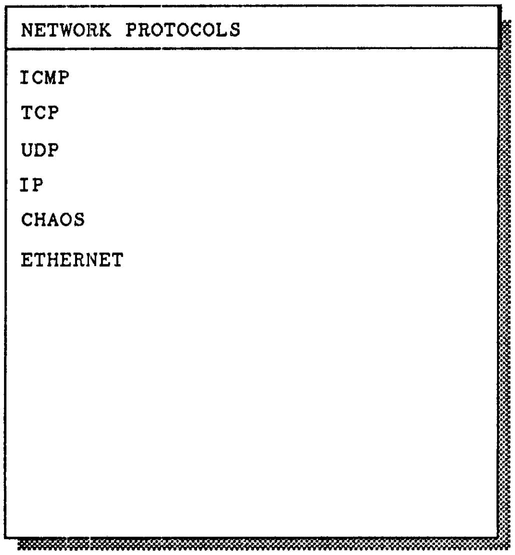
Peek Utility

7.3 The following paragraphs discuss the networking-applicable portions of the Peek utility. The Peek utility is made of modes; each mode of which has an associated window/interface.

You select a mode by pressing the key with the first letter of the item you want (N for Network, F for File Status, and so on). Alternately, you can select a Peek menu item with the mouse.

Network Mode

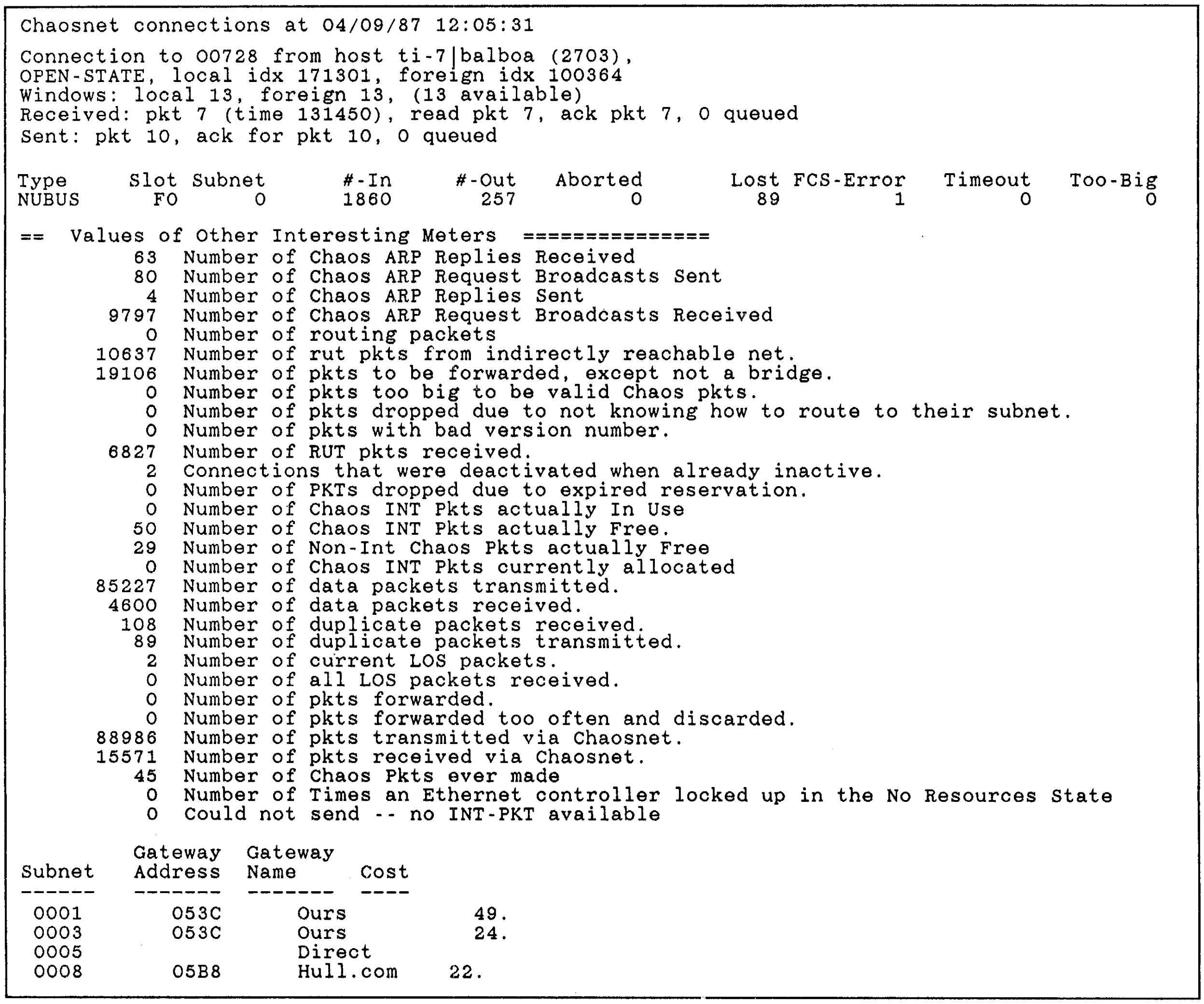
7.3.1 If you press N (for Network), a pop-up menu appears. The contents of that pop-up window depend on which networking protocols you have loaded in the current Explorer environment. If you have Chaosnet (a default) and TCP/IP loaded, the pop-up menu would appear as follows:



The ICMP, TCP, UDP, and IP selections are discussed in detail in the *Explorer TCP/IP User's Guide*. The following paragraphs discuss the CHAOS and ETHERNET selections.

CHAOS Selection **7.3.1.1** When you select the CHAOS option in the Peek network pop-up menu, a display appears, similar to the one in Figure 7-1, following:

Figure 7-1 Typical Contents of the Peek Network Mode Chaosnet Selection Screen

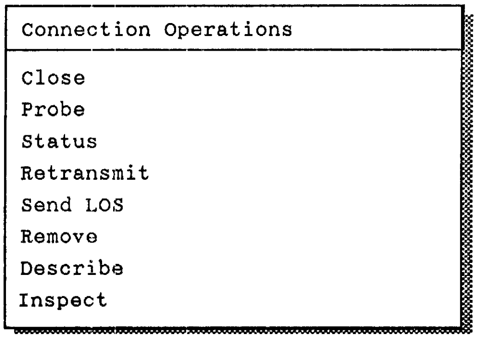


```
Chaosnet connections at 04/09/87 12:05:31

Connection to 00728 from host ti-7|balboa (2703),
OPEN-STATE, local idx 171301, foreign idx 100364
Windows: local 13, foreign 13, (13 available)
Received: pkt 7 (time 131450), read pkt 7, ack pkt 7, 0 queued
Sent: pkt 10, ack for pkt 10, 0 queued

Type     Slot  Subnet      #-In      #-Out   Aborted       Lost  FCS-Error   Timeout   Too-Big
NUBUS     FO      0        1860        257         0          89          1         0         0

==  Values of Other Interesting Meters ===============
         63  Number of Chaos ARP Replies Received
         80  Number of Chaos ARP Request Broadcasts Sent
          4  Number of Chaos ARP Replies Sent
       9797  Number of Chaos ARP Request Broadcasts Received
          0  Number of routing packets
      10637  Number of rut pkts from indirectly reachable net.
      19106  Number of pkts to be forwarded, except not a bridge.
          0  Number of pkts too big to be valid Chaos pkts.
          0  Number of pkts dropped due to not knowing how to route to their subnet.
          0  Number of pkts with bad version number.
       6827  Number of RUT pkts received.
          2  Connections that were deactivated when already inactive.
          0  Number of PKTs dropped due to expired reservation.
          0  Number of Chaos INT Pkts actually In Use
         50  Number of Chaos INT Pkts actually Free.
         29  Number of Non-Int Chaos Pkts actually Free
          0  Number of Chaos INT Pkts currently allocated
      85227  Number of data packets transmitted.
       4600  Number of data packets received.
        108  Number of duplicate packets received.
         89  Number of duplicate packets transmitted.
          2  Number of current LOS packets.
          0  Number of all LOS packets received.
          0  Number of pkts forwarded.
          0  Number of pkts forwarded too often and discarded.
      88986  Number of pkts transmitted via Chaosnet.
      15571  Number of pkts received via Chaosnet.
         45  Number of Chaos Pkts ever made
          0  Number of Times an Ethernet controller locked up in the No Resources State
          0  Could not send -- no INT-PKT available

         Gateway  Gateway
Subnet   Address  Name     Cost
------   -------  -------  ----
 0001       053C     Ours          49.
 0003       053C     Ours          24.
 0005                Direct
 0008       05B8     Hull.com    22.
```

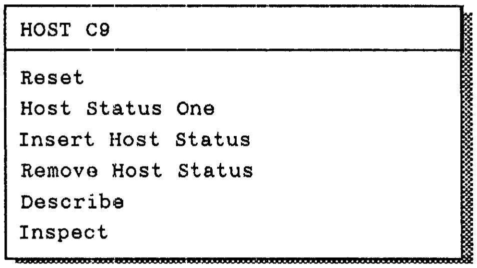
The following explains the meaning of the headers:

Chaosnet connections at — This gives you the current time. The time is expressed as a relative value that is incremented by 60 every second.

Connection to *server* at host *host* — If this item appears, it lists a Chaosnet connection to a specific host. If you click on the Connection to *server*, the following pop-up menu appears:



If you click on *host*, you get a pop-up menu similar to the one following:



Additional information is also displayed for the connection object. If you are interested in the concepts behind this additional information, see the paragraph titled Accessor Functions for a Conn in Section 5 of this manual, the section titled Chaosnet Applications Programming and Networking.

Below the connections listed in Figure 7-1, the following headers appear:

Type — The type of controller hardware used in this machine.

Slot — The number of the NuBus logical slot (F0 through F6).

Subnet — The number of the subnet for this controller.

#-In — The number of packets that were received since you last booted or reset the network.

#-Out — The number of packets that have been transmitted since you last booted or reset the network.

Aborted — The number of packets that were aborted during transmission.

Lost — The number of packets that the controller attempted to receive but could not. Remember, controlled packets are retransmitted.

FCS-Error — The number of packets that showed FCS errors. FCS stands for frame check sequence, which is the same as a cyclic redundancy check.

Timeout — The number of controller hardware timeouts.

Too-Big — The number of incoming packets that exceeded the legal size for packets under the Ethernet and Chaosnet protocols.

More information is given under the heading Values of Other Interesting Meters. This information is self-explanatory.

At the bottom of the screen, the following three headers describe the type of connection the host has to hosts on other subnets.

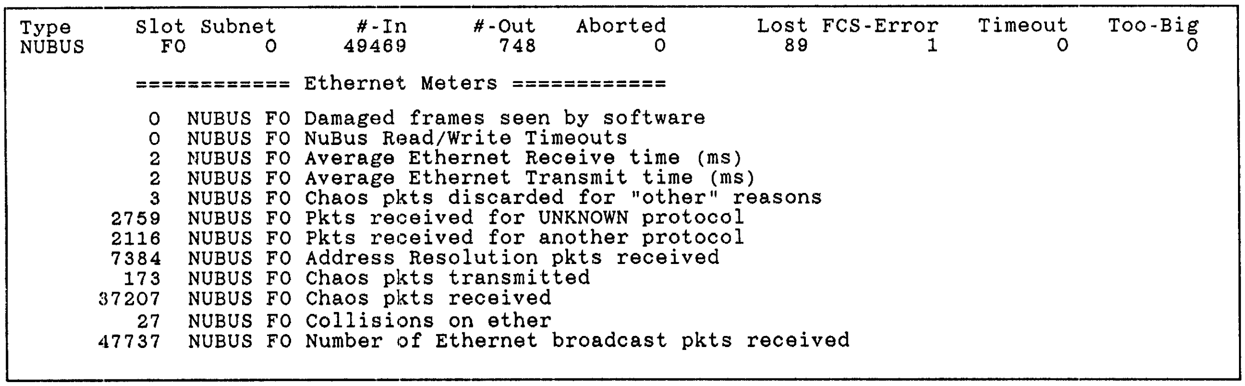
Subnet — This indicates the number of other subnets with which the host can communicate.

Gateway — This indicates whether the host has a direct connection over a particular subnet or whether it is first routed through another host.

Cost — This describes the difference in expected transmission time through relative numbers. For example, a cost of 24 indicates twice the transmission time as does a cost of 12.

ETHERNET Selection **7.3.1.2** When you select the ETHERNET option in the Peek network pop-up menu, a display appears, similar to the one in Figure 7-2, following:

Figure 7-2 Typical Contents of the Peek Network Mode Ethernet Selection Screen



```
Type      Slot Subnet      #-In     #-Out  Aborted       Lost FCS-Error   Timeout   Too-Big
NUBUS       FO      0     49469       748        0         89         1         0         0

         ============ Ethernet Meters ============

            0  NUBUS FO Damaged frames seen by software
            0  NUBUS FO NuBus Read/Write Timeouts
            2  NUBUS FO Average Ethernet Receive time (ms)
            2  NUBUS FO Average Ethernet Transmit time (ms)
            3  NUBUS FO Chaos pkts discarded for "other" reasons
         2759  NUBUS FO Pkts received for UNKNOWN protocol
         2116  NUBUS FO Pkts received for another protocol
         7384  NUBUS FO Address Resolution pkts received
          173  NUBUS FO Chaos pkts transmitted
        37207  NUBUS FO Chaos pkts received
           27  NUBUS FO Collisions on ether
        47737  NUBUS FO Number of Ethernet broadcast pkts received
```

At the top of Figure 7-2, the following headers appear:

Type — The type of controller hardware used in this machine.

Slot — The number of the NuBus logical slot (F0 through F6).

Subnet — The number of the subnet for this controller.

#-In — The number of packets that were received since you last booted or reset the network.

#-Out — The number of packets that have been transmitted since you last booted or reset the network.

Aborted — The number of packets that were aborted during transmission.

Lost — The number of packets that the controller attempted to receive but could not. Remember, controlled packets are retransmitted.

FCS-Error — The number of packets that showed FCS errors. FCS stands for frame check sequence, which is the same as a cyclic redundancy check.

Timeout — The number of controller hardware timeouts.

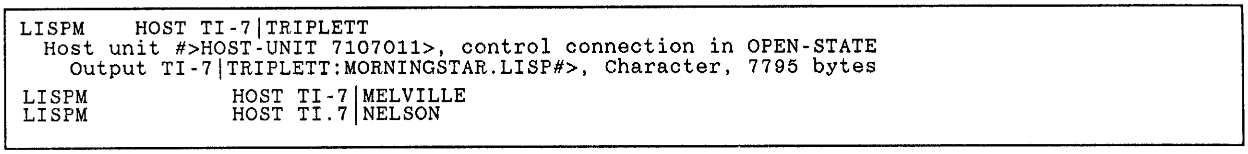
Too-Big — The number of incoming packets that exceeded the legal size for packets under the Ethernet and Chaosnet protocols.

More information is given under the heading Values of Ethernet Meters. This information is self-explanatory.

File Status Mode

7.3.2 When you select the File Status item from the command menu, Peek lists the file system host units as shown in Figure 7-3. Clicking on a host unit causes Peek to display a menu of useful operations on hosts. If you have any files open on a remote host when you select the File Status item, the status for each file is listed under its respective host, including such information as how far into the file you were when you called Peek. Clicking on any of these files causes Peek to display a menu of useful operations.

Figure 7-3 Typical Contents of the Peek File Status Mode Screen

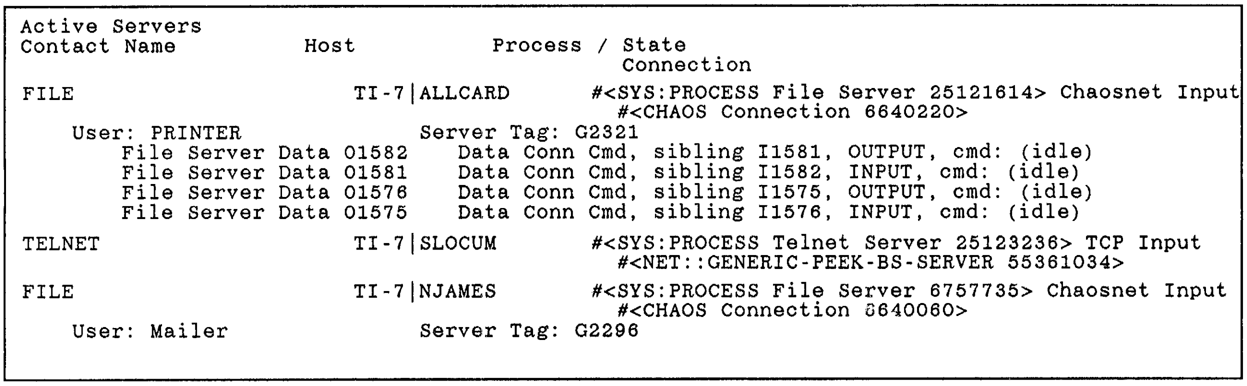


```
LISPM    HOST TI-7|TRIPLETT
  Host unit #>HOST-UNIT 7107011>, control connection in OPEN-STATE
    Output TI-7|TRIPLETT:MORNINGSTAR.LISP#>, Character, 7795 bytes
LISPM           HOST TI-7|MELVILLE
LISPM           HOST TI.7|NELSON
```

Servers Mode

7.3.3 When you select the Servers item from the command menu, the contact name, host, process, and connection of each active server are displayed as shown in Figure 7-4. The host, process, and connection are all mouse-sensitive items. Clicking on any of these items causes Peek to display a menu that lists the useful actions to be performed on these items.

Figure 7-4 Typical Contents of the Peek Servers Mode Screen



```
Active Servers
Contact Name          Host          Process / State
                                             Connection
FILE                      TI-7|ALLCARD      #<SYS:PROCESS File Server 25121614> Chaosnet Input
                                              #<CHAOS Connection 6640220>
    User: PRINTER             Server Tag: G2321
        File Server Data 01582   Data Conn Cmd, sibling I1581, OUTPUT, cmd: (idle)
        File Server Data 01581   Data Conn Cmd, sibling I1582, INPUT, cmd: (idle)
        File Server Data 01576   Data Conn Cmd, sibling I1575, OUTPUT, cmd: (idle)
        File Server Data 01575   Data Conn Cmd, sibling I1576, INPUT, cmd: (idle)
TELNET                    TI-7|SLOCUM       #<SYS:PROCESS Telnet Server 25123236> TCP Input
                                              #<NET::GENERIC-PEEK-BS-SERVER 55361034>
FILE                      TI-7|NJAMES       #<SYS:PROCESS File Server 6757735> Chaosnet Input
                                              #<CHAOS Connection 6640060>
    User: Mailer              Server Tag: G2296
```

Host Status Mode

7.3.4 You select the Host Status item by pressing H or clicking on Hostat at the bottom right corner of the Peek window with the mouse. When you select the Host Status mode, Peek calls the **net:host-status** function, and a type-out window appears with information similar to the following:

```
ADDR          HOST                   STATUS

1462          "Brigham-Young"        is responding on medium CHAOS.
1107337720   "Brigham-Young"        is responding on medium IP.
```

The following list explains the meaning of the headers:

ADDR -- The address of the listed host.

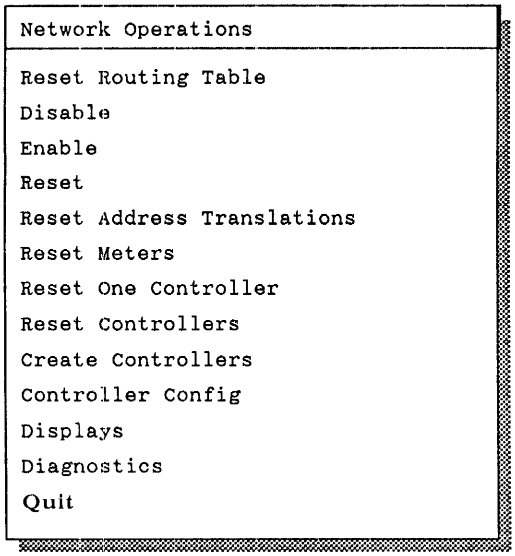
HOST — The name of the host.

STATUS — The status (responding or not responding) of the host, and the medium in which the particular protocol has been implemented.

Network Operations Menu

7.4 Note that the information about this and all subsequent menus described in this section is for debugging purposes only; it is not to be used for starting the network.

When you click on the Network item from the main System menu, the following menu appears:



The following list describes what happens when you click on each of the items in this menu:

Reset Routing Table — Resets the routing table to the proper state for the loaded configuration. The routing table then contains entries for only those subnets attached to the local host. This operation is useful only for debugging and is not needed for routine use.

Disable — Disables access to the Chaosnet system from or to the current host. The **chaos:disable** function is invoked.

`Enable` — Enables access to the Chaosnet system from or to the current host. The **chaos:enable** function is invoked.

`Reset` — Resets and enables access to the Chaosnet system from or to the current host. The **chaos:reset** function is invoked, with the **enable** parameter for that function set to **t**.

`Reset Address Translations` — Deletes all entries in the cache of Ethernet to logical address translations.

`Reset Meters` — Resets all the network meters to zero. The meters are displayed in the Peek utility and in submenus of the Network Display menu. The meters are counts of interesting items the Network has done or detected.

`Reset one Controller` — Resets one NuBus Ethernet controller board. If more than one controller is in use on this machine, then a pop-up menu lists the available controllers, and you must click on the one you want to reset. Resetting a controller does not disrupt any Chaosnet operations. This operation applies to both the software controller object and the NuBus Ethernet controller board.

`Reset Controllers` — Resets all the NuBus Ethernet controller boards. Resetting all the controllers does not disrupt any Chaosnet operations. This operation applies to both the software controller object and the NuBus Ethernet controller board.

`Create Controllers` — Creates new data objects for all the Ethernet controllers on the local host. Each NuBus Ethernet controller board on the system must have a related software object. You must first specify the configuration of these controllers by clicking on the Controller Config item. This operation is useful only for debugging and is not needed for routine use.

`Controller Config` — Specifies the configuration of the NuBus Ethernet controller boards in the chassis. A pop-up menu appears to help you with the configuration. The pop-up menu is explained in paragraph 7.7, titled Network Controller Configuration. This operation is useful only for debugging and is not needed for routine use.

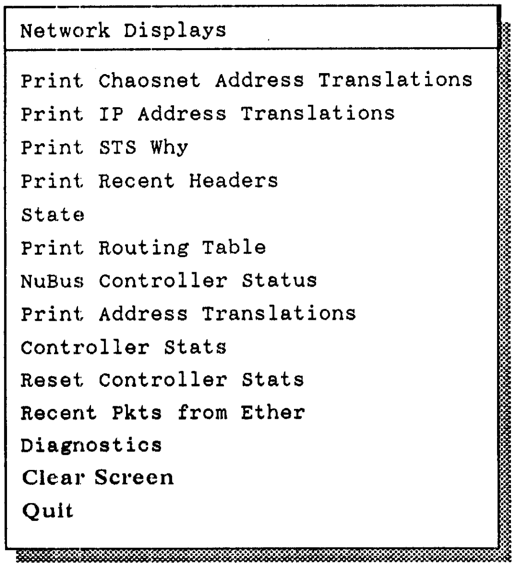
`Displays` — Enters the Network Displays menu. You will also get a typeout window.

`Diagnostics` — Selects the Network Diagnostics menu. You will also get a typeout window.

`Quit` — Returns to the previous window.

Network Displays Menu

7.5 When you click on the Displays item in the Network Operations menu or in the Network Diagnostics menu, the following menu appears, along with a typeout window:



Network Displays

Print Chaosnet Address Translations
Print IP Address Translations
Print STS Why
Print Recent Headers
State
Print Routing Table
NuBus Controller Status
Print Address Translations
Controller Stats
Reset Controller Stats
Recent Pkts from Ether
Diagnostics
Clear Screen
Quit

The following describes what happens when you click on each of the items in this menu:

`Print Chaosnet Address Translations` — Prints to the value of ***terminal-io*** all known translations from Chaosnet addresses to physical Ethernet addresses.

`Print IP Address Translations` — Prints to the value of ***terminal-io*** all known translations from IP addresses to physical Ethernet addresses.

`Print STS Why` — Prints the reasons why the last 64 status (STS) packets were sent. The following are possible reasons:

`Print Recent Headers` — Prints the headers from the last 50 Chaosnet packets received.

`State` — Prints out a summary about the state of the network.

`Print Routing Table` — Prints routing information for all known subnets. This information is taken from the routing table.

`NuBus Controller Status` — Displays the status of each NuBus Ethernet controller.

`Print Address Translations` — Prints to the value of ***terminal-io*** all known translations from IP *and* Chaosnet addresses to physical Ethernet addresses.

`Controller Stats` — Displays the network meters that contain statistics about the network controllers. The statistics include the number of received and transmitted packets, and the number of collisions.

`Reset Controller Stats` — Sets all the network meters (which were displayed by the previous command) to 0.

`Recent Pkts from Ether` — Prints the contents of the recent frames that were collected from Ethernet while the **ethernet:debug-ether-recv** variable was set to **t**.

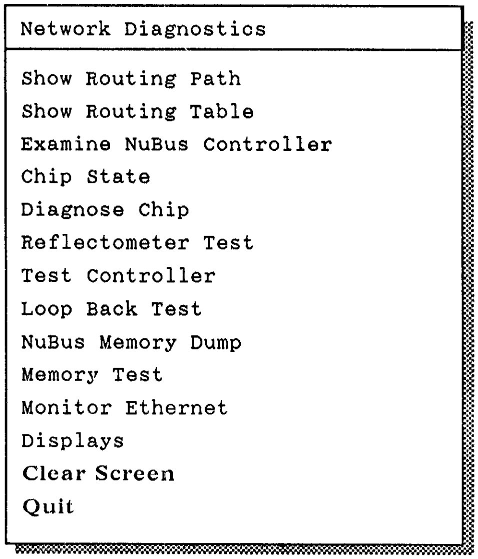
`Diagnostics` — Enters the Network Diagnostics menu.

`Clear Screen` — Clears the screen.

`Quit` — Exits this menu and returns to the Network Operations menu.

Network Diagnostics Menu

7.6 When you click on the Diagnostics item in the Network Operations menu or in the Network Displays menu, the following menu appears, along with a typeout window:



The following list describes what happens when you click on each of the items in this menu:

`Show Routing Path` — Computes the routing path between any two hosts on the same subnetwork. You are prompted for the names of the two hosts.

`Show Routing Table` — Prints the subnets that are reachable by a particular host. You are prompted for the name of the host.

NOTE: Several of the following items in the Network Diagnostics menu refer to structures described either in the *Explorer NuBus General Description* manual (part number 2243161-0001) or the *Intel® LAN Component User's Manual.*

`Examine NuBus Controller` — Interactively examines the Receive Area in a NuBus Ethernet controller. See the *Explorer NuBus Ethernet Controller General Description* manual.

`Chip State` — Dumps a portion of a NuBus Ethernet controller chip's data and gives the state of the chips. See the *Intel LAN Component User's Manual.*

`Diagnose Chip` — Sends the Diagnose command to a NuBus Ethernet controller chip and prints the results of command execution. See the *Intel LAN Component User's Manual.*

`Reflectometer Test` — Runs a Time Domain Reflectometer test on a NuBus Ethernet controller board and prints the results. See the *Intel LAN Component User's Manual.*

`Test Controller` — Runs the Memory, Diagnose Chip, Loopback, and Time Domain Reflectometer tests on a NuBus Ethernet controller board and prints the results. See the *Explorer NuBus Ethernet Controller General Description* manual and the *Intel LAN Component User's Manual.*

`Loop Back Test` — Runs a Loopback test on a NuBus Ethernet controller board. This test performs loopbacks at the controller chip, at the serial driver chip, and at the transceiver. See the *Explorer NuBus Ethernet Controller General Description* manual and the *Intel LAN Component User's Manual.*

`NuBus Memory Dump` — Dumps NuBus memory from a NuBus Ethernet controller board. See the *Explorer NuBus Ethernet Controller General Description* manual.

`Memory Test` — The network must be disabled to perform this test. This test exercises the memory of a NuBus Ethernet controller board. See the *Explorer NuBus Ethernet Controller General Description* manual.

`Monitor Ethernet` — Intercepts all Ethernet frames received on the local machine and displays their contents. Clicking on this item turns off the normal Ethernet receiver, so all Ethernet communications are off while this item is running. Once invoked, this state can be aborted by pressing the CTRL-ABORT key sequence.

`Displays` — Exits this menu and enters the Network Displays menu.

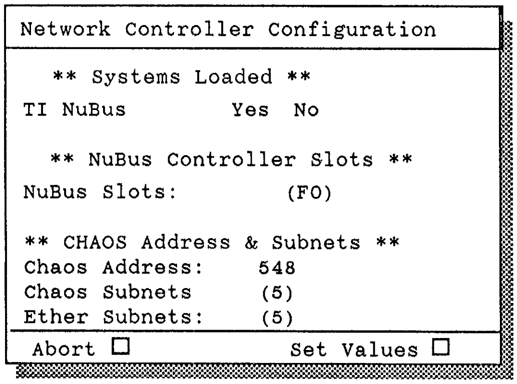
`Clear Screen` — Clears the screen.

`Quit` — Exits this menu and returns to the Network Operations menu.

Intel is a registered trademark of Intel Corporation.

Network Controller Configuration Menu

7.7 When you click on the Controller Config item in the Network Operations menu, the following menu appears:



```
Network Controller Configuration

  ** Systems Loaded **
TI NuBus          Yes  No

  ** NuBus Controller Slots **
NuBus Slots:          (F0)

** CHAOS Address & Subnets **
Chaos Address:     548
Chaos Subnets      (5)
Ether Subnets:     (5)

 Abort □                  Set Values □
```

Under the heading ** Systems Loaded ** is a prompt for designating which controller software is loaded on the host.

TI NuBus — Because Explorer systems use only the TI NuBus at present, answer Yes.

Under the heading ** NuBus Controller Slots ** is a prompt for designating which slot(s) the controller board is in.

NuBus Slots — Enter the numbers of the slots in a Lisp list. The range of slots in the chassis is from the leftmost in a Lisp list, #xF0, through the rightmost, #xF6. Enter these numbers in ascending order.

NOTE: When you enter lists in response to the NuBus Slots, Chaos Subnets, and Ether Subnets prompts, be sure that the ordering of each list corresponds logically with the ordering of the other lists.

Under the heading ** CHAOS Address & Subnets ** is a prompt for supplying information about the Chaosnet addresses and subnets associated with the controller.

Chaos Address — Enters the Chaosnet address of one of the NuBus Ethernet controller boards. You have one controller board for each physical subnet on which the host resides. It does not matter which controller board you choose for its Chaosnet address because that address will be parsed for its least-significant eight bits, which identify the host ID. (The host ID is the same for all the controller boards on a host.)

Chaos Subnets — Enters a list that contains the Chaos ID number for each subnet that is associated with a NuBus Ethernet controller board. For example, if subnets 12, 15, and 17 are each associated with the boards in this host, you would enter the following list: (12 15 17). Each element in the list is used to calculate the 16-bit Chaosnet address for its associated controller board. (A subnet number becomes the most-significant eight bits of the address; the host ID from the Chaos Address item becomes the most-significant eight bits of the address.)

Ethernet Subnets — Enter a list of all the Chaosnet subnet numbers for which this host has an Ethernet controller. This item currently does nothing. It is held over the days when Ethernet and Chaosnet used different hardware.

EXTERNAL DATA REPRESENTATION

A

Introduction

A.1 Sun Microsystems™ created the External Data Representation (XDR) protocol to allow two dissimilar machines to exchange operands over a network despite differences in byte ordering, word length, floating-point representation, and so on.

The XDR Technique

A.2 XDR defines a standard byte representation for certain primitive data types as they would appear on the network between two machines. The sending machine converts, or *filters*, its native data formats into the XDR standard representation and then outputs the filtered versions to the network. The receiving machine inputs the standard representations from the network and filters them into its own native representation.

Implied in this scheme are the following requirements:

- The sender and receiver must agree exactly on the order and XDR data type of the operands transferred.

- The XDR protocol must offer a sufficient selection of primitive data types to construct the higher level data structures that the two machines might need to exchange.

The Explorer Implementation

A.3 Although XDR was created to aid in networking, the XDR protocol itself does not involve networking. Instead, XDR is only a data representation standard, independent of where that data might reside.

On the Explorer, the XDR protocol is available as a stream mixin flavor, **rpc:xdr-stream**, for filtering Lisp operands to and from an Explorer stream. Explorer streams are usually byte-oriented; however, the XDR mixin causes the stream to be *operand*-oriented. That is, you input and output whole operands to an XDR stream — *never* individual bytes.

The **rpc:xdr-stream** flavor conveniently allows Explorer programmers to view network I/O as a serial operand stream. However, the network software itself always works in terms of buffers of operands. Therefore, it is necessary to serialize buffers received from the net into streams and to deserialize an Explorer stream into a buffer for .transfer to the net. The flavor **rpc:xdr-memory-stream** performs all the filtering functions of **rpc:xdr-stream**. In addition, it performs the serialization and deserialization needed for network transfer.

Sun Microsystems is a trademark of Sun Microsystems, Inc.

XDR Streams

A.4 The following paragraphs discuss the flavors and methods associated with XDR streams.

rpc:xdr-stream Flavor

The **rpc:xdr-stream** mixin converts an ordinary Explorer stream into an XDR operand stream. It is an abstract flavor with required methods **:byte-in** and **:byte-out**, which are synonymous with the standard **:tyi** and **:tyo** methods of streams. These alternate method names are needed by special underlying streams that must distinguish XDR-filtered I/O (carried by **:byte-in** and **:byte-out**) from direct stream I/O (carried by **:tyi** and **:tyo**).

:transfer-direction Method of **rpc:xdr-stream**
:set-transfer-direction *direction* Method of **rpc:xdr-stream**

XDR streams are inherently bidirectional. Therefore, the stream's transfer direction identifies the direction in which operands will be filtered.

- If the *direction* argument is **:encode**, then Lisp operands will be encoded into XDR operands and output to the stream.
- If the *direction* argument is **:decode**, then XDR operands will be input from the stream and decoded into Lisp operands.

For every XDR filter that encodes a Lisp variable into a certain XDR data type, there must be a matching filter to decode that XDR operand back into a Lisp variable. Instead of all filters being written in pairs, each filter is written to be bidirectional. A filter takes one data argument. For encoding, the value of that argument is read as any function's argument would be. For decoding, however, the decoded stream data is stored back into the contents of the filter's data argument.

Therefore, for decoding to work, the filter's data argument *must* have been passed as the locative of the variable that is to receive the decoded data. Encoding, on the other hand, does not need a locative argument; it will accept either the data value itself or a locative of that value.

The simplest way of obtaining the locative of a variable is to use the **locf** macro. To access the value of a variable passed as a locative, use the **contents** function. The last two forms increment the value of `var` by one:

```
(setf locf-of-var (locf var))

(setf var (1+ var))
(setf (contents locf-of-var) (1+ (contents locf-of-var)))
```

:xdr-enum *32-bit-signed* Method of **rpc:xdr-stream**
:xdr-integer *32-bit-signed* Method of **rpc:xdr-stream**
:xdr-unsigned *32-bit-unsigned* Method of **rpc:xdr-stream**
:xdr-hyper *64-bit-signed* Method of **rpc:xdr-stream**
:xdr-unsigned-hyper *64-bit-unsigned* Method of **rpc:xdr-stream**
:xdr-float *single-float* Method of **rpc:xdr-stream**
:xdr-double *double-float* Method of **rpc:xdr-stream**
:xdr-bool *boolean* Method of **rpc:xdr-stream**

These methods filter various simple data types to and from an XDR stream, depending upon the transfer direction. The **:xdr-enum** method is synonymous with the **:xdr-integer** method as far as Lisp is concerned. The two names are provided to conform to the names used in Sun's XDR documentation. None of these methods return any values of interest.

If the XDR stream is opened for encoding, then the argument to the filter may be either the Lisp operand to be encoded or the **locf** of that operand.

If the XDR stream is opened for decoding, then the argument *must* be the **locf** of the Lisp variable that is to receive the decoded XDR operand.

:xdr-void *ignore* Method of **rpc:xdr-stream**

This method is a dummy placeholder filter that ignores its operand, returns no value, and performs no I/O. It is used where the syntax of a function requires an XDR filter, but there is no operand to transfer.

For example, consider the problem of using **callrpc** to call a remote procedure that takes no arguments. The **:xdr-void** method would satisfy **callrpc**'s requirement for an *xdr-in* filter operand but would not produce other unwanted effects.

:xdr-string *string* Method of **rpc:xdr-stream**

This method is the appropriate filter to use for transferring variable-length strings in which no character translations are performed. It returns no values of interest.

If the XDR stream is opened for encoding, **:xdr-string** outputs the elements of *string* (which must be a vector or the **locf** of one) to the stream as 8-bit bytes. If *string* has a fill pointer, then it is observed.

Notice that during encoding, *string* actually need not be a string. It can be a vector of any element type that can be properly encoded as 8-bit unsigned bytes.

If the XDR stream is opened for decoding, **:xdr-string** inputs a variable-length string from the stream and stores it into the contents of *string* (which in this case *must* be a locative in order to unconditionally store back into it).

If the initial contents of *string* is *not* a vector, then **:xdr-string** allocates a suitable string and stores it into the contents of *string*. For example, if you bind a local variable to **nil** and then pass the **locf** of that variable as the *string* argument to **:xdr-string**, then that variable ends up with a newly created string containing the decoded operand.

If *string* initially contains a vector, then this method assumes that the existing vector is suitable for receiving the decoded string and reuses it. If that existing vector has a fill pointer, then the fill pointer is modified to reflect the length of the newly decoded string.

This reuse feature allows you to avoid consing new strings that need only to be examined, but not saved. You can allocate a permanent string buffer with a fill pointer and then pass the **locf** of that string as the *string* argument on each call. The fill pointer allows you to tell the length of each decoded string inside the permanent string buffer.

Unfortunately, this reuse feature may cause unexpected problems when you are expecting it to allocate its own strings. If **:xdr-string** is used inside a loop, then the first time through the loop it allocates a new string. The second time through the loop it sees an existing string and attempts to reuse it. However, if the second string is longer than the first, a subscript out-of-range error occurs.

Therefore, if you use **:xdr-string** in a loop, be sure to **setf** the destination variable to **nil** before each call to **:xdr-string** so that it will allocate a new string each time. Otherwise, **setf** the destination variable to a vector with a fill pointer large enough to handle all input.

:xdr-ascii-string *string* Method of **rpc:xdr-stream**

This method is similar to **:xdr-string** except that it assumes that *string* contains Explorer standard characters, whereas the XDR stream contains standard ASCII characters with the end of lines delimited by ASCII NEWLINE characters. That is, an Explorer #\RETURN character is encoded as an ASCII RETURN followed by an ASCII LINEFEED and visa-versa.

This method does for XDR string operands what the **sys:ascii-translating-mixin** does for Explorer streams.

:xdr-array *array elt-xdr-function* Method of **rpc:xdr-stream**

This filter is the generalized version of **:xdr-string**. While the elements filtered by **:xdr-string** are always 8-bit bytes, the elements of an XDR array (actually a vector) may be of any filterable XDR data type. Otherwise, the **:xdr-array** method offers all the options and features of **:xdr-string**.

The *elt-xdr-function* argument is used to filter each element of *array*.

- If *elt-xdr-function* is a keyword, then it is assumed to be the name of a method of **rpc:xdr-stream**. It will be sent as a message to the stream with an element from *array* as its single argument. For example:

  ```
  (send self elt-xdr-function (aref array i))
  ```

- If *elt-xdr-function* is *not* a keyword, then it is assumed to be a functional object. It will be **funcalled** with the stream and an element from *array* as its two arguments. For example:

  ```
  (funcall elt-xdr-function self (aref array i))
  ```

For example, to transfer a vector of integers, you would use **:xdr-integer** as *elt-xdr-function*.

:xdr-unsigned-vector *vector start end* Method of **rpc:xdr-stream**

This method is a specialization of **:xdr-array** with an implied *elt-xdr-function* argument of **:xdr-unsigned**. Instead of transferring the entire *vector* operand, this method transfers only the vector elements starting with the index *start* and ending one element before *end* (that is, *start* and *end* have the conventional meaning). Otherwise, the **:xdr-unsigned-vector** method offers all the options and features of **:xdr-array**.

Many of the higher-level data structures used in remote procedure calling include substructures that can be viewed as vectors of unsigned integers. This method is more efficient at filtering those substructures than an equivalent series of individual calls to **:xdr-unsigned**. The *start* and *end* indexes make it convenient to encode out of or decode into the middle of a larger data structure.

:xdr-opaque *string* Method of **rpc:xdr-stream**

This method is a specialization of **:xdr-string** for the case of fixed length strings whose length is known to both the sender and receiver. The opaque XDR data type is intended to be a *bag of bytes* that can be passed freely among different machines. However, the XDR opaque data type has meaning only to the machine that originally created it.

If the stream is opened for encoding, the *string* (a vector or a **locf** of a vector) is output to the stream as 8-bit bytes without any character translation. If *string* has a fill pointer, it is observed.

If the stream is opened for decoding, then *string* must be the **locf** of a preallocated string of the correct length. It is the length of this preallocated string that tells this method how many bytes to decode from the stream. If *string* has a fill pointer, it is observed.

:xdr-union *union discriminator discriminator-alist* &optional *default-discriminator* Method of **rpc:xdr-stream**

This method is defined to filter an operand called the *union*, which will be one of several possible types. The specific type is identified by the *discriminator*. The possible choices of operand type are in the *discriminator-alist* argument.

The *discriminator* argument is an enumeration (a signed integer).

The *discriminator-alist* argument is an association list of discriminator values dotted with *elt-xdr-functions* (see the preceding discussion of **:xdr-array**).

The *default-discriminator* argument, if present, is an *elt-xdr-function* to be used if *discriminator* is not present in the *discriminator-alist*.

If the XDR stream is open for encoding, the *discriminator*, a Lisp integer, is encoded as an XDR enumeration to tell the receiving end what type of operand follows. Next *discriminator* is looked up in the *discriminator-alist* to find its associated *elt-xdr-function*. If *discriminator* is not found, the value of *default-discriminator* is used as the *elt-xdr-function* instead; otherwise, an **rpc:unknown-union-discriminator** error is signaled.

Once the *elt-xdr-function* is known, then encoding proceeds as though the *elt-xdr-function* filter had been called with *union* (a value or a locative of a value) as its single argument.

If the XDR stream is open for decoding, a discriminator enumeration is decoded from the stream to determine the type of operand that follows. That discriminator is looked up on the *discriminator-alist* to find its associated *elt-xdr-function*. If the discriminator is not found, then the value of *default-discriminator* is used as the *elt-xdr-function* instead. If no default discriminator is found, the **rpc:unknown-union-discriminator** error is signaled.

Once the *elt-xdr-function* is known, then decoding proceeds as though the *elt-xdr-function* filter had been called with *union* (which, in this case, *must* be a locative) as its single argument.

Additional XDR Forms

A.5 The following paragraphs discuss additional XDR functions and macros.

value-of *object* Function

If *object* is a locative, **value-of** returns its contents. Otherwise, **value-of** returns *object* itself.

default-vector-and-resolve *variable length element-type* Macro

If the contents of the *variable* argument (a locative) is not a vector, then **default-vector-and-resolve** creates a new vector of element type *element-type* and length *length*. The macro then **setf**s the contents of the *variable* argument to this vector. Finally, if *variable* is a locative, **default-vector-and-resolve setf**s it to its own contents.

This macro returns no value of interest. It is used only for its side effects.

default-and-resolve *argument type-spec constructor-name* &rest *constructor-args* Macro

If *argument* is a locative whose contents are not of the type identified by the *type-spec* argument, then **default-and-resolve** first **funcall**s the *constructor-name* specifying *constructor-args* as its arguments. It then sets the contents of *argument* to the value returned from that **funcall**. That is, in the caller's original variable, **default-and-resolve** actually creates a new data structure of the proper type to contain the value decoded from the network.

In all cases, **default-and-resolve** replaces a locative with its contents.

This macro returns no value of interest. It is used only for its side effects.

resolve-locative *variable* Macro

If *variable* is a locative, then **resolve-locative setf**s it to its own contents. Otherwise, this macro takes no action.

round-to-quad *integer* Function

This function takes the *integer* argument and returns the value of *integer* rounded up to the next multiple of 4.

xdr-io *stream filter data* Macro

This macro performs I/O on the XDR stream identified by the *stream* argument, taking into account whether the *filter* argument is a keyword or a **funcall**able object.

If *filter* is a keyword, then it is assumed to be an operation on *stream* which takes *data* as its only argument. If *filter* is a function, then it is **funcall**ed with *stream* and *data* as its two arguments.

If the **rpc:transfer-direction** instance variable in *stream* is **:encode**, then the *filter* argument performs a transfer from *data* to *stream*. If **rpc:transfer-direction** is **:decode**, it performs the transfer from *stream* to *data*.

XDR Examples

A.6 For the first example, assume that you have a small data structure composed of two integers and a float:

```
(defstruct fool
  (a 0   :type integer)
  (b 0   :type integer)
  (c 0.0 :type single-float))
```

An XDR filter function for the fool structure would appear as follows:

```
(defun xdr-fool (stream  fool-obj)
  (send stream :xdr-integer (locf (fool-a fool-obj)))
  (send stream :xdr-integer (locf (fool-b fool-obj)))
  (send stream :xdr-float   (locf (fool-c fool-obj))))
```

NOTE: Do not enter the above example; it will not work. Later in this section, the example will be modified to work properly.

Since the **:xdr-integer** and **:xdr-float** methods are designed to be bidirectional, and since the **xdr-fool** function passes the operands to these methods as locatives, then **xdr-fool** is bidirectional also.

In this example, the data argument to the `xdr-fool` function (`fool-obj`) is passed as an ordinary variable—not a locative. This choice seems to contradict previous statements that the data argument to an XDR filter must always be a locative. Actually, the rule is this:

- If the filter function needs to modify the data argument itself back inside the caller of the filter, then that argument *must* be passed as a locative.
- If the filter needs to modify only the elements inside the data argument (for example, slots in a structure argument), then the argument itself does not need to be a locative.

In this example, then, the argument is a `fool` structure. The filter needs to modify slots in this structure, so it uses the `locf` of those slots. There is no need for the structure itself to be a locative. By way of contrast, if the argument were a nonstructured object such as an integer or a float, then that argument would have to be a locative.

Of course, saying that the arguments are sometimes locatives and sometimes not creates its own problems. The accessor function for slot A in the previous example is written as follows:

```
(fool-a fool-obj)
```

This form requires that the argument `fool-obj` be an ordinary variable. If, however, the argument were a locative of the `fool` structure, then the accessor would have to be written as follows:

```
(fool-a (contents fool-obj))
```

In this form, the `contents` function resolves the locative passed in `fool-obj` into the actual `fool` structure that the `fool-a` accessor needs.

This `xdr-fool` filter function would be more robust if it could accept either a `fool` structure or the locative of such a structure with equal grace. To accomplish this generalization, you can use either the **rpc:value-of** macro or the **rpc:resolve-locative** macro as shown in the examples that follow.

```
(defun xdr-fool (stream fool-obj)
  (send stream :xdr-integer (locf (fool-a (rpc:value-of fool-obj))))
  (send stream :xdr-integer (locf (fool-b (rpc:value-of fool-obj))))
  (send stream :xdr-integer (locf (fool-c (rpc:value-of fool-obj))))
)

(defun xdr-fool (stream fool-obj)
  (rpc:resolve-locative fool-obj)
  (send stream :xdr-integer (locf (fool-a fool-obj)))
  (send stream :xdr-integer (locf (fool-b fool-obj)))
  (send stream :xdr-integer (locf (fool-c fool-obj)))
)
```

Either use a `rpc:value-of` macro around every reference to an argument that may or may not have been passed as a locative, or use `rpc:resolve-locative` once at the beginning.

If `fool-obj` is a locative of a `fool` structure, then `rpc:value-of` resolves each usage of `fool-obj` while `rpc:resolve-locative` sets `fool-obj` to that structure. In the previous example, `rpc:value-of` would expand into something similar to the following:

```
(if (locativep fool-obj)
    (contents fool-obj)
   fool-obj)
```

The `rpc:resolve-locative` would expand into something similar to the following:

```
(when (locativep fool-obj)
  (setf fool-obj (contents fool-obj)))
```

Using rpc:resolve-locative, later accesses to fool-obj from inside xdr-fool will see just the fool structure. The caller's copy of the fool structure that it passes to xdr-fool is unchanged—only the function local variable fool-obj is changed. If fool-obj is not a locative, then rpc:resolve-locative does nothing.

Therefore, with the rpc:resolve-locative macro in place, the body of the function is written as if its argument is always a variable and never a locative. At the same time, the caller can pass either the structure itself or the locative of that structure.

Unfortunately, one more problem remains. This filter assumes that even when the stream is open for decoding (that is, filtering from the network to a Lisp data structure), the destination fool structure already exists. That is, the caller found (or created) an empty fool structure that it passed to the xdr-fool filter to be *filled up* with data decoded from the network.

An XDR filter function such as xdr-fool can perform another service for the caller by creating its own destination data structures when required. For example, if the stream is open for decoding (that is, filtering from the network to a Lisp data structure) and if the fool-obj is passed as a locative, then xdr-fool can examine the destination location in the caller to see if it really is a fool structure.

If the locative is pointing to a suitable structure, then it is used as is. However, if the locative is pointing to anything else, then the filter function can create the necessary structure and then store the structure itself back into the caller's data area using the locative that the caller passed. Once there is a suitable structure in the caller's data area (regardless of how it got there), you can proceed with the rpc:resolve-locative.

If all of this extra work were done in open code, the previous example would look something like the following:

```
(defun xdr-fool (stream fool-obj)
  (when (locativep fool-obj)
    (when (not (typep (contents fool-obj) 'fool))
      (setf (contents fool-obj) (make-fool)))
    (setf fool-obj (contents fool-obj)))
  (send stream :xdr-integer (locf (fool-a fool-obj)))
 * * *)
```

We can reduce the amount of code needed by using the **rpc:default-and-resolve** macro instead:

```
(defun xdr-fool (stream fool-obj)
  (rpc:default-and-resolve fool-obj fool make-fool)
  (send stream :xdr-integer (locf (fool-a fool-obj)))
 * * *)
```

Therefore, with the rpc:default-and-resolve macros as the first forms in each XDR filter function, the function automatically accommodates locative and nonlocative arguments and will create its own destination data structure when required.

As a second example, consider a filter that must behave differently when encoding than when decoding. The **:xdr-bool** method is reimplemented as a filter function (which is somewhat less efficient than **:xdr-bool**). A problem arises because Lisp sees true as non-**nil** and false as **nil** while XDR sees true as the integer 1 and false as the integer 0.

```
(defun xdr-boolean (stream  boolean)
  (ecase (send stream :transfer-direction)
     (:encode
        (rpc:resolve-locative boolean)
        (if boolean
            (send stream :xdr-unsigned 1)
            (send stream :xdr-unsigned 0)))

     (:decode
        (let ((number nil))
           (send stream :xdr-unsigned (locf number))
           (setf (contents boolean) (plusp number))))))
```

The `ecase` statement selects either encode or decode based on the transfer direction. It also signals an error if the transfer direction is anything other than `:encode` or `:decode`.

The encode logic has no use for a locative operand. It therefore starts by resolving `boolean` so that by the time the `if boolean` statement starts execution, `boolean` is an ordinary value, regardless of whether or not it initially was a locative. Once `boolean` is known, the encoding logic simply outputs an unsigned integer, 0 or 1.

The decode logic starts by decoding an unsigned integer into a local variable, `number`. Assuming that `number` decodes as either a 1 or 0, the `plusp` function effectively converts it into the equivalent Lisp boolean. The `setf` of the `contents` of `boolean` rather than of `boolean` itself effectively modifies the caller's variable that was passed to **xdr-boolean**.

The third example illustrates how a list of strings can be filtered. The XDR representation is a more-to-come boolean value followed by a string if the boolean is true. If the boolean is false, then the end of the list has been reached.

```
(defun xdr-string-list (stream str-lst)
  (ecase (send stream :transfer-direction)
    (:encode
       (rpc:resolve-locative str-lst)
       (dolist (str str-lst)
          (send stream :xdr-bool t) ;more to come
          (send stream :xdr-string str))
       (send stream :xdr-bool nil))  ;no more to come

    (:decode
       (let ((more-p nil)
             (str    nil))
          (setf (contents str-lst) ()) ; start empty
          (loop
             (send stream :xdr-bool (locf more-p))
             (unless more-p (return))
             (setf str nil)            ; don't reuse it
             (send stream :xdr-string (locf str))
             (push str (contents str-lst))))))))
```

Notice that the decode loop is careful to destroy the old value of str each time through the loop so that :xdr-string can allocate a new string. Also notice that since this loop used a push to attach each new string to the list, the order of the strings on the list are reversed from the way they appear in the XDR stream. If you want to maintain the order, then replace the last line with this one:

```
(setf (contents str-lst) (nconc (contents str-lst) (list str)))
```

This new line adds each new string to the end of the list rather than to the front, as push does.

XDR Memory

A.7 The following paragraphs discuss the flavors and methods associated with XDR memory.

rpc:xdr-memory-stream Flavor

This instantiable flavor allows a Lisp vector to be accessed as an XDR stream. **rpc:xdr-memory-stream** includes **rpc:xdr-stream** as a mixin so that all XDR primitive filters can be sent to this stream.

Unlike with most Explorer streams, you do not need to open an XDR memory buffer stream (although you may). The **:set-transfer-direction** method is called automatically based on the **:open**'s **:direction** argument. The **:set-transfer-direction** method causes the buffer and associated variables to be initialized for a new transfer.

:memory-buffer Method of **rpc:xdr-memory-stream**
:memory-buffer *new-memory-buffer-vector* Init option of **rpc:xdr-memory-stream**
:set-memory-buffer *new-memory-buffer-vector* Method of **rpc:xdr-memory-stream**

If the stream is open for encoding, then the XDR encoded bytes are collected in this buffer. If the stream is open for decoding, then the XDR encoded bytes are read from this buffer. The *new-memory-buffer-vector* argument represents the memory buffer you are filtering to or from.

If the buffer has a fill pointer, then it is set to the total size of the buffer by each **:set-transfer-direction** message.

:memory-buffer-end Method of **rpc:xdr-memory-stream**
:set-memory-buffer-end *end* Method of **rpc:xdr-memory-stream**

The **memory-buffer-end** is the upper exclusive limit beyond which any attempt to encode or decode signals the error **rpc:end-of-memory-buffer**. This value is automatically set to the total size of the buffer by **:set-transfer-direction**. Note that some filter primitives check *before* attempting a transfer so that the error is signaled early (before the end-of-file marker). For example, when dealing with a very large string, **:xdr-string** may see that the buffer will overflow. The method signals the error, even though the buffer pointer is still positioned in the middle of the buffer.

:memory-buffer-pointer Method of **rpc:xdr-memory-stream**
:set-memory-buffer-pointer *index* Method of **rpc:xdr-memory-stream**

This pointer is the index into **:memory-buffer** at which the next encoded byte will be read or the next decoded byte written. This value is automatically set to zero by the **:set-transfer-direction** message and is automatically incremented by the various XDR filters. No checks are made to assure that *index* lies within the **:memory-buffer-end** limit.

:read-pointer Method of **rpc:xdr-memory-stream**
:set-pointer *index* Method of **rpc:xdr-memory-stream**

These methods are the standard Explorer stream interface to the information contained in **:memory-buffer-pointer**. Attempts to set the pointer beyond **:memory-buffer-end** signals the standard file system errors.

XDR Conditions

A.8 A number of conditions can be signaled when XDR encounters an error. Each condition that is signaled contains a hierarchy of the other conditions upon which it was built. Each condition described in the following paragraphs is followed by a list displaying its condition-calling hierarchy. Although it is not listed, the condition being described is the last condition in the hierarchy.

rpc:unknown-union-discriminator (ferror rpc:xdr-error) Condition

This condition is signaled whenever a union discriminator is not in the discriminator alist, and no default discriminator has been provided.

The **:stream** message identifies the XDR stream.

rpc:end-of-memory-buffer (sys:end-of-file rpc:xdr-error) Condition

This condition is signaled whenever a remote procedure call attempts to read or write an XDR memory stream beyond the value of **memory-buffer-end**. Note that this error condition can be signaled before or after the actual transfer is attempted. For example, when dealing with a very large string, **:xdr-string** may see that the buffer will overflow. The method signals the error, even though the buffer pointer is still positioned in the middle of the buffer.

The **:stream** message identifies the XDR stream.

REMOTE PROCEDURE CALL (RPC)

B

Introduction

B.1 Sun Microsystems defined the Remote Procedure Call (RPC) protocol so that machines of different types could interact with each other on a procedure level. This interaction means that one machine can call a procedure on the other, pass arguments to that procedure, and then receive any returned values.

Because it is based on the XDR protocol, RPC can insure data compatibility between the machines (byte order, word length, and so on).

The RPC protocol is divided into two parts: a caller and a server. The following paragraphs discuss the Explorer implementation of these two parts.

The RPC Caller

B.2 An RPC caller issues a procedure call to an RPC server on a remote host, identifying the remote procedure by specifying the following minimum information:

- Remote program number
- Remote program version
- Remote procedure number

On the Explorer, the RPC caller is implemented by two functions: **callrpc** and **callrpc-spec**.

callrpc

B.2.1 The function description of **callrpc** is as follows:

callrpc *host prog# vers# proc# xdr-in in xdr-out out* &optional *credentials* (*protocol* **:udp**) — Function

The **callrpc** function issues a procedure call to an RPC server on a remote host.

The *host* argument identifies the remote host on which the procedure resides. The type of this argument must be acceptable to the **sys:parse-host** function (symbol, string, or host object). Also, LM is recognized as shorthand for the local Explorer system itself.

The *prog#*, *vers#*, and *proc#* arguments identify the remote program number, version number, and procedure number, respectively. These arguments must be specified as 32-bit unsigned integers.

The *in* argument represents the argument passed to the remote procedure. The RPC interface allows only one argument to be passed with a procedure call. The value of the *xdr-in* argument resembles the value of the *elt-xdr-function* argument of several of the **rpc:xdr-stream** methods; that is, it is the XDR filter function required to encode *in* onto the network.

NOTE: Both the *xdr-in* and the *xdr-out* arguments must be acceptable to the **xdr-io** macro discussed in the previous section.

The *out* argument is a locative of the structure that will receive the argument returned by the remote procedure. The *xdr-out* argument is the XDR filter function required to decode *out* from the network. If you supply no value for *out*, **callrpc** initializes a temporary variable to **nil** and passes the locative of that variable to *xdr-out*. On receiving the locative of something other than its output value, *xdr-out* must create the necessary value for the contents of the locative. XDR filter functions that use **rpc:default-and-resolve** automatically create this value, as do all of the XDR primitives.

You *should* supply a value for *out* if you are not familiar with the *xdr-out* function you are using. You *must* supply a value for *out* if the function specified by *xdr-out* does not automatically create its own output value. In these situations, you must define your own output variable, initialize it to the type of structure that the *xdr-out* function expects, and then pass the locative of your variable as *out*.

The optional *credentials* argument can be one of these possible values:

- The credentials symbol of an **nfs:with-credentials** macro
- An **rpc:opaque-auth** structure such as one returned by **rpc:authunix-create** or **rpc:make-opaque-auth**
- An **rpc:cred-verifier** structure such as that returned by **rpc:make-cred-verifier**
- **nil**

The optional *protocol* argument is one of two protocol keywords: either **:udp** (for accesses by the User Datagram Protocol) or **:busnet** (for accesses over the BusNet to the local 68020-based processor).

rpc:make-spec and callrpc-spec

B.2.2 Although some of the arguments that must be passed to **callrpc** are variable, several are usually constant. For the constant arguments, you can create a structure containing their values by using the **rpc:make-spec** function. Then you can use **callrpc-spec** in place of **callrpc** and its numerous arguments. The following paragraphs describe both **rpc:make-spec** and **callrpc-spec**.

rpc:make-spec *prog# vers# proc#* &optional (*xdr-in* **:xdr-void**) (*xdr-out* **:xdr-void**) (*protocol* **:udp**) Function

The **rpc:make-spec** function creates an **rpc:spec** structure that contains pertinent information about an RPC remote procedure. The **rpc:spec** structure is used in turn by the **callrpc-spec** function.

The arguments for **rpc:make-spec** have the same meanings that they have for the **callrpc** function just documented.

NOTE: Both the *xdr-in* and the *xdr-out* arguments must be acceptable to the **xdr-io** macro discussed in the previous section.

callrpc-spec *host spec in out* &optional *credentials* Function

The **callrpc-spec** function calls a procedure on a remote host.

The *host*, *in*, *out*, and *credentials* arguments are the same as those for **callrpc**.

The *spec* argument identifies an **rpc:spec** structure that has been previously created by **rpc:make-spec**.

The example that follows shows how an **rpc:spec** structure is created and used in a **callrpc-spec** function.

Example:

```
(defconstant spec
   (rpc:make-spec prog# vers# proc# xdr-in xdr-out :udp))

(callrpc-spec host spec in out cred)
```

RPC Port Mapping

B.3 RPC is port-oriented. Before **callrpc** or **callrpc-spec** can find a specific remote procedure, it must first locate the port associated with that procedure.

Each RPC host has a port map server (or *port mapper*) that contains the port numbers of all the program/version pairs available on that host. By convention, port mappers themselves are always on a known port.

To find the port number of a procedure on a remote host, issue an RPC request to the port mapper on the remote host, asking for the port number of the procedure. After you get the remote procedure's port number, you can issue another RPC request (specifying the port number you just acquired) to call the remote procedure.

Each new program/version pair must be logged with the port mapper before it can be called remotely. For more details about the port mapper, see Sun Microsystem's *Remote Procedure Call Protocol Specification*.

NOTE: The Explorer port mapper does not yet support the **callit** procedure described in Sun Microsystem's *Remote Procedure Call Protocol Specification*.

---

rpc:*pmap-getport-cache-p* Variable

Another, perhaps easier, way to allow **callrpc** or **callrpc-spec** to find the port number of a remote procedure is to enable the foreign-port cache of your local machine. This cache contains the port numbers of every remote procedure that has been called previously by the local machine. To enable the foreign-port cache, you must set **rpc:*pmap-getport-cache-p***.

If non-**nil**, **rpc:*pmap-getport-cache-p*** causes the local host to check its foreign-port cache before issuing a remote get port request. If the port number for the remote request resides in the cache, then that information is appended to the request and then sent to the remote host. In this way, one network access can be eliminated, because the remote host's port mapper need not be queried before sending a request.

Starting Port Map Servers

B.4 The port map server is designed to survive most situations. In particular, internal errors encountered during the execution of the port map server simply cause the error to be recorded and the server to be restarted. However, the port map server can be killed deliberately from the outside. Once it has been killed (for whatever reason), you must restart it manually from a Lisp Listener.

To restart the port map server, use **rpc:start-port-map-server**. Whenever you use this function to start a new instance of a server, any existing instances of that server are killed first. That is, this *start server* function always provides a clean start regardless of the initial conditions.

The **rpc:start-port-map-server** function has only one required argument—the protocol (which is usually **:udp**). This function also has several other optional arguments for user convenience. For example, **rpc:start-port-map-server** does *not* destroy the current port map when it restarts unless you explicitly request it to do so with an optional argument.

Ultimately, the most complex form you should ever have to enter in a Lisp Listener is the following:

```
(rpc:start-port-map-server :udp)
```

If you attempt to start a port map server and none can be found, then a new one is started.

Resetting Existing Servers

B.4.1 If a port map server appears to be running, according to the Peek Process menu, but you suspect that the server has become wedged for some reason, then you need not kill the process. Instead, select the Reset & Enable item from the Peek Process menu. This process has been designed so that a Reset & Enable does almost everything that killing and restarting the process does but without having to enter a Lisp form in a Lisp Listener. This is true for any RPC server.

Arresting Existing Servers

B.4.2 If you want a port map server to temporarily stop honoring requests from the network, then select the Arrest item from the Peek Process menu. When you are ready for the server to resume honoring requests, select UnArrest from the Peek Process menu. This procedure works for any RPC server.

The following constants, functions, and variables are associated with the port map server.

rpc:start-port-map-server *protocol* &optional *clear-port-map-p* *clear-foreign-port-map-cache-p* &rest *make-server-process-args* — Function

This function first kills all existing port map servers for the specified *protocol*. Then it starts a new instance of the port map server for the specified *protocol*.

The *protocol* argument can be **:udp**, **:busnet**, or an SRI NIC protocol number.

If *clear-port-map-p* is true, this function deletes all port map entries for *protocol*.

If *clear-foreign-port-map-cache-p* is true, this function deletes all port map entries of any foreign hosts for *protocol*.

The *make-server-process-args* argument is a list of parameters that are passed to the **rpc:make-server-process** function. These parameters are alternating keywords and values.

rpc:pmapprog — Constant

The value of this constant is the port map server program number to be used as the second argument to **callrpc**. The default value is 100000.

rpc:pmapvers — Constant

The value of this constant is the port map server version number to be used as the third argument to **callrpc**. The default value is 2.

rpc:pmapproc-null Constant
rpc:pmapproc-set Constant
rpc:pmapproc-unset Constant
rpc:pmapproc-getport Constant
rpc:pmapproc-dump Constant
rpc:pmapproc-callit Constant
rpc:pmapproc-limit Constant

The values of these constants are the RPC procedure numbers for the corresponding port map procedures to be used as the fourth argument to **callrpc**. The default values are 0, 1, 2, 3, 4, 5, and 6, respectively.

rpc:pmap-null-spec Constant
rpc:pmap-set-spec Constant
rpc:pmap-unset-spec Constant
rpc:pmap-getport-spec Constant
rpc:pmap-dump-spec Constant

These constants are the symbols to be used as the second argument to **callrpc-spec**.

The RPC Server

B.5 Stated somewhat simply, an RPC server accepts a remote procedure call from the network and then executes the call. Each machine running RPC has a minimum of one RPC server; Explorer's RPC allows multiple servers.

Each Explorer RPC server runs as a separate process that is associated with a unique program/version number pair. Within a particular program/version number pair, multiple procedures can reside.

Making a Function Available to an RPC Server

B.5.1 If you write a procedure to be called remotely by RPC, keep in mind that the procedure must accept only one argument and must return only one value. If multiple arguments or values are required, you must define a structure that contains the multiple arguments or values, so that RPC can pass the structure as a single unit.

If the structure you define does not correspond to the data types accepted by the XDR protocol, you must then define an XDR filter for it.

Once the procedure is written, you must use the **registerrpc** function to inform RPC of its existence.

registerrpc *prog# vers# proc# function xdr-in xdr-out* Function
&key (*protocol* **:udp**) &allow-other-keys

The **registerrpc** function registers the procedure identified by the *function* argument with the local machine's port mapper.

NOTE: Be sure that you register a procedure correctly the first time. For any program/version number pair, certain information is observed only on the first call to **registerrpc**. If you try to add new information in a subsequent call, that information is ignored.

The *prog#* and *vers#* arguments determine which local RPC server will handle the procedure. In selecting a program number for your procedure, remember the RPC standard developed by Sun Microsystems. The standard assigns program numbers in certain range groups, as follows:

#x00000000 — #x1FFFFFFF	Defined by Sun
#x20000000 — #x3FFFFFFF	Defined by user
#x40000000 — #x5FFFFFFF	Transient
#x60000000 — #xFFFFFFFF	Reserved

The *proc#* argument uniquely identifies a procedure within a particular program/version number pair.

The *function* argument is the name of the local function that implements the procedure identified by *proc#*.

The *xdr-in* argument filters the argument(s) received as input from the network back into a single XDR-format argument. If *xdr-in* is **:xdr-void**, the *function* argument should be the name of a function that has no required input arguments.

The *xdr-out* argument filters the single XDR-format argument so that it is acceptable for transfer across the network.

NOTE: Both the *xdr-in* and the *xdr-out* arguments must be acceptable to the **xdr-io** macro discussed in the previous section.

The *protocol* argument specifies which communications protocol that the particular RPC implementation is built on. The value of the argument is one of two protocol keywords: either **:udp** or **:busnet**.

The &allow-other-keys argument permits you to specify additional arguments to be supplied to the **rpc:make-server-process** function. These parameters are alternating keywords and values that include such information as the required port number and the process's pretty-print name.

Making Your Own RPC Server

B.5.2 At times, you may want to place several programs into a single RPC server, especially if several programs are relatively trivial in terms of execution time. Also, you may want to increase the efficiency of lower-level control, such as reading arguments directly (rather than having them filtered as single arguments by the default dispatcher). To do this, use the **rpc:make-server-process** function.

rpc:make-server-process *prog# vers#* &key (*protocol* **:udp**) *port* Function
(*dispatcher* '**rpc:universal-rpc-dispatcher**)
(*name* (**princ-to-string** *program*))
(*initial-form* '**rpc:universal-rpc-initial-form**)
initial-form-args run-reason server-id
(*flavor* '**rpc:server**)
(*receive-whostate* **rpc*default-server-who-state***)
make-process-args

The **rpc:make-server-process** creates a new RPC server process.

The *prog#* and *vers#* arguments (which are unsigned 32-bit integers or **nil**) identify the newly created process.

The *protocol* argument specifies which communications protocol that the particular RPC implementation is built on. The value of the argument is one of two protocol keywords: either **:udp** or **:busnet**. The default value is **:udp**.

Although *port* defaults to a randomly selected port, you can specify a particular port for the procedure. If you supply a value for *port*, it must be an unsigned 16-bit integer greater than 1.

The *dispatcher* argument is a **funcall**able Lisp object that performs server functions. Normally when you create your own server, you can accept the default value for the *dispatcher* argument, **rpc:universal-rpc-dispatcher**. The dispatcher is discussed in greater detail later in this appendix.

The *name* argument is a user-recognizable name of the *prog#* argument. This form is used by various Explorer utilities (such as Peek). The default value for *name* is the *prog#* written as a string. For example, network file system (NFS) would default to "100003", although a preferred name would be "NFS".

The *initial-form* is a **funcall**able object that starts the process. Its default value is **rpc:universal-rpc-initial-form**.

The *initial-form-args* are those arguments (if any) required by *initial-form*.

The *run-reason* argument, if non-**nil**, sends the newly created process a run reason, thereby activating the process. The default value is **nil**, meaning the process is not activated.

The *server-id* argument assigns each server an ID that can be used by the **rpc:kill-server-process** function to kill any previous instances of itself that are still active. The *server-id* argument can be any symbol other than **nil**. The **rpc:kill-server-process** function will not kill a *server-id* of **nil**.

The *flavor* argument identifies the flavor for the **make-process** function to instantiate. It should be **rpc:server** or a flavor based on **rpc:server**.

The *receive-whostate* argument is a string to be displayed as the who-state while waiting on an RPC call. The default value is **rpc:*default-server-who-state***.

The *make-process-args* argument is a list of parameters that are passed to the **make-process** function. These parameters are alternating keywords and values.

Registering Processes in a New Server

B.5.3 Because **registerrpc** creates a separate server for each program/version number pair, you cannot use it to register your procedures from different programs with the same server. Instead, you must use the **:register** method.

:register *program version dispatcher* &optional (*set-port-map-p* **t**)) Method of **rpc:server**

This method associates the version number *version* of the RPC program number *program* with the dispatch function *dispatcher*. If *set-port-map-p* is true (the default), then the program/version number pair is registered with the local port mapper for the server's protocol. If **:register** cannot register the program, it signals an **rpc:unable-to-set-port-map** error.

:unregister *program version* &optional (*arrest-on-empty-p* **t**) Method of **rpc:server**

This method disassociates version number *version* of the RPC program number *program* from any dispatch function. If *arrest-on-empty-p* is true and no programs remain registered, then **:unregister** posts an arrest reason of **rpc:no-registered-programs** for this process. If **:unregister** is unable to unregister the program, it signals an **rpc:unable-to-unset-port-map-warning** error.

The Dispatcher

B.6 When an RPC server receives a remote procedure call, it calls a dispatcher (either the default dispatcher or one that you have provided). The default dispatcher takes the incoming message from the network and extracts the destination program/version number pair from it. Matching this information to a function in its procedure registry, the default dispatcher then calls that function. The definition of the default dispatcher function follows:

rpc:universal-rpc-dispatcher *request stream* Function

This function is the default dispatcher function provided for **rpc:make-server-process**. The **rpc:universal-rpc-dispatcher** function executes the calling procedure and returns its result.

The *request* argument contains the header information from the RPC call in the form of an **rpc:svc-req** structure. This information includes the program/version number pair, procedure number, XDR filters, protocol, and so on. This information can be obtained by executing the functions discussed immediately after this description.

The *stream* argument identifies the XDR stream (positioned to begin reading arguments) that is received from the remote host requesting an RPC transfer.

The default dispatcher function also binds the **rpc:rpc-call-msg-header** variable to *request*. By declaring **rpc:rpc-call-msg-header** special in the *function* argument of **registerrpc**, you can access the extra **rpc:svc-req** information even without writing your own dispatcher.

Because it can access the program/version number pair, a single dispatcher can control multiple programs.

The following functions obtain information for the *request* argument of **rpc:universal-rpc-dispatcher**.

rpc:svq-req-program *request* Function
rpc:svq-req-version *request* Function
rpc:svq-req-procedure *request* Function
rpc:svq-req-credentials *request* Function

These functions extract the program/version number pair, procedure number, or credentials from the **rpc:svc-que** structure containing the RPC call header information.

RPC Functions

B.7 The following functions manipulate RPC server processes:

rpcinfo &quote *flag* &optional *host program version-number* Function

This function, which accepts its arguments in a UNIX-style syntax (no quoting is needed), displays information about the status of the specified RPC server. The following are the accepted values for *flag* along with the appropriate arguments (those in brackets are optional for the corresponding *flag* value):

–p [*host*] — Dumps the entire port map for *host* (which defaults to the local host) to the value specified by ***standard-output***.

–u *host program* [*version-number*] — Prints the status of the RPC server for version *version-number* of *program* on *host* for protocol UDP.

–b *host program* [*version-number*] — Prints the status of the RPC server for version *version-number* of *program* on *host* (which defaults to the local LX host) for protocol BusNet.

–t *host program* [*version-number*] — Prints the status of the RPC server for version *version-number* of *program* on *host* for protocol TCP.

The *program* argument can be an integer or a symbol. The *version-number* argument defaults to 1.

rpc:universal-rpc-initial-form &optional *signal-errors-p* Function

This function is the default initial form provided for the **rpc:make-server-process** function. As such, its only purpose is to send the newly created process a **:run** message.

If the *signal-errors-p* argument is non-**nil** and an error occurs in the server, the error enters the debugger. If this argument is **nil** and an error occurs in the server, the server is restarted.

rpc:kill-server-process *server-id* &optional *protocol* Function

The **rpc:kill-server-process** function kills RPC server processes identified by the *server-id* argument.

To kill a specific RPC server process, use the server ID provided by the **rpc:make-server-process** function. If *server-id* is the symbol **t**, the function kills all RPC server processes. If *server-id* is **nil** or is not a symbol, the function does nothing.

The optional *protocol* argument allows you to further identify the server process to be killed by its associated protocol. If *protocol* is **nil**, then all server processes with a server ID the same as *server-id* are killed, regardless of their associated protocol.

rpc:protocol-no *protocol* — Function

The **rpc:protocol-no** function returns the number associated with the protocol specified by *protocol.* The *protocol* can be either a protocol keyword (such as **:udp**) or the protocol number itself. Either way, the protocol number is returned. If the protocol is not known, this function signals **rpc:unknown-protocol**.

rpc:protocol-keyword *protocol* — Function

The **rpc:protocol-keyword** function returns the keyword associated with the protocol specified by *protocol,* which can be either a protocol number or the protocol keyword itself (such as **:udp**). Either way, the protocol keyword is returned. If the protocol is not known, this function signals **rpc:unknown-protocol**.

rpc:make-cred-verifier &key *credentials verifier* — Function
rpc:cv-credentials *cred-verifier-structure* — Function
rpc:cv-verifier *cred-verifier-structure* — Function

The **rpc:make-cred-verifier** function creates a *cred-verifier-structure* structure from the **rpc:opaque-auth** structures *credentials* and *verifier.*

The **rpc:cv-credentials** accessor function extracts the credentials from the **rpc:opaque-auth** structure. The **rpc:cv-verifier** accessor function extracts the verifier (or **nil** if there is no returned verifier) from the **rpc:opaque-auth** structure. Both of these functions are **setfable**.

RPC Variables

B.8 The descriptions of the more important RPC variables are as follows:

rpc:*rpc-default-area* — Variable

The value of this variable is the default consing area number for remote procedure calls and related servers.

rpc:*call-who-state* — Variable
rpc:*default-client-who-state* — Variable

The **rpc:*call-who-state*** variable, if non-**nil**, contains a string that replaces the who-state string displayed during a wait for an RPC client call reply. If **rpc:*call-who-state*** is **nil**, the value of **rpc:*default-client-who-state*** is used.

rpc:*default-server-who-state* — Variable

The **rpc:*default-server-who-state*** variable contains a string seen in the Peek utility whenever the server is waiting.

rpc:*default-unix-uid* — Variable

This variable contains the default UNIX user ID. Normally, this value is 0.

rpc:*default-unix-gid* — Variable

This variable contains the default UNIX group ID. Normally, this value is 1.

rpc:*callrpc-timeout* Variable

This variable contains the number of seconds allowed before each **:call** or **:broadcast** to a client times out. The default value is 10 seconds.

rpc:*callrpc-retrys* Variable

This variable contains the number of retries performed for each unsuccessful **:call** or **:broadcast** to a client. The default value is three retries.

rpc:*pmap-getport-timeout* Variable

This variable contains the number of seconds allowed before each **rpc:pmap-getport** times out. The port mapper uses this variable in lieu of **rpc:*callrpc-timeout***. The default value is five seconds.

rpc:*pmap-getport-retrys* Variable

This variable contains the number of retries allowed before each **rpc:pmap-getport** times out. The port mapper uses this variable in lieu of **rpc:*callrpc-retrys***. The default value is 12 retries.

rpc:*rpcinfo-name-number-alist* Variable

This parameter is an association list containing dotted pairs formed from RPC server name symbols and their associated RPC program numbers or synonym name symbols. The alist ignores case and maps all #_ (underscore) characters to #\- (hyphens) before lookup. This mapping also supports symbolic arguments to **rpcinfo**.

RPC Conditions

B.9 A number of conditions can be signaled when an RPC transfer encounters an error. Each condition that is signaled contains a hierarchy of the other conditions upon which it was built. Each condition described in the following paragraphs is followed by a list displaying its condition-calling hierarchy. Although it is not listed, the condition being described is the last condition in the hierarchy.

rpc:unknown-port (rpc:rpc-error) Condition

This condition is signaled when a network port number cannot be mapped back to its port object.

The **:protocol** message returns the number of the protocol that the request attempted to use, and **:port** returns the number of the port that was assigned to the request.

rpc:unknown-protocol (rpc:rpc-error) Condition

This condition is signaled when one of the protocol information access functions (such as **rpc:protocol-no** or **rpc:protocol-keyword**) cannot find the information it was asked for.

The **:protocol** message returns the number of the protocol that the request attempted to use, and **:attribute** returns the type of information that **rpc:unknown-protocol** could not find about the protocol.

rpc:unsupported-protocol (rpc:rpc-error) Condition

This condition is signaled when a client is instantiated with a known, but unsupported, protocol. If the protocol is unknown, **rpc:unknown-protocol** is signaled instead.

The **:protocol** message returns the number of the protocol that the request attempted to use.

rpc:non-call-msg-received (rpc:rpc-error rpc:server-error) Condition

This condition is signaled when an RPC server receives an RPC message that does not have a call message type.

The message **:msg-type** returns the message type actually received by the RPC request, and the **:xid** message returns the transaction ID of the request.

rpc:unable-to-set-port-map (rpc:rpc-error rpc:server-error) Condition

This condition is signaled when **rpc:pmap-set** indicates to **(:method rpc:server :register)** that it cannot set the specified program, version, and protocol to the specified port.

The **:program** and **:version** messages return the program and version numbers, respectively, of the RPC request. The **:protocol** message returns the number of the protocol that the request attempted to use, and **:port** returns the number of the port that was assigned to the request.

rpc:unable-to-unset-port-map-warning (sys:warning rpc:server-warning) Condition

This condition is signaled when **rpc:pmap-unset** indicates to **(:method rpc:server :unregister)** that it cannot unset the specified program and version.

The **:program** and **:version** messages return the program and version numbers, respectively, of the RPC request.

rpc:conflicting-ports (rpc:register-error) Condition

This condition is signaled when **registerrpc** requests to register a procedure on a specific port other than the one that the program was previously registered on.

The **:program**, **:version**, and **:procedure** messages return the program, version, and procedure numbers of the RPC request. The **:protocol** message returns the number of the protocol that the request attempted to use. The **:port** message returns the number of the port that the current call to **registerrpc** was requesting for this program/version number pair, and **:registered-port** returns the port it was already registered on.

---

rpc:call-error Flavor
rpc:call-warning Flavor

The **rpc:call-error** and the **rpc:call-warning** flavors provide the basis for a number of RPC conditions whose descriptions immediately follow this one. After each condition described in the following paragraphs is a list displaying its condition-calling hierarchy.

Both **rpc:call-error** and **rpc:call-warning** use **rpc:call-info-mixin**, which supports a number of messages that obtain general information about the call being attempted at the time of the error:

- **:rpc-host** — Who was being called
- **:program, :version, :procedure** — What was being called
- **:protocol, :protocol-keyword** — How they were being called
- **:port** — On which protocol port
- **:client** — Which **rpc:client** instance was handling the call

rpc:call-timeout (rpc:call-error) Condition

This condition is signaled when the **:call** method times out while waiting for a reply.

The **:retrys** message returns the number of retries attempted before time-out occurred, **:seconds** returns the number of seconds allotted for each try before the remote host times out, and **:discarded-replys** returns the number of reply messages (received from the remote host during the wait) that were not replies to the current call.

rpc:prog-unregistered (rpc:call-error) Condition

This condition is signaled when **rpc:pmap-getport** cannot find a port number for the remote host's requested program and version numbers.

rpc:unknown-reply-stat (rpc:call-error) Condition

This condition is signaled when an RPC reply is received with an unknown **reply-stat** field. You can obtain the unknown **reply-stat** field with the **:status** message.

rpc:garbage-args (rpc:call-error rpc:unsuccessful-accept-stat) Condition

This condition is signaled when an **accept-stat** of **rpc:garbage-args** is received in an RPC reply.

rpc:proc-unavail (rpc:call-error rpc:unsuccessful-accept-stat) Condition

This condition is signaled when an **accept-stat** of **rpc:proc-unavail** is received in an RPC reply.

rpc:prog-mismatch (rpc:call-error rpc:unsuccessful-accept-stat) Condition

This condition is signaled when an **accept-stat** of **rpc:prog-mismatch** is received in an RPC reply.

The **:low** and **:high** messages return the lowest and highest versions of the program served by the remote host, respectively.

rpc:prog-unavail (rpc:call-error rpc:unsuccessful-accept-stat) Condition

This condition is signaled when an **accept-stat** of **rpc:prog-unavail** is received in an RPC reply.

rpc:system-err (**rpc:call-error rpc:unsuccessful-accept-stat**) Condition

This condition is signaled when an **accept-stat** of **rpc:system-err** is received in an RPC reply.

rpc:unknown-accept-stat (**rpc:call-error rpc:unsuccessful-accept-stat**) Condition

This condition is signaled when an unknown **accept-stat** is received in an RPC reply.

The **:status** message returns the unknown accept status.

rpc:auth-error (**rpc:call-error rpc:reject-stat**) Condition

This condition is signaled when a **reject-stat** of **rpc:auth-error** is received in an RPC reply.

The **:auth-status** message returns a specific reason for the authentication error.

rpc:rpc-mismatch (**rpc:call-error rpc:reject-stat**) Condition

This condition is signaled when a **reject-stat** of **rpc:rpc-mismatch** is received in an RPC reply.

The **:low** and **:high** messages return the lowest and highest versions of RPC supported by the remote host.

rpc:unknown-reject-stat (**rpc:call-error rpc:reject-stat**) Condition

This condition is signaled when an unknown **reject-stat** is received in an RPC reply.

The **:reject-status** message returns the unknown status.

rpc:non-reply-msg-received-warning (**rpc:call-warning**) Condition

This condition is signaled when the reply to an RPC call does not have a msg-type of reply.

The **:msg-type** message returns the type of the RPC message actually received.

rpc:wrong-reply-xid-warning (**rpc:call-warning**) Condition

This condition is signaled when the XID field sent with the previous call does not match the XID of the current reply.

The **:call-xid** message returns the expected transaction ID, and **:reply-xid** returns the transaction ID actually received.

WRITING RPC SERVERS

Introduction

C.1 There are two ways to write a Remote Procedure Call (RPC) server. The first way uses **registerrpc** and is simpler because it lets **registerrpc** do most of the work. The generality of **registerrpc** imposes a small additional overhead, which should be undetectable. The only functional disadvantage of the **registerrpc** approach is that it forces you to treat all remote procedures as virtually standalone programs. With **registerrpc** you cannot, for example, establish a common set of error handlers for all remote procedures in a program.

The other way of writing RPC servers is to write your own dispatcher function to replace **rpc:universal-rpc-dispatcher**, the general-purpose dispatcher function that **registerrpc** uses. Your own dispatcher function gives you a central point from which you can dispatch execution to one of several action routines based on the program, version, and procedure number of the call. Similarly, having your own dispatcher function gives you a common point for call authentication and error handling.

A Sketch of RPC Serving

C.2 On the Explorer system, each RPC server is represented by a process built on the **rpc:server** flavor. The function **rpc:make-server-process** creates the process, initializes it appropriately from the arguments you give it, and then starts the server running or leaves it stopped as you request.

Each RPC server process spends most of its time waiting for data in the form of a call to arrive on a specified port of a specified transport protocol. One server process cannot wait on more than one port or on more than one protocol. However, it is a trivial matter to start extra identical servers if you need to handle, for example, two separate protocols.

When a call arrives, the server examines it enough to be sure that it really is a call that the server knows how to handle. Once satisfied, the server uses the program number and version number in the call to decide which dispatcher function, of those it has previously been told about, corresponds to this call (see the **:register** method of **rpc:server**). Finally, the server calls that dispatcher function with the RPC structure and the XDR stream from which the dispatcher can read any arguments to the call. It is up to this dispatcher function to do all further processing. Therefore, to summarize:

- A unique RPC process server must be dedicated to each port of each transport protocol that calls can arrive on.

- Each server can act as a front end to one or more dispatcher functions. Usually, there is exactly one dispatcher function per server.

- The server chooses a dispatcher function based on the program number and version number of the call. One dispatcher function can handle any number of program-version number pairs. (Usually there is exactly one program-version number pair per dispatcher.)

- The dispatcher function is called with the header information from the call and an XDR stream positioned to read the first procedure argument.

Once the dispatcher function is called, it is in complete charge of what—if anything—is done about actually executing the call. The server flavor merely provides methods to return values and report standard RPC errors.

In the case of **rpc:universal-rpc-dispatcher**, the default dispatcher function used by **registerrpc**, dispatching proceeds as follows:

1. Enforces the Null Procedure convention by immediately returning with no value if the procedure number is 0.

2. Determines the call's program, version, and procedure numbers plus the protocol the call arrived on.

3. Uses this information to look up the input XDR filter function, the procedure, and the return XDR filter function, which a previous **registerrpc** recorded in **rpc:*procedure-registry***.

4. Uses the XDR input filter to decode the input arguments, if any. That is, the **:xdr-void** filter reads nothing from the network.

5. Applies the procedure to the newly decoded input arguments. If the input filter is **:xdr-void**, then the procedure is called with no arguments.

6. Uses the XDR output filter to encode the procedure's return values, if any. That is, the **:xdr-void** filter writes no value into the RPC reply that is about to be returned to the caller.

A Four-Function Calculator Server

C.3 Before discussion of the two methods, perhaps it would be instructive to construct a simple RPC server by both techniques. This example demonstrates the differences in functionality and capability.

The following example defines an RPC server to add, subtract, multiply, and divide two integers. For simplicity, real world problems such as overflow and dividing by zero will be ignored. The server will have four procedures, one per operation. Each procedure will accept a structure of two integers as an argument and will return one integer as a result.

Procedures vs. Programs vs. Processes

C.3.1 Even at this early stage of definition, there is the question of how to decide on program numbers, version numbers, and procedure numbers. From one standpoint, you can look at the 32-bit program, version, and procedure numbers as one large 96-bit procedure identifier. In fact, if you have no particular requirements as to which procedures should be grouped together or kept separate or which should be allowed to run in parallel, then you could probably ignore the distinction of program, version, and procedure number fields in the 96-bit identifier.

On the other hand, a number of places in RPC processing treat a program-version number pair as a single identifier and the procedure number as a subidentifier. For example, the RPC port map server remembers the transport protocol port number of a program-version pair. That is, you cannot query a foreign host's port mapper about the port number on that foreign host of a program without also specifying the program's version, nor can you directly ask about a procedure. Similarly, an incoming RPC is handed off to a dispatcher function based upon the program-version number of that call.

Therefore, you should think in terms of a two-part RPC identifier: a program-version number pair and a procedure number within that program version. If the program and version number are always used as one unit, then why give names to the two subfields?

Actually, these subfields help solve the problem of upgrading software in the field. As an example, assume that you picked the following program, version, and procedure numbers for your calculator:

```
(defconstant calcprog  #x20000123)          ; Program number
(defconstant calcvers           1)          ; Version number
(defconstant calcproc-add       1)          ; Procedure number
(defconstant calcproc-subtract  2)
(defconstant calcproc-multiply  3)
(defconstant calcproc-divide    4)
```

Of course, if your calculator server is to be of much use, then you need to advertise the value of `calcprog` so that others on the network can call it. Now, what happens if you later decide to upgrade your server to handle the overflow and divide-by-zero problems ignored previously?

The solution is to leave the program number as is and change the version. You still have to inform other users of the new version number, but there is some help built into the RPC protocol for that. For example, if you make a remote procedure call to a foreign host to a program and version number that the foreign host has never heard of, then RPC returns an error as you would expect.

If, however, the foreign host has a different version of the program you asked for, then it informs you that it knows about the program you want, but that it only supports versions X through Y of that program (that is, the RPC error reply contains machine-readable version number limits). Therefore, a user at the terminal is informed of what is available and a client program could even perform automatic retry of the new version.

Although programmers speak of servers, that term does not appear in the RPC protocol specification. That is, the specification makes the following tacit assumptions (not enforced by RPC):

- All procedures that belong together in some sense constitute a server and share a common program number.

- Different versions of the same program number represent minor variations, updates, or enhancements to the same server.

- You can decide on a server-by-server basis whether a server runs in its own process or shares a process with several other servers.

This last point is actually unique to the Explorer system and outside the RPC standard. On the Explorer system, if two servers share one process, then only one procedure from one server or the other can execute at one time, and each procedure must complete execution before the other procedure starts. If the two servers are in different processes, then procedures from each server can execute in parallel. That is, the Explorer system's central-processing unit (CPU) is time-sliced between them.

Note that since the Explorer system's RPC process flavor is named **rpc:server**, there is a natural tendency to call the process the server. It is also natural to call each unique RPC program number a server as explained previously.

One of the assumptions made by **registerrpc** is that each program-version pair constitutes one server and runs in its own process independently of all other servers. Actually, whenever **registerrpc** is asked to register a procedure, it makes a quick check of the local port mapper. If this program-version pair is not already registered, then **registerrpc** registers it immediately and creates a new server process for it. Notice that **registerrpc** creates a new process based upon the absence of a port map entry, not upon the absence of a server process.

Arguments and Returned Values

C.3.2 The returned value of each of the calculator procedures is a single integer that is already supported by an XDR primitive method, **:xdr-integer**. The input arguments are a pair of integers not supported by a single existing XDR method. Thus, you must define a simple structure that holds the two integer arguments and then define an XDR function to filter that structure:

```
(defstruct (calc-args (:conc-name "ARG-")) x y)

(defun xdr-calc-args (stream args)
   (rpc:default-and-resolve args calc-args make-calc-args)
   (send stream :xdr-integer (locf (arg-x args)))
   (send stream :xdr-integer (locf (arg-y args))))
```

This code defines a simple structure named calc-args that has two slots, x and y, accessed with arg-x and arg-y. The xdr-calc-args function is bidirectional and filters a pair of integers between a calc-args structure in Lisp and a stream of bytes on the network.

You should probably use the **rpc:default-and-resolve** macro at the beginning of each of your XDR filters because it allows your users a number of standard options. In this particular case, it arranges to create a new calc-args structure if necessary to record integers freshly decoded from the network. That is, if stream is open for encoding Lisp to network and if args is a locative, then args is *resolved* into the actual argument, which is the contents of that locative. If args is not a locative, then it is used as is.

On the other hand, if stream is open for decoding network to Lisp, then it must be possible for xdr-calc-args to store the two freshly decoded integers back into the caller. There are several alternatives for what the caller can pass to as the args value to xdr-calc-args:

1. It is sufficient for the caller to pass a calc-args structure because the xdr-calc-args can store the decoded integers into the slots of that structure, thereby modifying the caller's original structure.

2. If the caller passes a locative of a calc-args structure, then that locative can simply be resolved into the structure itself and then used as in the first case.

3. If the caller passes a locative of something other than a calc-args structure, then rpc:default-and-resolve defaults the contents of that locative by calling make-calc-args and storing the results in the contents of the locative. The locative is then resolved.

Then again, if you do not want to bother with remembering all of that, just put a **rpc:default-and-resolve** at the beginning of your XDR filters.

Procedure Definitions

C.3.3 Next, you need to define four procedures to add, subtract, multiply, and divide two integers. Since these functions are going to be used with **registerrpc**, they must take exactly one argument (which is the `calc-args` structure) and return exactly one value, an integer.

```
(defun calc-add (args)
   (+ (arg-x args) (arg-y args)))

(defun calc-subtract (args)
   (- (arg-x args) (arg-y args)))

(defun calc-multiply (args)
   (* (arg-x args) (arg-y args)))

(defun calc-divide (args)
   (truncate (arg-x args) (arg-y args)))
```

At this point, you have defined the procedures necessary to use **registerrpc**. To register these procedures, simply do this:

```
(registerrpc calcprog calcvers calcproc-add      'calc-add      'xdr-calc-args :xdr-integer)
(registerrpc calcprog calcvers calcproc-subtract 'calc-subtract 'xdr-calc-args :xdr-integer)
(registerrpc calcprog calcvers calcproc-multiply 'calc-multiply 'xdr-calc-args :xdr-integer)
(registerrpc calcprog calcvers calcproc-divide   'calc-divide   'xdr-calc-args :xdr-integer)
```

Once your action routines are registered, the server is up and running. Actually, a new server is started in its own process as soon as the registration of `calcproc-add` is complete. The three following registrations to the same program-version pair simply record more procedure numbers for that server to handle.

Notice that **registerrpc** does something different the first time it is called for a given program-version number pair. It creates a new server process on the first call and merely adds to that original process's workload on all following calls for the same program-version pair. The **registerrpc** function takes a number of optional arguments, most of which are of interest only when the server is created.

For example, if you were to use Peek (press SYSTEM P) to look at the current processes, you would probably see one with a name something like `RPC UDP nnnnnnn v1`, where `nnnnnnn` is the decimal version of the program number that was registered. However, if you include a `:name "Calculator"` keyword in the first call to **registerrpc** for the `calcprog` program, then Peek shows something a little more informative, such as `RPC UDP Calculator v1`. If you include `:name` arguments to any of the following **registerrpc** requests, then they would be ignored because the server would already have been created.

When this server starts running, it can handle only add requests. As the remaining **registerrpc** requests execute, the server learns how to handle the other functions. While this piecemeal start-up creates no problem for this simple calculator server, other servers may require an all-or-nothing approach. That is, the server must be able to handle all of its defined procedures if it handles any of them. You can override **registerrpc**'s urge to immediately start a new server by doing something like the following:

```
(setf server-process (registerrpc calcprog calcvers
   calcproc-add 'calc-add 'xdr-calc-args :xdr-integer :run-reason nil))
```

If this were the first **registerrpc** to be executed for `calcprog` and `calcvers`, then a new server would have been created and recorded in `server-process`, but that server would be stopped because it would have no run reason. After all of the server's other procedures have been registered, then you could do the following to enable the process and start it running:

```
(process-enable server-process)
```

Actually, **registerrpc** has several other optional arguments that give your server a better outward appearance, especially from the Peek process menu. For example, your calculator server would show up under Peek as follows:

```
RPC protocol 536871203 v1          Waiting RPC Call
```

In this display, *protocol* is the keyword value of the `:protocol` parameter of **registerrpc** (usually UDP or BUSNET), and `536871203` is the decimal equivalent of `calcprog`, `#x20000123`, and `Waiting RPC Call` is the default RPC receive status. If you add `:name "Calculator"` to the initial call to `registerrpc`, then Peek would have shown the following:

```
RPC protocol Calculator v1          Waiting RPC Call
```

Similarly, you can change the receive status displayed by Peek by using `:receive-whostate "Waiting Calculation"` to produce the following display:

```
RPC protocol Calculator v1          Waiting Calculation
```

The `:server-id` is a more practical option to **registerrpc**. You can kill any process from Peek with a menu, but it is also sometimes useful to be able to kill a process from another function. For example, if you were to write a utility function, `start-calculator-server`, then it would be nice if that function could kill any existing servers so that it could start afresh. All processes known to the system are recorded in the global variable **sys:all-processes** and it is easy to recognize RPC server processes (they are of type **rpc:server**). But, how do you tell which RPC server process is a calculator server?

If you include `:server-id 'calculator` in your original call to **registerrpc**, then you can call `(rpc:kill-server-process 'calculator)` to get rid of any previous calculator servers.

An Alternate Calculator Service

C.3.4 If the simple dispatching action of the default dispatcher function used by **registerrpc** is not sufficient, then you can write your own dispatcher. Once you have a custom dispatcher function, you can either use **registerrpc** to establish your new dispatcher as an RPC server, or you can call **rpc:make-server-process** yourself.

```
(defun calc-dispatcher (svc-request stream)
  "Calculator Server dispatcher function"
  (let ((args  (make-calc-args)))      ;argument structure
    (flet ((get-calc-args-or-return ()
             "decode CALC-ARGS from STREAM or return decode error"
           (when (null (send stream :getargs :xdr-calc-args (locf args)))
             ;;then failed to decode args, so report it and return
             (send stream :svcerr-decode)
             (return-from calc-dispatcher)))

           (calc-error-handler (condition)
              "report any error as SYSTEMERR and return"
              (declare (ignore condition))
              (send stream :svcerr-systemerr)
              (return-from calc-dispatcher)))

      (condition-bind ((error #'calc-error-handler))
        (case (rpc:svc-req-procedure svc-request)
              (0                          ; null procedure
               (send stream :sendreply :xdr-void nil))
              (#.CALCPROC-ADD             ; add procedure
               (get-calc-args-or-return)
               (send stream :sendreply :xdr-integer (calc-add args)))
              (#.CALCPROC-SUBTRACT        ; subtract procedure
               (get-calc-args-or-return)
               (send stream :sendreply :xdr-integer (calc-subtract args)))
              (#.CALCPROC-MULTIPLY        ; multiply procedure
               (get-calc-args-or-return)
               (send stream :sendreply :xdr-integer (calc-multiply args)))
              (#.CALCPROC-DIVIDE          ; divide procedure
               (get-calc-args-or-return)
               (send stream :sendreply :xdr-integer (calc-divide args)))
              (otherwise                  ; unsupported procedure
               (send stream :svcerr-noproc))))))
```

This previous example was intended to be as much like the **registerrpc** version as possible for illustration purposes. Of course, the whole point of writing your own dispatcher is to allow you to do things differently. Therefore, consider a variation of this custom dispatcher example. Differences between these versions are shown by striking out lines which have been deleted in the new version and marking new lines with revision bars on the left.

```
(defun calc-dispatcher (svc-request stream)
  "Calculator Server dispatcher function"
   ~~(let ((args (make-calc-args)))-------------; argument structure--------~~
   (let ((x nil)                               ; individual x and y operands
         (y nil))
      (flet ((get-calc-arg-or-return (arg)
               "Decode an integer from STREAM or return decode error"
               (when (null (send stream :getargs :xdr-integer arg))
               ;; Then failed to decode this arg, so report it and return
                   (send stream :svcerr-decode)
                   (return-from calc-dispatcher)))

          . . .

              (case (rpc:svc-req-procedure svc-request)
                 (0                            ; null procedure
                   (send stream :sendreply :xdr-void nil))
                 (#.CALCPROC-ADD               ; add procedure
                   ~~(get-calc-args-or-return)------------------------------~~
                   ~~(send stream :sendreply :xdr-integer (calc-add args)))~~
                   (get-calc-arg-or-return (locf x))
                   (get-calc-arg-or-return (locf y))
                   (send stream :sendreply :xdr-integer (+ x y)))

          . . .

                 (otherwise                    ;unsupported procedure
                   (send stream :svcerr-noproc)))))))
```

The principal changes in this version are *. The let statement now binds the actual arguments x and y rather than a structure to hold the pair of arguments *. The get-calc-args-or-return is now get-calc-arg-or-return (singular) and now just reads one integer argument off the network and stores it into its argument. Inside the case statement clauses, the two arguments are now read separately and the operation is performed directly rather than calling an intermediate function.

This example shows that the repeated statement that remote procedures must be written to accept exactly one argument is false (or at least not universally true). For simple arguments such as the two integers, it does not matter too much whether they are first collected into a structure or they are used directly off the wire. However, for remote procedures that must read large data buffers, the extra step could mean a large overhead.

How to Avoid Consing

C.3.5 In the first example with the make-calc-args call in the let, a calc-args structure is consed up each time the dispatcher is called (which is once per remote procedure call). In this case, the structure is so small that it is probably not worth trying to prevent consing. But in other cases, it might be important.

One alternative is to simply replace the let in the dispatcher with a global variable. For example:

```
(defvar *args* (make-calc-args))
(defun calc-dispatcher (svc-request stream)
   (flet ((get-calc-args-or-return ()
            ~~(when (null (send stream :getargs :xdr-calc-args (locf args)))~~--
            (when (null (send stream :getargs :xdr-calc-args (locf *args*)))
              (send stream :svcerr-decode)
              (return-from calc-dispatcher)))
      . . .)
```

Now, each call to the dispatcher reuses the same calc-args structure in the global variable *args* and avoids consing up a new one.

Of course, this alternative tacitly assumes that there will never be more than one calculator server running at one time. If you want to avoid consing of multiple calculator servers, then you have to provide a permanent copy of a calc-args structure for each server. The simplest way to do this is to create a calculator server flavor:

```
(defflavor calc-server
          ((args nil))                      ; instance variable
          (rpc:server)                      ; component flavor
   :settable-instance-variables)            ; make args gettable, settable, and inittable

(defmethod (calc-server :after :init) (init-plist)
   (declare (ignore init-plist))
   (setf args (make-calc-args)))

(defun-method calc-dispatcher calc-server (svc-request stream)
   (flet ((get-calc-args-or-return ()
           (when (null (send stream :getargs :xdr-calc-args (locf args)))
             (send stream :svcerr-decode)
             (return-from calc-dispatcher)))
      . . .)
```

This defines the flavor calc-server to be just like the rpc:server flavor except that calc-server includes an extra instance variable, the calc-args structure. The calc-dispatcher function has become a **defun-method** so that the compiler will know that the (locf args) is referring to an instance variable of the flavor calc-server created; each will always have its own private copy of its input argument structure.

rpc:server Features Error Handling

C.3.6 The **rpc:server** flavor is designed to be highly survivable. That is, to the greatest extent possible, errors in the dispatcher function do not cause a server built on **rpc:server** to terminate or to unexpectedly enter the error handler. There are a number of consequences of this design.

First, it is difficult to intentionally debug a process that is deliberating hiding all of its errors. Therefore, each server built on **rpc:server** includes three debug-related instance variables: **signal-errors-p**, **condition-count**, and **condition-history**. If **signal-errors-p** is false when an error occurs, then **condition-count** is incremented, the error condition object is pushed onto **condition-history**, and the server is restarted to wait for the next call.

The simplest way to determine if a server has suffered an error is to use Peek to inspect the server process. The Inspector's display includes the server's instance variables such as **condition-count** and **condition-history**. The Inspector also allows you to modify the value of **signal-errors-p** to be true. If anything in the server signals an error while **signal-errors-p** is true or if any signaled error is ever classed as a dangerous or debugging condition, then the server enters the error handler. Since RPC servers are usually running in background (that is, they have no window of their own), then when the server enters the error handler, you receive a notification that some process got an error. You can then press TERM 0 S to see the error handler typeout.

In case of error, one of the available proceed types is to *restart* the server. Similarly, in the Peek display, one of the options is to `Reset` or to `Reset & Enable` the process. Either of these options causes the server process to stop whatever the server is doing (that is, all outstanding **unwind-protects** are honored). Any outstanding RPCs are returned to their RPC callers as system errors.

Next, the server attempts to reestablish itself as nearly as possible to the state it was in immediately after being created. That is, the intent is that you should never need to kill and then restart a server that has become wedged. Instead, you should never have to do anything other than click on `Reset & Enable` in the Peek menu.

Normally, a flavor instance receives an **:init** message exactly once in its life: immediately after it is instantiated. Part of **rpc:server**'s technique of resetting everything is to send itself another **:init** message each time its process is reset. This technique is the actual reason why the previous example initialized its `args` instance variable in its `:after :init` method rather than by providing a default value to the instance variable. If the **defflavor** reset the server, you are guaranteed to get a new `calc-args` structure in the `args` instance variable just in case it had been corrupted in some manner.

If, despite all of these precautions, you do need to kill an RPC server, then **rpc:server** has been designed to die immediately and completely. Without these special provisions, a server can get into certain states from which it cannot be killed.

Once the RPC server has been killed, you need to create a new server manually. Nothing in the system automatically restarts permanent servers. The principal server that supports RPC is the port mapper. You can restart the port mapper with the **rpc:start-port-map-server** function. Use **nfs:start-mount-server** and **nfs:start-nfs-server** functions to start those servers. It would be a good idea to create a similar start server function for any RPC servers you might write.

Note that UNIX implementations of RPC warn you that if you restart the UNIX port map server, that you must restart all other RPC servers on that system. You do not have this problem on the Explorer system. The port map information is held in a global variable, not inside the process itself. Therefore, killing one port map server and instantiating another one has no effect on previously registered entries. The **rpc:start-port-map-server** function does have optional arguments to allow you to clear out the local port map and the foreign port map cache if you wish.

Tracing

C.4 There are two levels of built-in trace: RPC message level and RPC buffer level. The message level is controlled by the **rpc:*rpc-msg-trace-p*** trace flag, which basically tells you who called whom and who replied to whom. The buffer level is controlled by the **rpc:*rpc-buffer-trace-p*** trace flag and gives you a dump of the RPC message buffer with the header portion formatted. There is no particular advantage in turning both of these traces on at the same time.

If either of these global special variables is non-**nil**, then the corresponding trace is printed to the ***trace-output*** stream. If an RPC client on the Explorer system is active, then your trace is usually displayed in the window you are using. However, RPC servers usually run in background and have no window. Therefore, your trace appears in one of the small typeout windows.

A better alternative for tracing background RPC activity is to set the appropriate trace flag to a stream rather than to another non-**nil** value, such as the background typeout window. For example, if your are in a Lisp Listener and set **rpc:*rpc-msg-trace-p*** to ***terminal-io***, then all of your traces from all RPC clients and servers are displayed in your Lisp Listener. If you use **dribble-all** before you set the trace flag, then the dribble file collects your trace.

Annotation

C.5 Although RPC deals exclusively in program, version, and procedure numbers, you can have your traces annotated with program and procedure names. Also, you can provide mnemonic program names for the **rpcinfo** utility. Use the function **rpc:define-program-procedure-name** to associate a comment string with a given program and procedure number during tracing. For example:

```
(rpc:define-program-procedure-name 100000 1 "Pmap Getport")
```

This form causes any trace of procedure 1 of program 100000 to be labeled as `"Pmap Getport"`.

Use the function **rpc:define-rpcinfo-name** to associate mnemonic symbols with program number for input to and display by **rpcinfo**:

```
(rpc:define-rpcinfo-name 100000 'PMAP 'PORT-MAP 'PORT-MAPPER)
```

This form allows an **rpcinfo** port map dump to label an entry with a program number of 100000 as `"PMAP"`. Similarly, instead of entering the program number 100000 as an argument to **rpcinfo**, you now have the option of specifying `'PORT-MAP`, `'PMAP`, or `'PORT-MAPPER`.

GLOSSARY

a

acknowledgment A signal indicating that a transmitted packet has been received by the user at a remote host.

b

bad state A state of the connection that is inappropriate for the current operation that is being attempted.

c

channel A communications link. Packets or data streams are not sent between the hosts themselves; they are sent between channels that are associated with user processes on the hosts. Each channel is identified by a 32-bit string, which includes fields for the subnet involved, the host involved, the index entry that identifies the user process, and a unique index entry.

Chaosnet A communications protocol that provides an Ethernet protocol between two user processes on different machines. The protocol is a full-duplex, packet-transmission channel. The term *Chaos* in the name refers to the fact that there is no central control.

Chaosnet address A 16-bit number that is used to identify a host on the network. The most-significant eight bits identify the subnet that the host uses, and the least-significant eight bits are a unique ID for the host itself. Neither the subnet number nor the host ID can be zero. Addresses are used in the routing of packets.

conn A data object representing one side of a connection. A conn is associated with the local process involved in a connection.

connection An object that represents a data transmission channel between two user processes. The channel itself is full-duplex and transmits data in packets. To maintain its reliability, the channel may halt communications in the event of an error, rather than running the risk of garbling, losing, duplicating, or resequencing any packets in transmission. In the event of such a halt in communications, Chaosnet informs both user processes. A connection is identified by the Chaosnet addresses and the local indexes assigned by the two hosts. The interpretation of packets or byte streams is dependent on the program or high-level protocol that is using the connection.

contact name

A string that is used to find the process with which to connect. A connection includes a requestor process and a listener process. The requestor process issues a request for connection (RFC) packet that contains the contact name. The RFC packet is like an invitation, and the contact name is like the name of the person invited. When the listener process hears its contact name, it decides whether to agree to a connection. Once a connection is established, the contact name of a user process is no longer used by the software. The Explorer system remembers what contact name was used to open a connection, but this is for a user's information only. On the other hand, the contact name can be the name of a standard server, such as Telnet. In this case, the receiving host creates a process with which to respond; the process executes the program for that server. In the case where two existing processes that already know about each other want to establish a connection, the two of them must agree on a contact name. Then, one process must send the request while the other process listens. The two of them must agree between themselves about what each process is to do. Contact names can be any string of characters terminated by a space. Explorers accept lowercase as well as uppercase characters in a contact name. However, other hosts may not accept lowercase. All hosts accept uppercase letters, numbers, and ASCII punctuation. The maximum length of a contact name is limited only by the packet size. However, on Incompatible Timesharing System (ITS) hosts, the names of automatically started servers are limited to six characters by the file system. A contact name is terminated by a space. If an RFC packet contains data beyond the contact name, the data is intended for interpretation by the listener process. The listener process can use this extra data to decide whether to accept the connection.

controlled packet

A packet that is retransmitted until receipt of the packet is acknowledged by the other end of a connection.

e

Ethernet

The DEC-INTEL-XEROX® (DIX) standard network communications system, version 1.0, 30 September 1983.

f

foreign host

A foreign host is a non-Explorer machine.

foreign protocol

A foreign protocol is a non-Chaosnet protocol.

h

header

The first part of a data transmission, containing information about the following data packet, such as the length of the packet.

XEROX is a registered trademark of XEROX Corporation.

n

Network Control Program (NCP)

A program that works to maintain the integrity of data transmission with an optimal rate of data flow.

p

packet

The basic unit for data transmission in Chaosnet. Each packet contains an *opcode* , which has an 8-bit number, a three-letter code, and a name. The 8-bit number identifies the function of the packet. Opcodes less than 200 octal have a special purpose. Each of these opcodes has an assigned name and a specific function. Opcodes 200 through 277 are used for 8-bit user data; opcodes 300 through 377 are used for 16-bit user data. The three-letter code identifies what kind of packet it is. For example, the code for an RFC packet is RFC, which stands for request for connection. Packets can be either *uncontrolled* or *controlled.* Uncontrolled (UNC) packets are used for low-level functions of Chaosnet such as error control. They are never retransmitted. Uncontrolled packets are different from controlled packets, which are retransmitted by a host until receipt of the packet is signaled. Most packets are controlled packets. User packets are always controlled packets unless a user deliberately uses an uncontrolled packet. The following are important packets:

answer (ANS) packet — A packet sent by a server process to a requestor process. It contains a string that is the server's response to an RFC. However, it does not open a connection.

close (CLS) packet — A packet that can be used in the following ways:

- To act as a negative answer to a request for connection. In this case, a server refused the RFC packet.
- To close a connection that had previously been open. Any packets in transit may be lost.

A CLS packet is an uncontrolled packet that contains a string that explains the reason for refusal. Use of a CLS packet is not absolutely necessary. One process may simply shut down its end of a connection. The next time the other process tries to use the connection it still thinks it has, it receives an LOS packet.

end-of-file (EOF) packet — A packet that signals the end of a stream of data. It is a controlled packet that contains only an end-of-file mark. Do not put data in an EOF packet, since Chaosnet will ignore any data present. In other words, the byte count for the data in an EOF packet should always be zero.

forward (FWD) packet — A packet used by a server process to turn down a request for connection. However, an FWD packet contains a string that suggests to the user process that it try a different host or a different contact name.

lossage (LOS) packet — An uncontrolled packet that one Network Control Program uses to tell another that an error has occurred. It contains a string that explains the nature of the problem. An LOS packet is sent in response to receiving a packet for a broken or nonexistent connection.

listen (LSN) packet — A packet used by the listener process to listen for a contact name.

open (OPN) packet — A packet sent by a server when the server agrees to a connection with a user process. Since it is a controlled packet, it is retransmitted until it is acknowledged.

request for connection (RFC) packet — A packet that a user process sends to initiate the opening of a connection with a server process. Sending an RFC packet is always the first step the user process takes to open the connection. It contains a contact name, which can be followed by other arguments to the server. The arguments are delimited by a space character. Since an RFC is a controlled packet, it is retransmitted until the server sends back a response, or the request times out.

sense (SNS) packet — An uncontrolled packet that is used when one end of a connection wants the other to send back a status (STS) packet.

status (STS) packet — An uncontrolled packet sent from one host to another in order to acknowledge the receipt of packets. The second host then knows it does not have to attempt retransmission of the packets.

r

receipt

A signal indicating that a transmitted packet has been received by the machine at a remote host.

route

The path that a packet takes to reach the host specified by the destination address field of the packet. A route is direct if there is a direct hardware connection between the two hosts involved. The packet is simply transmitted on the subnet between the two hosts. A route is indirect if the packet must go through several subnets before it reaches the target host. Any host that is connected to more than one subnet can be called on by the network to forward the packet. Explorer tries to determine the best route available for the packet to take with a *routing table*.

routing table

Each host has its own *routing table*, which tells the host the best way to send packets to different hosts. The table is arranged according to subnet because destination hosts are identified with a particular subnet. When forwarding a packet, a host uses the table to forward the packet to the host that is the best bridge to the destination host. Each packet contains a forwarding-count field that is incremented by one for each host that forwards the packet. If the field reaches a maximum value of 15, the packet is discarded. An Explorer signals an error condition when a packet is discarded for this reason. An Explorer may indicate that no viable connection can be established between the two particular hosts.

s

server process

The best way to imagine the server process in relation to a user process is to think of the two as responder and initiator. For example, when one process sends an RFC packet, it is acting as a user process. When the other process sends back an OPN packet, it is acting as the server process. Keep in mind that the concept of a server process and a user process is dynamic. Like two people in a conversation, who is questioning and who is answering can switch back and forth.

simple transaction

This is when a user process sends an RFC packet to a server process and the server process returns an ANS packet.

u

uncontrolled packet

A packet that is not retransmitted. Receipt of the packet is not acknowledged by the other end of a connection.

user process

The best way to imagine the user process in relation to a server process is to think of the two as initiator and responder. For example, when one process sends an RFC packet, it is acting as a user process. When the other process sends back an OPN packet, it is acting as the server process. Keep in mind that the concept of a server process and a user process is dynamic. Like two people in a conversation, who is questioning and who is answering can switch back and forth.

INDEX

Introduction

The indexes for this Explorer software manual are divided into several subindexes. Each subindex contains all the entries for a particular category, such as functions, variables, or concepts. The various subindexes for this manual and the pages on which they begin are as follows:

Alphabetization Scheme

The alphabetization scheme used in this index ignores package names and nonalphabetic symbol prefixes for the purposes of sorting. For example, the **rpc:*callrpc-retrys*** variable is sorted under the entries for the letter C rather than under the letter R.

Hyphens are sorted after spaces. Consequently, the multiple menus entry precedes the multiple-choice facility entry.

Underscore characters are sorted after hyphens. Consequently, the **xdr-io** macro precedes the **xdr_destroy** macro.

General

Conditions

Flavors

Functions

Operations

Variables

Data Systems Group – Austin
Documentation Questionnaire

Explorer Networking Reference

Do you use other TI manuals? If so, which one(s)?

____________________ ____________________

____________________ ____________________

____________________ ____________________

How would you rate the quality of our manuals?

	Excellent	Good	Fair	Poor
Accuracy	______	______	______	______
Organization	______	______	______	______
Clarity	______	______	______	______
Completeness	______	______	______	______
Overall design	______	______	______	______
Size	______	______	______	______
Illustrations	______	______	______	______
Examples	______	______	______	______
Index	______	______	______	______
Binding method	______	______	______	______

Was the quality of documentation a criterion in your selection of hardware or software?

□ Yes □ No

How do you find the technical level of our manuals?

□ Written for a more experienced user than yourself

□ Written for a user with the same experience

□ Written for a less experienced user than yourself

What is your experience using computers?

□ Less than 1 year □ 1-5 years □ 5-10 years □ Over 10 years

We appreciate your taking the time to complete this questionnaire. If you have additional comments about the quality of our manuals, please write them in the space below. Please be specific.

____

____

____

____

____

Name ____________________ Title/Occupation ____________________

Company Name ____

Address ____________________ City/State/Zip ____________________

Telephone ____________________ Date ____________________

TAPE EDGE TO SEAL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DATA SYSTEMS GROUP
ATTN: PUBLISHING CENTER
P.O. Box 2909 M/S 2146
Austin, Texas 78769-9990

FOLD