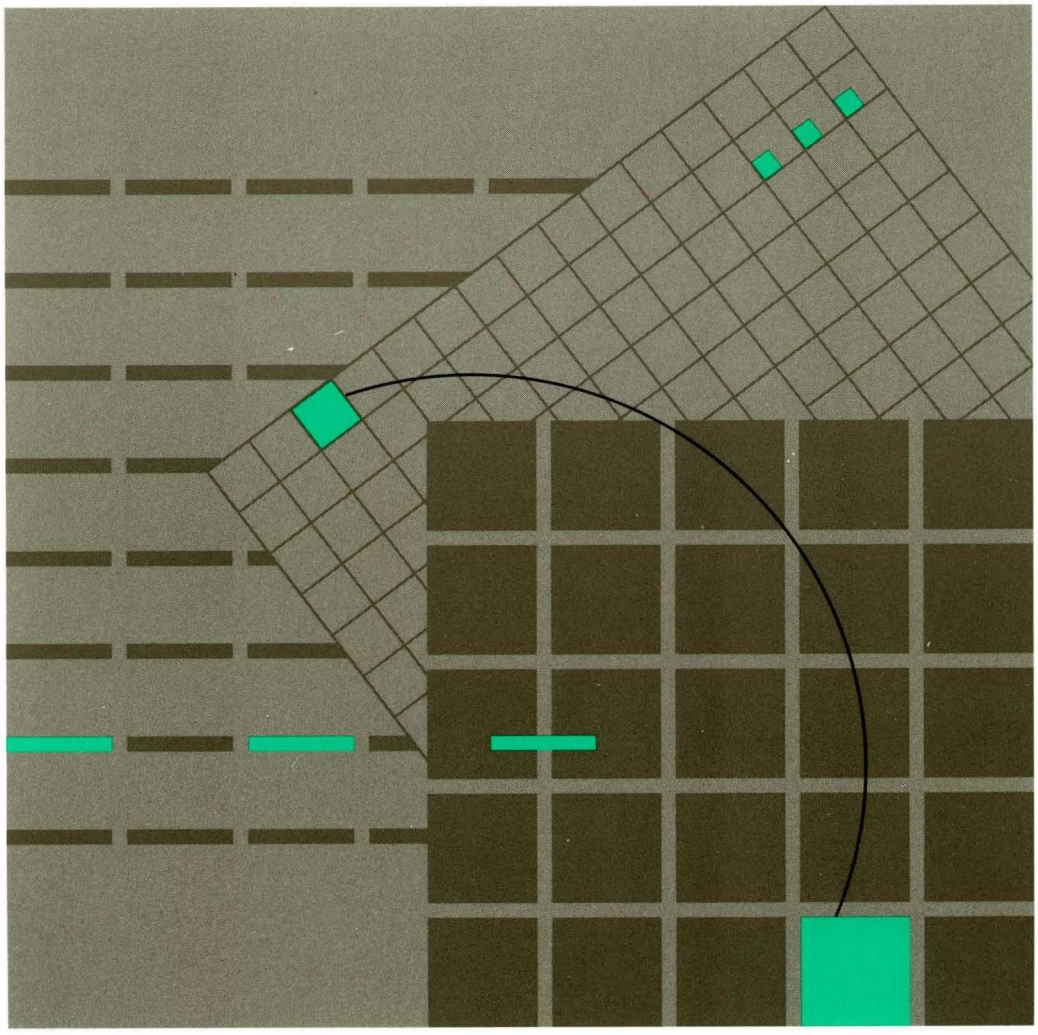
Transmission Control Protocol/
Internet Protocol

TCP/IP Version 2.0 for DOS: Programmer's Reference



Transmission Control Protocol
Internet Protocol

TCP/IP Version 2.0 for DOS:
Programmer's Reference

SC31-6153-0

IBM Transmission Control Protocol/ Internet Protocol Version 2.0 for DOS:

Programmer's Reference

SC31-6153-0

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xiii.

First Edition (September 1991)

This edition applies to the IBM Transmission Control Protocol/Internet Protocol Version 2.0 for DOS licensed program.

Publications are not stocked at the address given below. If you want more IBM publications, ask your IBM representative or write to the IBM branch office serving your locality.

A form for your comments is provided at the back of this document. If the form has been removed, you may address comments to:

IBM Corporation
Department E15
P.O. Box 12195
Research Triangle Park, North Carolina 27709
U.S.A.

IBM may use or distribute any of the information you supply in any way or distribute any of the information you supply without incurring any obligation to you.

IBM is required to include the following statements in order to distribute portions of this document and the software described herein to which contributions have been made by Sun Microsystems, Massachusetts Institute of Technology, Digital Equipment Corporation, and The University of California.

Contents

Figures

Tables

Notices

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Program Licensed Agreement.

Any reference to an IBM licensed program in this document is not intended to state or imply that only IBM's program may be used.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send inquiries, in writing, to the IBM Director of Commercial Relations, International Business Machines Corporation, Purchase, New York, 10577.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make them available in all countries in which IBM operates.

This document is not intended for production use and is furnished as is without any warranty of any kind, and all warranties are hereby disclaimed including the warranties of merchantability and fitness for a particular purpose.

Trademarks

The following terms, denoted by an asterisk (*) at their first occurrences in this publication, are trademarks of IBM Corporation in the United States or other countries:

AIX	IBM	Micro Channel
PC/XT	Personal Computer AT	Personal Computer XT
Personal System/2	PS/2	

The following terms, denoted by a double asterisk (**) at their first occurrences in this publication, are trademarks of other companies:

Trademark	Owned By
Ethernet	Xerox Corporation
Intel	Intel Corporation
Microsoft C	Microsoft Corporation
Motorola	Motorola, Inc.
MS-DOS	Microsoft Corporation
NDIS	3Com Corporation/Microsoft Corporation
Network File System	Sun Microsystems, Inc.
NFS	Sun Microsystems, Inc.
NICps/2	Ungermann-Bass Corporation
PostScript	Adobe Systems, Inc.
Ungermann-Bass	Ungermann-Bass Corporation
UNIX	UNIX System Laboratories, Inc.

About This Book

IBM TCP/IP Version 2.0 for DOS: Programmer's Reference describes the rountines for application programming in IBM* Transmission Control Protocol/Internet Protocol Version 2.0 for Disk Operating System (TCP/IP Version 2.0 for DOS) software on a personal computer (PC). TCP/IP for DOS is the base product. The following optional kits are offered separately:

- Network File System** (NFS**) Kit
- Programmer's Tool Kit.

The NFS Kit is a communication option that allows you to communicate with other NFS servers, and access files and output devices located on that server.

The Programmer's Tool Kit is a set of Application Programming Interfaces (APIs) that allow a programmer to develop custom code that accesses the capabilities of TCP/IP for DOS.

Note: In this book, PC refers to personal computer. See "How the Term "PC" Is Used" on page xvii. DOS refers to IBM DOS Version 3.3 or later or MS-DOS** Version 3.3 or later.

Who Should Use This Book

This book is intended for application and system programmers with experience in writing application programs on a personal computer. You should also be familiar with the DOS operating system, and the C programming language. Knowledge of the TCP/IP protocols and standard TCP/IP user applications is also helpful. In this book, the term ***protocol*** is a set of rules for handling communication tasks.

If you are not familiar with TCP/IP concepts, see *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architectures* and *Internetworking With TCP/IP Volume II: Implementation and Internals*.

How to Use This Book

Before you start programming, verify that TCP/IP for DOS and the Programmer's Tool Kit is installed on your PC. For information about installing TCP/IP for DOS and the Programmer's Tool Kit, see *IBM TCP/IP Version 2.0 for DOS: Installation and Maintenance*.

How This Book Is Organized

Read the beginning section of each chapter to familiarize yourself with the topics that you need to know for application programming.

Chapter 1, "Introducing Computer Networks and Protocols," describes computer networks, an internet environment, and protocols supported by TCP/IP for DOS. Also included in this chapter is an overview of the routing and addressing schemes used by TCP/IP for DOS.

Chapter 2, "General Programming Information," contains fundamental, technical information about application program interfaces (API) provided with TCP/IP for DOS.

Chapter 3, "Sockets," describes the TCP/IP socket interface and how to use the socket routines in a user-written application.

Chapter 4, "Remote Procedure Calls (RPCs)," describes the remote procedure calls and how they are used in a user-written application.

Chapter 5, "File Transfer Protocol Application Programming Interface," describes the file transfer protocol routines and how they are used in a user-application.

Chapter 6, "Timer Routines," describes the use of timer routines in creating, setting, clearing, and removing timers.

Chapter 7, "Tasking Routines," describes the use of tasking routines to make a DOS system appear to run tasks simultaneously.

Appendix A, "Well-Known Port Assignments," provides the TCP and UDP well-known port numbers, and includes a description of the services provided with each port assignment.

Appendix B, "Sample Socket Programs," provides sample TCP and UDP client and server C socket communication programs.

Appendix C, "Sample RPC Programs," provides sample client and server RPC programs.

Appendix D, "Sample Tasking Program," provides sample tasking programs.

Appendix E, "Socket Quick Reference," describes each socket call supported by TCP/IP for DOS.

Appendix F, "Remote Procedure Call Quick Reference," describes each remote procedure call supported by TCP/IP for DOS.

Appendix G, "FTP API Quick Reference," describes each file transfer call supported by TCP/IP for DOS.

Appendix H, "Timer Quick Reference," describes each timer routine supported by TCP/IP for DOS.

Appendix I, "Tasking Quick Reference," describes each tasking routine supported by TCP/IP for DOS.

Appendix J, "NETWORKS File Structure," provides examples of network names contained in the NETWORKS file.

Appendix K, "Messages and Codes," provides a list of messages and codes for TCP/IP for DOS.

Appendix L, "Related Protocol Specifications," provides a listing of Requests for Comments (RFC), upon which many features of TCP/IP for DOS are based.

The book also includes a glossary, a bibliography, and an index.

For comments and suggestions about *IBM TCP/IP Version 2.0 for DOS: Programmer's Reference* use the Reader's Comment Form located at the back of this book. Use this form to give IBM information that might improve the book.

How the Term "internet" Is Used

In this book, an internet is a logical collection of networks supported by gateways, routers, hosts, and various layers of protocols that permit the network to function as a large, virtual network.

Note: The term "internet" is used as a generic term for a TCP/IP network, and should not be confused with the Internet, which consists of large national backbone networks (such as MILNET, NFSNet, and CREN) and a myriad of regional and local campus networks worldwide.

How the Term "PC" Is Used

In this book, PC refers to models of the IBM Personal System/2* (PS/2*), IBM Personal Computer XT* (PC/XT*), IBM Personal Computer AT* (PC AT*), and any other personal computer that is fully IBM compatible and can run DOS Version 3.30 or later.

How the Term <TCPBASE> Is Used

In this book, the generic term *<TCPBASE>* refers to the specific name of the base directory in which TCP/IP for DOS is installed. The default base directory for TCP/IP for DOS is C:\TCPDOS.

Coding Conventions Used in This Book

The following coding conventions are used throughout this book:

- Lowercase letters represent values that must be entered in lowercase.
- Lowercase italicized terms represent variable parameters where the user may supply the values.
- Uppercase letters represent commands and file names, which can be typed in either uppercase or lowercase.
- Periods in numbers separate the whole and the fractional portions of the numeral.

How to Read a Syntax Diagram

The syntax diagram shows you how to specify a command so that the operating system can correctly interpret what you type. Read the syntax diagram from left to right and from top to bottom, following the horizontal line (the main path).

The following symbols are used in syntax diagrams:

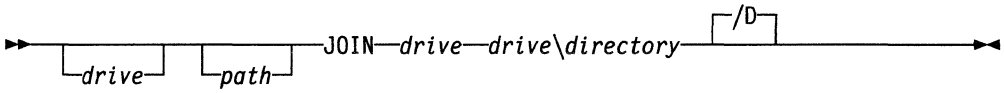
Symbol	Description
►►	Marks the beginning of the command syntax.
►	The command syntax is continued.
|	Marks the beginning and end of a fragment or part of the command syntax.
►◄	Marks the end of the command syntax.

Required parameters are displayed on the main path. Optional parameters are displayed below the main path. Default parameters are displayed above the main path.

Parameters are classified as keywords or variables. Keywords are displayed in uppercase letters and can be typed in uppercase or lowercase. A command is a keyword.

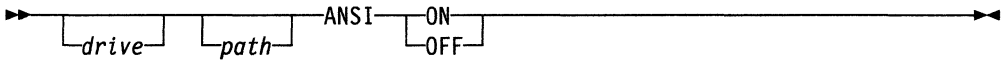
Variables are italicized, appear in lowercase letters, and represent names or values you supply. A file name is a variable.

In the following example, *drive*, *path*, and *drive\directory* are variable parameters. Replace them with the values you want.

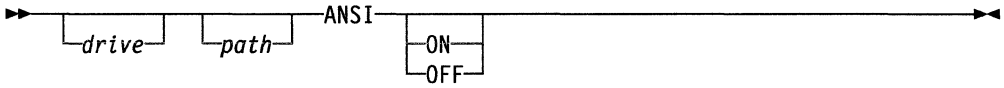


Include all punctuation such as colons, semicolons, commas, quotation marks, and minus signs shown in the diagram.

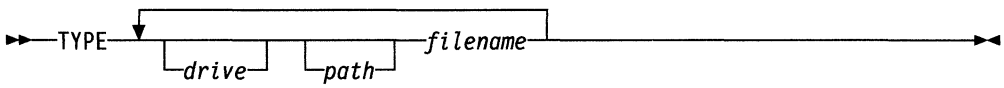
Choose One Required Item from a Stack: A stack of parameters with the first on the main path means that you must choose only one from the stack.



Choose One Optional Item from a Stack: A stack of parameters with the first below the main path means that you do not have to choose any from the stack, but if you do, you cannot choose more than one.



Specify a Sequence More Than Once: An arrow above the main path that returns to a previous point means the sequence of items included by the arrow can be specified more than once.



How Numbers Are Used in This Book

In this book, numbers over four digits are represented in metric style. A space is used rather than a comma to separate groups of three digits. For example, the number sixteen thousand, one hundred forty-seven is written 16 147.

Where to Find More Information

The following is a list of related publications that you might want to read for more information about TCP/IP for DOS:

- *IBM TCP/IP Version 2.0 for DOS: Installation and Maintenance*
- *IBM TCP/IP Version 2.0 for DOS: User's Guide*
- *Introducing IBM's TCP/IP Products for OS/2, VM, and MVS*
- *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architectures*
- *Internetworking With TCP/IP Volume II: Implementation and Internals.*

For more information about related publications, see the "Bibliography" at the back of this book.

Chapter 1. Introducing Computer Networks and Protocols

Chapter 1. Introducing Computer Networks and Protocols

This chapter introduces the concepts of computer networks and an internet environment. The protocols used by TCP/IP are listed by layer, and then described. Routing and addressing guidelines are also described.

Computer Networks

A computer network is a group of connected nodes that are used for data communication. A computer network configuration consists of data processing devices, software, and transmission media that are linked for information interchange.

Nodes are the functional units, located at the points of connection among the data circuits. A node, or end point, can be a host computer, a communication controller, a cluster controller, a video display terminal, or another peripheral device.

Computer networks can be local area networks (LANs), which provide direct communication among data stations on the user's local premises, or wide area networks (WANs), which provide communication services to a geographic area larger than that served by a LAN. Typically, WANs operate at a slower rate of speed than LANs.

Different types of networks provide different functions. Network configurations vary, depending on the functions required by the organization. Different organizations implement different types of networks. The technology used by these networks varies not only from organization to organization, but often varies within the same company.

Networks can differ at any or all layers. At the physical layer, networks can run over various network interfaces, such as token-ring, Ethernet**, and serial line. Networks can also vary as to the architectures they use to implement network strategies. Some of the more common architectures used today are OSI, TCP/IP, SNA, and ISDN. Networks use different protocols to communicate over the different physical interfaces available. In addition to these differences, networks can all use different software packages to implement various functions.

To exchange information among these different networks, the concept of an internet emerged.

Internet Environment

An internet is a logical collection of networks supported by gateways, routers, bridges, hosts, and various layers of protocols. An internet permits different physical networks to function as a single, large virtual network, and permits dissimilar computers to communicate with each other, regardless of their physical connections. Processes within gateways, routers, and hosts originate and receive packet information. Protocols specify a set of rules and formats required to exchange these packets of information.

Protocols are used to accomplish different tasks in TCP/IP software. To understand TCP/IP, you should be familiar with the following terms and relationships.

A ***client*** is a computer or process that requests services on the network. A ***server*** is a computer or process that responds to a request for service from a client. A ***user*** accesses a service, which allows the use of data or some other resource.

A ***datagram*** is the basic unit of information, consisting of one or more data packets that are passed across an internet at the transport level.

A ***gateway*** is a functional unit that connects two computer networks of different network architectures. A ***router*** is a device that connects networks at the ISO Network Layer. A router is protocol-dependent and connects only networks operating the same protocol. Routers do more than transmit data; they also select the best transmission paths and optimum sizes for packets. A ***bridge*** is a router that connects two or more networks and forwards packets among them. The operations carried out by a bridge are done at the physical layer and are transparent to TCP/IP and TCP/IP routing.

A ***host*** is a computer, connected to a network, which provides an access point to that network. A host can be a client, a server, or a client and server simultaneously. In a communication network, computers are both the sources and destinations of the packets. The ***local host*** is the computer to which a user's terminal is directly connected without the use of an internet, such as a PC running TCP/IP. A ***foreign host*** is any host on the network including the local host. A ***remote host*** is any foreign host not including the local host. A host is identified by its internet address.

An ***internet address*** is a unique 32-bit address identifying each node in an internet. An internet address consists of a network number and a local address in dotted-decimal notation. Internet addresses are used to route packets through the network.

Mapping relates internet addresses to physical hardware addresses in the network. For example, the Address Resolution Protocol (ARP) is used to map internet addresses to token-ring or Ethernet physical hardware addresses.

A ***network*** is the combination of two or more nodes and the connecting branches among them. A ***physical network*** is the hardware that makes up a network. A ***logical network*** is the abstract organization overlaid on one or more physical networks. An internet is an example of a logical network.

Packet refers to the unit or block of data of one transaction between a host and its network. A packet usually contains a network header, at least one high-level protocol header, and data blocks. Generally, the format of the data blocks does not affect how packets are handled. Packets are the exchange medium used at the internetwork layer to send and receive data through the network.

A ***port*** is an end point for communication between applications, generally referring to a logical connection. A port provides queues for sending and receiving data. Each port has a port number for identification. When the port number is combined with an internet address, a ***socket*** address results.

Protocol refers to a set of rules for achieving communication on a network.

TCP/IP Protocols and Functions

This section categorizes the TCP/IP protocols and functions by their functional group (network layer, internetwork layer, transport layer, and application layer). Figure 1 shows the relationship of these protocols and functions within the TCP/IP layered architecture.

- Network Layer
  - Serial Line Internet Protocol (SLIP)
- Internetwork Layer
  - Internet Protocol (IP)
  - Internet Control Message Protocol (ICMP)
  - Routing Information Protocol (RIP)
  - Address Resolution Protocol (ARP)
- Transport Layer
  - Transmission Control Protocol (TCP)
  - User Datagram Protocol (UDP)
- Application Layer
  - Telnet
  - File Transfer Protocol (FTP)
  - Trivial File Transfer Protocol (TFTP)
  - Simple Mail Transfer Protocol (SMTP)
  - Domain Name System (DNS)
  - Remote Printing (LPR)
  - RouteD
  - Network File System (NFS)
  - Remote Procedure Call (RPC)
  - Remote Execution Protocol (REXEC)
  - Post Office Protocol Version 2 (POP2)
  - Time Protocol (TIME)
  - Quote of the Day Protocol (COOKIE)
  - Finger Protocol (FINGER)
  - NICNAME/WHOIS Protocol
  - Socket Interfaces.

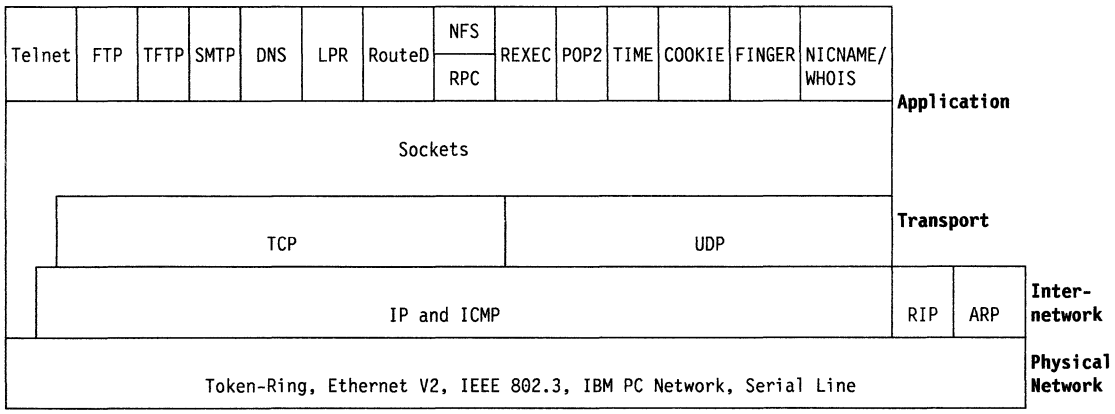


Figure 1. The TCP/IP Layered Architecture

Network Protocols

This section describes the protocols that compose the network layer available in TCP/IP for DOS. Network protocols define how data is transported over a physical network. These network protocols are not defined by TCP/IP. After a TCP/IP packet is created, a transport-dependent network header is added by the network protocol before the packet is sent out onto the network.

Serial Line Internet Protocol (SLIP)

In TCP/IP for DOS, the Serial Line Internet Protocol (SLIP) allows you to set up a point-to-point connection between two TCP/IP hosts over a serial line, such as a serial cable or an RS-232 connection using a modem and a telephone line. You can use SLIP to access a remote TCP/IP network from your local host, or to route datagrams between two TCP/IP networks.

Internetwork Protocols

Protocols in the internetwork layer provide connection services for TCP/IP. These protocols connect physical networks and transport protocols. This section describes the internetwork protocols in TCP/IP.

For information about TCP/IP in general, see RFCs 1118, 1180, 1206, 1207, and 1208. See Appendix L, "Related Protocol Specifications" for a list of related RFCs.

Internet Protocol (IP)

Internet Protocol (IP) provides the interface from the transport level (host-to-host, TCP, or UDP) protocols to the physical-level protocols. IP is the basic transport mechanism for routing IP packets to the next gateway, router, or destination host.

IP provides the means to transmit blocks of data (or packets) from sources to destinations. Sources and destinations are hosts identified by internet addresses. Outgoing packets automatically have an IP header prefixed to them, and incoming packets have their IP header removed before being sent to the higher-level protocols. This protocol provides the universal addressing of hosts in an internet network.

IP does not ensure a reliable communication, because it does not require acknowledgments from the sending host, the receiving host, or intermediate hosts. IP does not provide error control for data; it provides only a header checksum. IP treats each packet as an independent entity unrelated to any other packet. IP does not perform retransmissions or flow control. A higher-level protocol that uses IP must implement its own reliability procedures.

For more information about IP, see RFC 791.

Internet Control Message Protocol (ICMP)

Internet Control Message Protocol (ICMP) passes control messages between gateways, routers, and hosts. For example, ICMP messages can be sent in any of the following situations:

- When a host checks to see if another host is available (PING).
- When a packet cannot reach its destination.
- When a gateway or router can direct a host to send traffic on a shorter route.

- When a host requests a netmask or a time stamp.
- When a gateway or router does not have the buffering capacity to forward a packet.

ICMP provides feedback about problems in the communication environment; it does not make IP reliable. ICMP does not guarantee that an IP packet will be delivered reliably or that an ICMP message will be returned to the source host when an IP packet is not delivered or is incorrectly delivered.

For more information about ICMP, see RFC 792.

Routing Information Protocol (RIP)

Routing Information Protocol (RIP) is used by gateways, routers, and hosts to exchange routing information. This information can be used to maintain routing table entries.

For more information about RIP, see RFC 1058.

Address Resolution Protocol (ARP)

Address Resolution Protocol (ARP) maps internet addresses to hardware addresses. TCP/IP uses ARP to collect and distribute the information for mapping tables.

ARP is not directly available to users or applications. When an application sends an internet packet, IP requests the appropriate address mapping. If the mapping is not in the mapping table, an ARP broadcast packet is sent to all hosts on the network requesting the physical hardware address for the host.

For more information about ARP, see RFC 826.

Transport Protocols

The transport layer of TCP/IP consists of transport protocols, which allow communication between application programs. This section describes the transport protocols in TCP/IP.

Transmission Control Protocol (TCP)

The Transmission Control Protocol (TCP) provides a reliable vehicle for delivering packets between hosts on an internet. TCP takes a stream of data, breaks it into datagrams, sends each one individually using IP, and reassembles the datagrams at the destination node. If any datagrams are lost or damaged during transmission, TCP detects this fact and resends the missing datagrams. The received data stream is a reliable copy of the transmitted data stream.

For more information about TCP, see RFC 793.

User Datagram Protocol (UDP)

User Datagram Protocol (UDP) provides an unreliable mode of communication between source and destination hosts. UDP is built upon the service of the IP protocol in the internetwork layer. UDP provides a procedure for application programs to send data to other programs with a minimum of protocol overhead.

Like IP, UDP does not offer reliable datagram delivery or duplication protection. UDP does provide checksums for both the header and data portions of a datagram. However, applications that require reliable delivery of streams of data should use TCP.

For more information about UDP, see RFC 768.

Applications, Functions, and Protocols

Applications are provided with TCP/IP for DOS that allow users to use network services. These applications are included in the application layer of TCP/IP. The application layer is built upon the services of the transport layer. This section describes the applications, functions, and protocols in TCP/IP.

Telnet Protocol

Telnet Protocol provides a standard method to interface terminal devices and terminal-oriented processes with each other. Telnet is built upon the services of TCP in the transport layer. Telnet provides duplex communication and sends data either as ASCII characters or as binary data.

Telnet is commonly used to establish a logon session on a foreign host. Telnet can also be used for terminal-to-terminal communication and interprocess communication.

For more information about Telnet, see RFCs 854, 856, 857, 885, and 1091.

File Transfer Protocol (FTP)

File Transfer Protocol (FTP) makes it possible to transfer data between local and foreign hosts or between two foreign hosts. FTP is built upon the services of TCP in the transport layer. FTP transfers files as either ASCII characters or as binary data.

FTP provides functions such as listing remote directories, changing the current remote directory, creating and removing remote directories, and transferring one or more files in a single request. Security is handled by passing user IDs and account passwords to the foreign host.

For more information about FTP, see RFC 959.

Trivial File Transfer Protocol (TFTP)

Trivial File Transfer Protocol (TFTP) is designed only to read and write files to and from a foreign host. TFTP is built upon the services of UDP in the transport layer. TFTP allows you to limit drive and directory access.

TFTP, like FTP, can transfer files as either ASCII characters or as binary data. However, unlike FTP, TFTP cannot be used to list or change directories at a foreign host, and it has no provisions for user authentication.

For more information about TFTP, see RFC 783.

Simple Mail Transfer Protocol (SMTP)

Simple Mail Transfer Protocol (SMTP) is an electronic mail protocol with both client (sender) and server (receiver) functions.

You do not interface directly with SMTP. Instead, electronic mail software is used to create mail, which in turn uses SMTP to send the mail to its destination.

TCP/IP for DOS provides an SMTP client for sending mail to SMTP servers. TCP/IP for DOS does not have an SMTP server; the Post Office Protocol Version 2 (POP2) is used for receiving mail.

For more information about SMTP, see RFCs 821, 822, and 974. For more information about POP2, see RFC 937.

Domain Name System (DNS)

Domain Name System (DNS) uses a hierarchical-naming system for naming hosts. Each host name is composed of domain labels separated by periods. Local network administrators have the authority to name local domains within an internet. Each label represents an increasingly higher-domain level within an internet. The fully qualified domain name of a host connected to one of the larger internets generally has one or more subdomains. For example:

host.subdomain.subdomain.rootdomain
or
host.subdomain.rootdomain

Domain names often reflect the hierarchy level used by network administrators to assign domain names. For example, the domain name `eng.mit.edu` is the lowest level domain name, which is a subdomain of `mit.edu`. The subdomain `mit.edu` is a subdomain of edu.

Figure 2 is an example of the DNS used in the hierarchy naming structure across an internet.

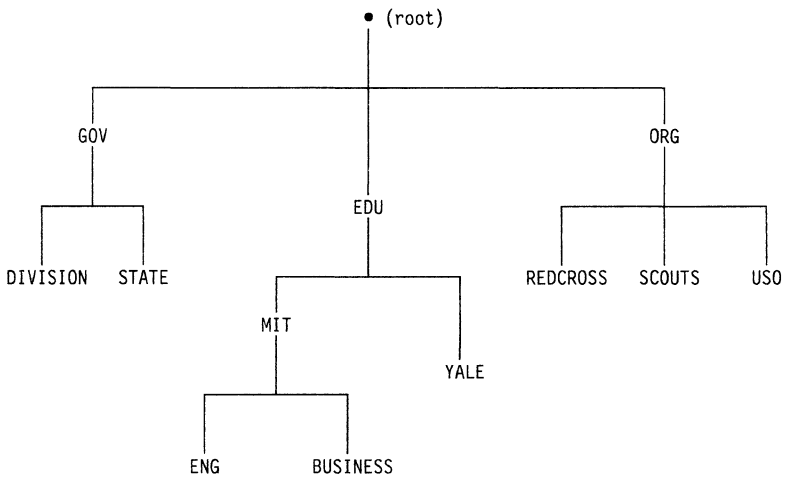


Figure 2. Hierarchical Tree

You may refer to hosts in your domain by host name only; however, a name server requires a fully qualified domain name. The local resolver combines the host name with the domain name before sending the address resolution request to the domain name server.

TCP/IP for DOS uses the local resolver functions of a local name resolution file. This file, called HOSTS, resides in the <*TCPBASE*>\ETC directory and contains entries that allow you to map symbolic names to internet addresses. If a name server is defined using the CUSTOM command, the resolver sends the request to the name server before using the local HOSTS file.

When using the HOSTS file on a small internet, it is not necessary to use the hierarchical-naming system used by the larger internets. The following example is a token-ring network of three users and their entries in the HOSTS file.

```
129.5.24.1 Host1 vjsPC PC1 mathdept
129.5.24.3 PC3 kensPC Host3 # This is Ken's PC
129.5.24.4 PC4 bobsPC
```

A carriage return must be entered at the end of each line.

In this example, each time the user enters the host name of `Host1` or the aliases `vjsPC`, `PC1`, or `mathdept`, the local name resolver translates it to the internet address of `129.5.24.1`. For more information about the format of network addresses, see "Network Address Format" on page 13.

For more information about DNS, see RFCs 1034 and 1035.

Remote Printing (LPR)

TCP/IP for DOS provides client support for remote printing. This application allows you to spool files remotely to a line printer daemon (LPD).

For more information about LPR, see RFC 1179.

RouteD

RouteD uses the Routing Information Protocol (RIP) to dynamically create and maintain network routing tables. The RIP protocol arranges to have gateways and routers periodically broadcast their routing tables to neighbors. Using this information, a RouteD server can update a host's routing tables. For example, RouteD determines if a new route has been created, if a route is temporarily unavailable, or if a more efficient route exists.

For more information about RouteD, see RFC 1058.

Network File System (NFS)

The Network File System (NFS) client allows you to manipulate files on remote TCP/IP hosts as if they reside on your local host. NFS is based on the NFS protocol, and uses the Remote Procedure Call (RPC) protocol to communicate between the client and the server. The files to be accessed reside on the server host, and are made available to the user on the client host.

NFS supports a hierarchical file structure. The directory and subdirectory structure can be different for individual client systems.

For more information about NFS, see RFC 1094.

Remote Procedure Call (RPC)

The Remote Procedure Call Protocol (RPC) is a programming interface that allows programs to execute subroutines on a foreign host. RPCs are high-level program calls, which can be used in place of the lower-level calls that are based on sockets.

For more information about RPC, see RFC 1057.

Remote Execution Protocol (REXEC)

Remote Execution Protocol allows you to execute a command on a foreign host and receive the results on the local host. Remote Execution Protocol provides automatic logon and user authentication depending on the parameters set by the user.

Post Office Protocol Version 2 (POP2)

The Post Office Protocol Version 2 (POP2) allows you to access electronic mail from a remote mailbox server. Mail should be posted from hosts to the mailbox server using SMTP.

For more information about POP2, see RFC 937.

Time Protocol (TIME)

The Time Protocol (TIME) provides a site-independent, machine-readable date and time. TIME can use either UDP or TCP as the underlying protocol. The TIME server returns the number of seconds since midnight on January 1, 1900 Universal Time (UT).

For more information about TIME, see RFC 868.

Quote of the Day Protocol (COOKIE)

The Quote of the Day Protocol (COOKIE) retrieves thoughts for the day from a network quote server. When a packet is sent to a COOKIE server, COOKIE returns a message and discards any data contained in the packet. COOKIE can use either UDP or TCP as the underlying protocol. There is no specific syntax for the message returned by COOKIE.

For more information about COOKIE, see RFC 865.

Finger Protocol (FINGER)

The Finger Protocol (FINGER) provides an interface for querying the current status of a remote host or a user ID on a remote host. FINGER uses TCP as the underlying protocol.

For more information about FINGER, see RFC 1196.

NICNAME/WHOIS Protocol

The NICNAME/WHOIS Protocol provides an interface to the NICNAME/WHOIS directory service at the Network Information Center, `nic.ddn.mil`, and to other NICNAME/WHOIS servers on the Internet. NICNAME/WHOIS uses TCP as the underlying protocol.

For more information about NICNAME/WHOIS, see RFC 954.

Socket Interfaces

Socket interfaces allow users to write their own applications to supplement those supplied by TCP/IP for DOS. Most of these additional applications communicate with either TCP or UDP. Some applications are written to communicate directly with IP. To write applications that use the socket interfaces of TCP/IP for DOS, you must be able to compile and link the programs using the Microsoft C** compiler, Version 5.10 or later.

Sockets are duplex, which means that data can be transmitted and received simultaneously. Sockets allow you to send to, and receive from, the socket as if you are writing to and reading from any other network device.

Routing

The routing functions in an internet are performed at the internetwork layer. Routing is the process of deciding where to send a packet based on its destination address. Two kinds of routing are involved in communication within an internet: direct and indirect.

Direct routing is used when the source and destination nodes are on the same logical network within an internet. The source node maps the destination internet address into a hardware address and sends packets to the destination node using this address. This mapping is normally performed through a translation table. If a match cannot be found for a destination internet address, ARP is invoked to determine this address.

Indirect routing is used when the source and destination nodes are on different logical networks within an internet. The source node sends packets to a gateway or router on the same network using direct routing. From there, the packets are forwarded through intermediate gateways or routers, as required, until they arrive at the destination network. Direct routing is then used to forward the packets to the destination host on that network. Each gateway, router, and host in an internet has a routing table that defines the address of the next gateway to other networks (as well as other nodes on other networks) in an internet.

Internet Addressing

Each internet host is assigned at least one unique internet address. This address is used by IP and other higher-level protocols. When gateway hosts are used, more than one address may be required. Each interface to an internet is assigned its own unique address. Internet addresses are used to route packets through the network.

Addresses within an internet consist of a network number and a local address. A unique network number is assigned to each network when it connects to another internet. If a local network is not going to connect to other internets, any convenient network number is assigned. Some networks are divided into subnets. For more information about subnets, see "Subnetwork Address Format" on page 14.

Hosts that exchange packets on the same physical network should have the same network number. Hosts on different physical networks might also have the same network number. If hosts have the same network number, part of the local address is used as a subnetwork number. All host interfaces to the same physical network are given the same subnetwork number.

An internet can provide standards for assigning addresses to networks, broadcasts, and subnetworks. Examples of these standard formats are described in the following sections.

Network Address Format

A standard internet address uses a two-part, 32-bit address field. The first part of the address field contains the network address; the second part contains the local address. The four different types of address fields are classified as A, B, C, or D, depending on the bit allocation.

Figure 3 represents a class A address. Class A addresses have a 7-bit network number and a 24-bit local address. The highest order bit is set to 0.

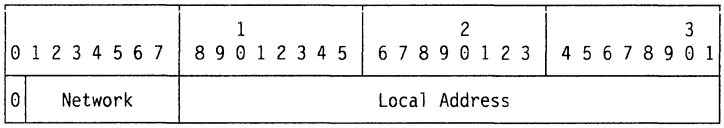


Figure 3. Class A Address

Figure 4 represents a class B address. Class B addresses have a 14-bit network number and a 16-bit local address with the highest order bits set to 10.

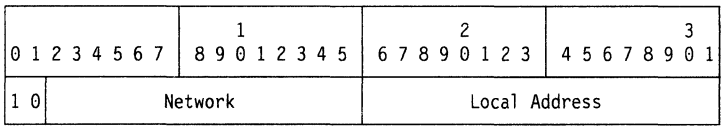


Figure 4. Class B Address

Figure 5 represents a class C address. Class C addresses have a 21-bit network number and an 8-bit local address with the three highest order bits set to 110.

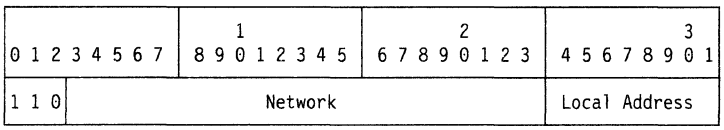


Figure 5. Class C Address

Figure 6 represents a class D address. Class D networks have a multicast address that is sent to selected hosts on the network. The four highest order bits are set to 1110.

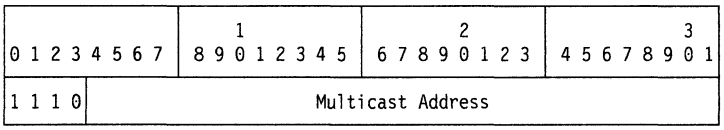


Figure 6. Class D Address

Note: Class D addresses are not supported in TCP/IP for DOS.

A commonly used notation for internet host addresses is the dotted-decimal, which divides the 32-bit address into four 8-bit fields. The value of each field is specified as a decimal number, and the fields are separated by periods (for example, 010.002.000.052 or 10.2.0.52).

Address examples in this book use dotted-decimal notation in the following forms:

Class A *nnn.lll.lll.lll*
Class B *nnn.nnn.lll.lll*
Class C *nnn.nnn.nnn.lll*

where *nnn* represents part or all of a network number and *lll* represents part or all of a local address.

Broadcast Address Format

TCP/IP uses IP broadcasting to send datagrams to all the TCP/IP hosts on a network or subnetwork. A datagram sent to the broadcast address is received by all of the hosts on the network and processed as if the datagram was sent directly to the host's IP address. The IP broadcast address is formed by setting all of the host bits to ones.

For more information about broadcast address format, see RFCs 919 and 922.

Subnetwork Address Format

The subnetwork capability of TCP/IP divides a single network into multiple logical networks (subnets). For instance, an organization can have a single internet network address that is known to users outside the organization, yet configure its internal network into different departmental subnets. Subnetwork addresses enhance local routing capabilities, while reducing the number of network numbers required.

For a subnet, the local address part of an internet address is divided into a subnet number and a host number, for example:

network_number subnet_number host_number

where:

network_number Is the network portion of the internet address.
subnet_number Is a field of a constant width for a given network.
host_number Is a field that is at least 1-bit wide.

If the width of the *subnet_number* field is 0, the network is not organized into subnets, and addressing to the network is done with an internet network address (*network_number*).

Figure 7 represents a class B address with a 6-bit wide subnet field.

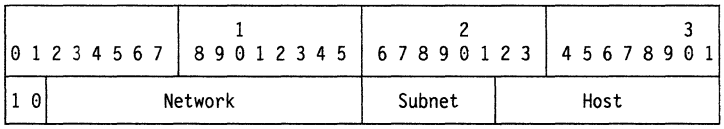


Figure 7. Class B Address with Subnet

The bits that identify the subnet are specified by a bit mask. A bit mask is a pattern of binary digits used to assign subnet addresses. The subnet bits are not required to be adjacent in the address. However, the subnet bits generally are contiguous and located as the most significant bits of the local address.

For more information about subnetwork address format, see RFC 950.

Chapter 2. General Programming Information

Chapter 2. General Programming Information

This chapter contains technical information that you need to know before you attempt to work with the application programming interfaces (API) provided with TCP/IP for DOS, and described in this book.

You should have installed TCP/IP for DOS and the application programming interfaces (APIs) in the *<TCPBASE>* directory.

TCP/IP for DOS Component Interfaces

Figure 8 shows the relationship between the major components of TCP/IP for DOS. The timer and task routines are the interface between TCP/IP applications and UTIL, the utility terminate and stay resident (TSR) program. The socket routines are the interface between TCP/IP applications and INET or RIPINET TSRs. UTIL, INET, or RIPINET, and the hardware TSRs communicate with one another. However, only the hardware TSRs communicate with hardware device drivers, such as NDIS**, MAC Drivers, and packet device drivers.

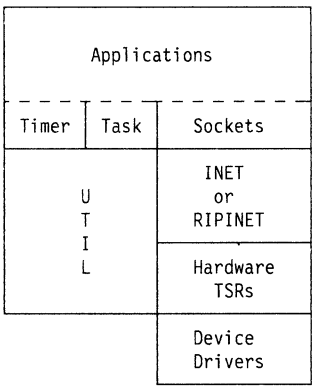


Figure 8. TCP/IP for DOS Architecture

Header Files

This section lists the header files for each API. These files are in the *<TCPBASE>*\INCLUDE directory.

Sockets

The following is a list of socket application header files.

- TCPERRNO.H
- NETDB.H
- NETINET\IN.H
- SYS\SOCKET.H
- SYS\TIME.H
- TYPES.H

Remote Procedure Calls (RPCs)

The following is a list of RPC application header files.

- RPC\AUTH.H
- RPC\A_UNIX.H
- RPC\CLNT.H
- RPC\P_CLNT.H
- RPC\P_PROT.H
- RPC\P_RMT.H
- RPC\RPC.H
- RPC\R_MSG.H
- RPC\TYPES.H
- RPC\SVC.H
- RPC\SVC_AUTH.H
- RPC\XDR.H

File Transfer Protocol Application Programming Interface (FTP API)

The following is a list of FTP API application header files.

- FTPAPI.H

Library Files

The following is a list of library files to which an application must link.

Library File	Application
FTPAPI.LIB	FTP Application Programming Interface (API) calls
SUNRPC.LIB	Sun** Remote Procedure calls
TCPIP.LIB	Socket calls

Porting Considerations

This section contains information about how to port your application.

- To access system return values, you only need to use the `errno.h` include statement supplied with the compiler.
- To access network return values, you must add the following include statement:

```
#include <tcperrno.h>
```

For more information about porting, see the respective chapter for that interface.

Chapter 3. Sockets

Chapter 3. Sockets

This chapter describes the socket application program interface (API) provided with TCP/IP for DOS. Use the socket routines to interface with the TCP, UDP, ICMP, and IP protocols. This allows a program to communicate across networks with other programs. You can, for example, make use of socket routines when you write a client program that must communicate with a server program running on another computer.

To use the sockets, you must know C programming language. For more information about sockets, see *IBM AIX Version 3 for RISC/6000 Communications Programming Concepts*.

Programming with Sockets

The DOS socket API provides a standard interface to the transport and internetwork layer interfaces of TCP/IP. It supports two socket types: stream and datagram. Stream and datagram sockets interface to the transport layer protocols. You choose the most appropriate interface for an application.

Socket Programming Concepts

Before programming with the sockets API, it is helpful to consider some important concepts.

What is a Socket?

A socket is an endpoint for communication that can be named and addressed in a network. From an application program perspective, it is a resource allocated by the operating system. It is represented by an integer called a socket descriptor.

The socket interface was designed to provide applications with a network interface that hides the details of the physical network. The interface is differentiated by the different services that are provided. Stream and datagram sockets define a different service available to applications.

Socket Types

The stream socket (SOCK_STREAM) interface defines a reliable connection-oriented service. Data is sent without errors or duplication and is received in the same order as it is sent. Flow control is built in to avoid data overruns. No boundaries are imposed on the data; it is considered to be a stream of bytes. An example of an application that uses stream sockets is the File Transfer Protocol (FTP).

The datagram socket (SOCK_DGRAM) interface defines a connectionless service. Datagrams are sent as independent packets. The service provides no guarantees; data can be lost or duplicated, and datagrams can arrive out of order. The size of a datagram is limited to the size that can be sent in a single transaction (currently the default is 8192 and the maximum is 32 768). No disassembly and reassembly of packets is performed. An example of an application that uses datagram sockets is the Network File System (NFS).

The socket interface can be extended; therefore, it is possible to define new socket types to provide additional services. An example of this is the transaction type sockets defined for interfacing to the Versatile Message Transfer Protocol (VMTP). Transaction type sockets are not supported by TCP/IP for DOS. Because socket

interfaces isolate you from the communication functions of the different protocol layers, the interfaces are largely independent of the underlying network. In the DOS implementation of sockets, stream sockets interface to TCP and datagram sockets interface to UDP. In the future, the underlying protocols may change, but the socket interface will remain the same. For example, stream sockets may eventually interface to the International Standards Organization (ISO) Open System Interconnection (OSI) transport class 4 protocol. This means that applications will not have to be rewritten as underlying protocols change.

Guidelines for Using Socket Types

The following considerations help you choose the appropriate socket type for an application.

If you are communicating to an existing application, you must use the same protocols as the existing application. For example, if you interface to an application that uses TCP, you must use stream sockets. For other applications you should consider the following factors:

- Consider reliability. Stream sockets provide the most reliable connection. Datagram sockets are unreliable because packets can be discarded, corrupted, or duplicated during transmission. This may be acceptable if the application does not require reliability, or if the application implements the reliability on top of the sockets interface. The tradeoff is the increased performance available over stream sockets.
- Performance is another consideration. The overhead associated with reliability, flow control, packet reassembly, and connection maintenance degrade the performance of stream sockets so that they do not perform as well as datagram sockets.
- The amount of data to be transferred is another consideration. Datagram sockets impose a limit on the amount of data transferred in a single transaction. If you send less than 2048 bytes at a time, use datagram sockets. As the amount of data in a single transaction increases, it makes more sense to use stream sockets.

Address Families

Address families define different styles of addressing or communication domain. All hosts in the same addressing family understand and use the same scheme for addressing socket endpoints. TCP/IP for DOS supports one addressing family: AF_INET. The AF_INET domain defines addressing in the internet domain. AF_INET is also referred to as a PF_INET. Both are equivalent. PF stands for Protocol Family. The address families are defined in the <SYS\SOCKET.H> header file.

Socket Address

A socket address is defined by the *sockaddr* structure in the <SYS\SOCKET.H> header file. It has two fields, as shown in the following example:

```
struct sockaddr
{
    u_short sa_family;      /* address family */
    char  sa_data[14];      /* up to 14 bytes of direct address */
};
```

The *sa_family* field contains the addressing family. It is AF_INET for the internet domain. The *sa_data* field is different for each address family. Each address family defines its own structure, which can be overlaid on the *sockaddr* structure.

Addressing within an Internet Domain: A socket address in an internet addressing family comprises four fields: the address family (AF_INET), an internet address, a port, and a character array. The structure of an internet socket address is defined by the following *sockaddr_in* structure, which is found in the <NETINET\IN.H> header file:

```
struct in_addr
{
        u_long s_addr;
};

struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field is set to AF_INET. The *sin_port* field is the port used by the application, in network byte order. The *sin_addr* field is the internet address of the network interface used by the application. It is also in network byte order. The *sin_zero* field should be set to all zeros.

Internet Addresses

Internet addresses are 32-bit quantities that represent a network interface. Every internet address within an administered AF_INET domain must be unique. A common misunderstanding is that every host must have only one internet address. In fact, a host has as many internet addresses as it has network interfaces. For more information about internet address formats, see *Internetworking With TCP/IP Volume I: Principles, Protocols, and Architectures*, and *Internetworking With TCP/IP Volume II: Implementation and Internals*.

Ports

A port is used to differentiate between different applications using the same protocol (TCP or UDP). It is an additional qualifier used by the system software to get data to the correct application. Physically, a port is a 16-bit integer. Some ports are reserved for particular applications and are called well-known ports. For more information, see Appendix A, "Well-Known Port Assignments" or see the <*TCPBASE*>\ETC\SERVICES file.

Network Byte Order

Ports and addresses are usually specified to calls using the network byte ordering convention. Network byte order is also known as big endian byte ordering, as in Motorola ** microprocessors (compared with little endian byte ordering in Intel** microprocessors). Using network byte ordering for data exchanged between hosts allows hosts using different architectures to exchange address information. See pages 24, 25, and 27 for examples of using the htons() call to put ports into network byte order. For more information about network byte order, see: "accept()" on page 36, "bind()" on page 38, "htonl()" on page 67, "htons()" on page 68, "ntohl()" on page 76, and "ntohs()" on page 77.

Note: The sockets interface does not handle application data byte ordering differences. Application writers must handle byte order differences themselves or use higher-level interfaces, such as Remote Procedure Calls (RPC).

Main Socket Calls

With few socket calls, you can write a very powerful network application.

1. First, an application must be initialized with sockets using the sock_init() call, as in the example in Figure 9, or using the dosip_init()call. For a more detailed description, see "sock_init()" on page 91 or "dosip_init()" on page 44.

```
void sock_init();
:
sock_init();
```

Figure 9. An Application Uses the sock_init() Call

The code fragment in Figure 9 initializes the process with the socket library.

2. Next, an application must get a socket descriptor using the socket() call, as in the example in Figure 10. For a more detailed description, see "Socket Interfaces" on page 12.

```
int socket(int domain, int type, int protocol);
:
int s;
:
s = socket(AF_INET, SOCK_STREAM, 0);
```

Figure 10. An Application Uses the socket() Call

The code fragment in Figure 10 allocates a socket descriptor *s* in the internet addressing family. The *domain* parameter is a constant that specifies the domain where the communication is taking place. A domain is the collection of applications using the same naming convention. TCP/IP for DOS supports one addressing family: AF_INET. The *type* parameter is a constant that specifies the type of socket, SOCK_STREAM or SOCK_DGRAM. The *protocol* parameter is a constant that specifies the protocol to use. Passing 0 chooses the default protocol. If successful, socket() returns a positive integer socket descriptor.

3. Once an application has a socket descriptor, it can explicitly bind a unique name to the socket, as in the example in Figure 11. For a more detailed description, see "bind()" on page 38.

```
int rc;
int s;
struct sockaddr_in myname;
int bind(int s, struct sockaddr *name, int namelen);

/* clear the structure to be sure that the sin_zero field is clear */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
:
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

Figure 11. An Application Uses the bind() Call

This example binds *myname* to socket *s*. The name specifies that the application is in the internet domain (AF_INET) at internet address 129.5.24.1, and is bound to port 1024. Servers must bind a name to become accessible from the network. The example in Figure 11 shows two useful utility routines:

- inet_addr() takes an internet address in dotted decimal form and returns it in network byte order. For a more detailed description, see "inet_addr()" on page 69.
- htons() takes a port number in host byte order and returns the port in network byte order. For a more detailed description, see "htons()" on page 68.

```
int rc;
int s;
struct sockaddr myname;
int bind(int s, struct sockaddr name, int namelen);
struct servent *sp;
 :
sp = getservbyname("login","tcp");   /* get application specific */
                                     /* well-known port          */

/* clear the structure to be sure the sin_zero field is clear */
memset(&myname,0,sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY;
myname.sin_port = htons(sp->s_port);
 :
rc = bind(s,(struct sockaddr *)&myname,sizeof(myname));
```

Figure 12. A bind() Call Using the getservbyname() Call

Figure 12 shows another example of the bind() call on the server side. It uses the network utility routine getservbyname() to find a well-known port number for specific service from the <*TCPBASE*>\ETC\SERVICES file. Figure 12 also shows wildcard value INADDR_ANY. If a host has several network addresses (multi-homed host), it is likely that messages sent to any of the addresses should be deliverable to a socket.

4. After binding a name to a socket, a server using stream sockets must indicate its readiness to accept connections from clients. The server does this with the listen() call as illustrated in the example in Figure 13.

```
int s;
int backlog;
int rc;
int listen(int s, int backlog);
 :
rc = listen(s, 5);
```

Figure 13. An Application Uses the listen() Call

The listen() call tells the TCP/IP software that the server is ready to begin accepting connections and that a maximum of five connection requests can be queued for the server. Additional requests are ignored. For a more detailed description, see "listen()" on page 75.

5. Clients using stream sockets initiate a connection request by calling connect(), as shown in the example in Figure 14 on page 26.

```
int s;
struct sockaddr_in servername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);
:
memset(&servername, 0,sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1");
servername.sin_port = htons(1024);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
```

Figure 14. An Application Uses the connect() Call

The connect() call attempts to connect socket *s* to the server with name *servername*. This could be the server that was used in the previous bind() example. The caller optionally blocks until the connection is accepted by the server. On successful return, the socket *s* is associated with the connection to the server. For a more detailed description, see "connect()" on page 41.

```
int s;
struct sockaddr servername;
char *hostname = "serverhost";
int rc;
int connect(int s, struct sockaddr *name, int namelen);
struct servent *sp;
struct hostent *hp;
:
sp = getservbyname("login","tcp");  /* get application specific */
                                    /* well-known port          */

hp = gethostbyname(hostname);

/* clear the structure to be sure that the sin_zero field is clear */
memset(&servername,0,sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr.s_addr = *((u_long *)hp->h_addr);
servername.sin_port = htons(sp->s_port);
:
rc = connect(s,(struct sockaddr *)&servername,sizeof(servername));
```

Figure 15. An Application Uses the gethostbyname() Call

Figure 15 shows an example of a network utility routine gethostbyname() call to find out the internet address of *serverhost* from the name server or the <*TCPBASE*>\ETC\HOSTS file.

6. Servers using stream sockets accept a connection request with the accept() call, as shown in the example in Figure 16.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen);
:
addrlen = sizeof(clientaddress);
:
clientsocket = accept(s, &clientaddress, &addrlen);
```

Figure 16. An Application Uses the accept() Call

If connection requests are not pending on socket *s*, the accept() call optionally blocks the server. When a connection request is accepted on socket *s*, the name of the client and length of the client name are returned, along with a new socket descriptor. The new socket descriptor is associated with the client that initiated

the connection and *s* is again available to accept new connections. For a more detailed description, see "accept()" on page 36.

7. Clients and servers have many calls from which to choose for data transfer. The send() and recv() calls can be used only on sockets that are in the connected state. The sendto() and recvfrom() calls can be used at any time. The example in Figure 17 illustrates the use of send() and recv().

```
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
int send(int socket, char *buf, int buflen, int flags);
int recv(int socket, char *buf, int buflen, int flags);
int s;
:
bytes_sent = send(s, data_sent, sizeof(data_sent), 0);
:
bytes_received = recv(s, data_received, sizeof(data_received), 0);
```

Figure 17. An Application Uses the send() and recv() Calls

The example in Figure 17 shows an application sending data on a connected socket and receiving data in response. The *flags* field can be used to specify additional options to send() or recv(), such as sending out-of-band data.

8. If the socket is not in a connected state, additional address information must be passed to sendto() and may be optionally returned from recvfrom(). An example of the use of the sendto() and recvfrom() calls is in Figure 18.

```
int bytes_sent;
int bytes_received;
char data_sent[256];
char data_received[256];
struct sockaddr_in to;
struct sockaddr from;
int addrlen;
int sendto(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int recvfrom(int socket, char *buf, int buflen, int flags,
           struct sockaddr *addr, int addrlen);
int s;
:
to.sin_family = AF_INET;
to.sin_addr   = inet_addr("129.5.24.1");
to.sin_port   = htons(1024);
:
bytes_sent = sendto(s, data_sent, sizeof(data_sent), 0, &to, sizeof(to));
:
addrlen = sizeof(from); /* must be initialized */
bytes_received = recvfrom(s, data_received,
   sizeof(data_received), 0, &from, &addrlen);
```

Figure 18. An Application Uses the sendto() and recvfrom() Call

The sendto() and recvfrom() calls take additional parameters that allow the caller to specify the recipient of the data or to be notified of the sender of the data. See "recvfrom()" on page 79, and "sendto()" on page 83, for more information about these additional parameters. Usually, sendto() and recvfrom() are used for datagram sockets, and send() and recv() are used for stream sockets.

9. Applications can handle multiple sockets. In such situations, use the select() call to determine the sockets that have data to be read, those that are ready for data to be written, and the sockets that have pending exceptional conditions. An example of how the select() call is used, is in Figure 19 on page 28.

```
fd_set readsocks;
fd_set writesocks;
fd_set exceptsocks;
struct timeval timeout;
int number_of_sockets;
int number_found;
:
/* set bits in read write except bit masks.  To set mask for a descriptor s use
*   readsocks |= fd_set(s);
*
* set number of sockets to be checked
* number_of_sockets = x;
*/
:
number_found = select(number_of_sockets,
                      &readsocks, &writesocks, &exceptsocks, &timeout);
```

Figure 19. An Application Uses the select() Call

In this example, the application sets bit masks to indicate the sockets being tested for certain conditions and also indicates a time-out. If the timeout parameter is NULL, the call does not wait for any socket to become ready on these conditions. If the timeout parameter is nonzero, select() waits up to this amount of time for at least one socket to become ready on the indicated conditions. This is useful for applications servicing multiple connections that cannot afford to block, waiting for data on one connection. For a more detailed description, see "select()" on page 80.

10. A socket descriptor, s, is deallocated with the so_close() call. For a more detailed description, see "so_close()" on page 95. An example of the so_close() call is shown in Figure 20.

```
:
/* close the socket */
so_close(s);
:
```

Figure 20. An Application Uses the so_close() Call

A Typical TCP Socket Session

You can use TCP sockets for both passive (server) and active (client) processes. While some commands are necessary for both types, some are role-specific. See Appendix B, "Sample Socket Programs," for sample socket communication client and server programs.

Once you make a connection, it exists until you close the socket. During the connection, data is either delivered or an error code is returned by TCP/IP.

See Figure 21 on page 30 for the general sequence of calls to be followed for most socket routines using TCP sockets.

A Typical UDP Socket Session

UDP socket processes, unlike TCP socket processes, are not clearly distinguished by server and client roles. Instead, the distinction is between connected and unconnected sockets. An unconnected socket can be used to communicate with any host; but a connected socket, because it has a dedicated destination, can send data to, and receive data from, only one host.

Both connected and unconnected sockets send their data over the network without verification. Consequently, once a packet has been accepted by the UDP interface, the arrival of the packet and the integrity of the packet cannot be guaranteed.

See Figure 22 on page 31 for the general sequence of calls to be followed for most socket routines using UDP sockets.

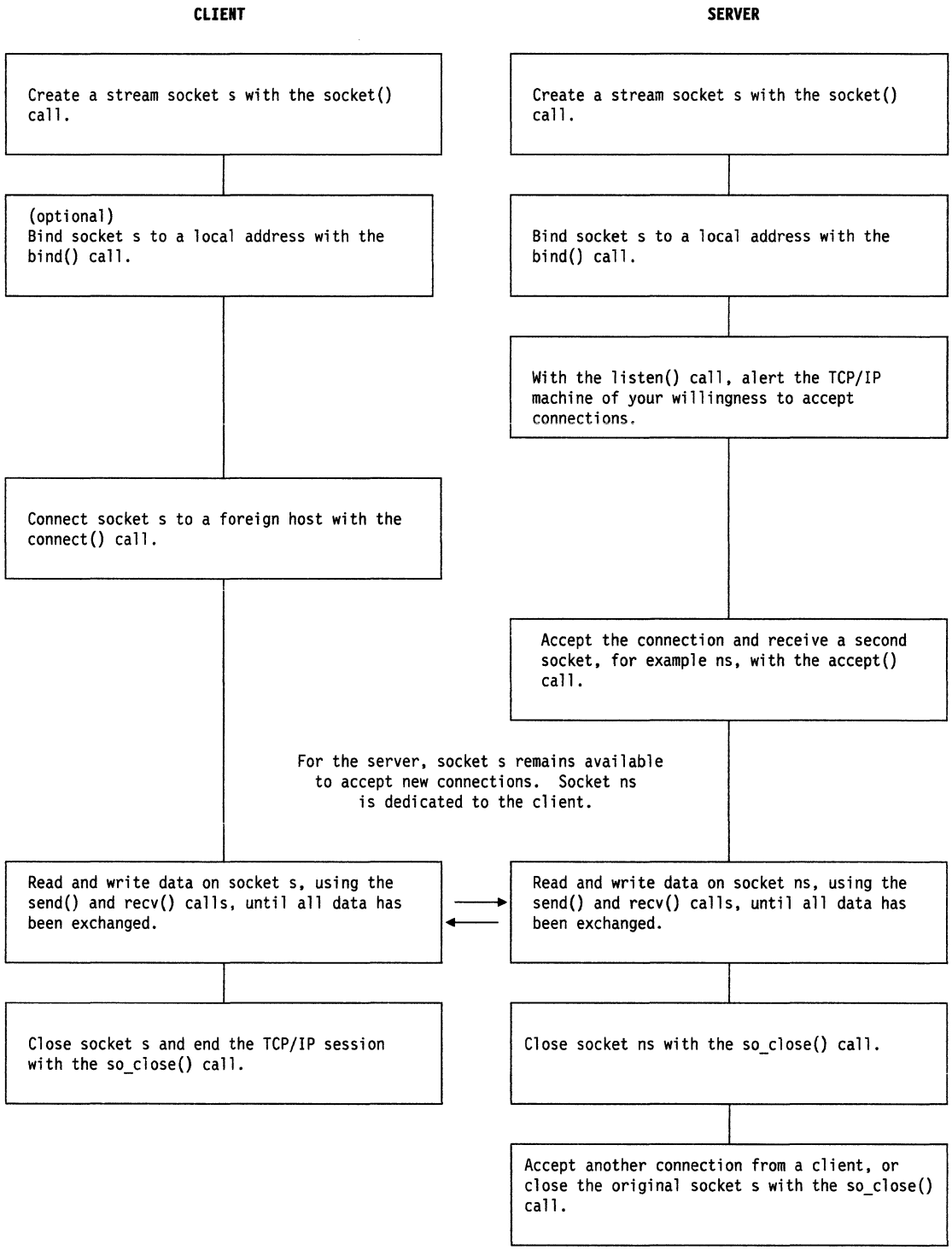


Figure 21. A Typical TCP Socket Session

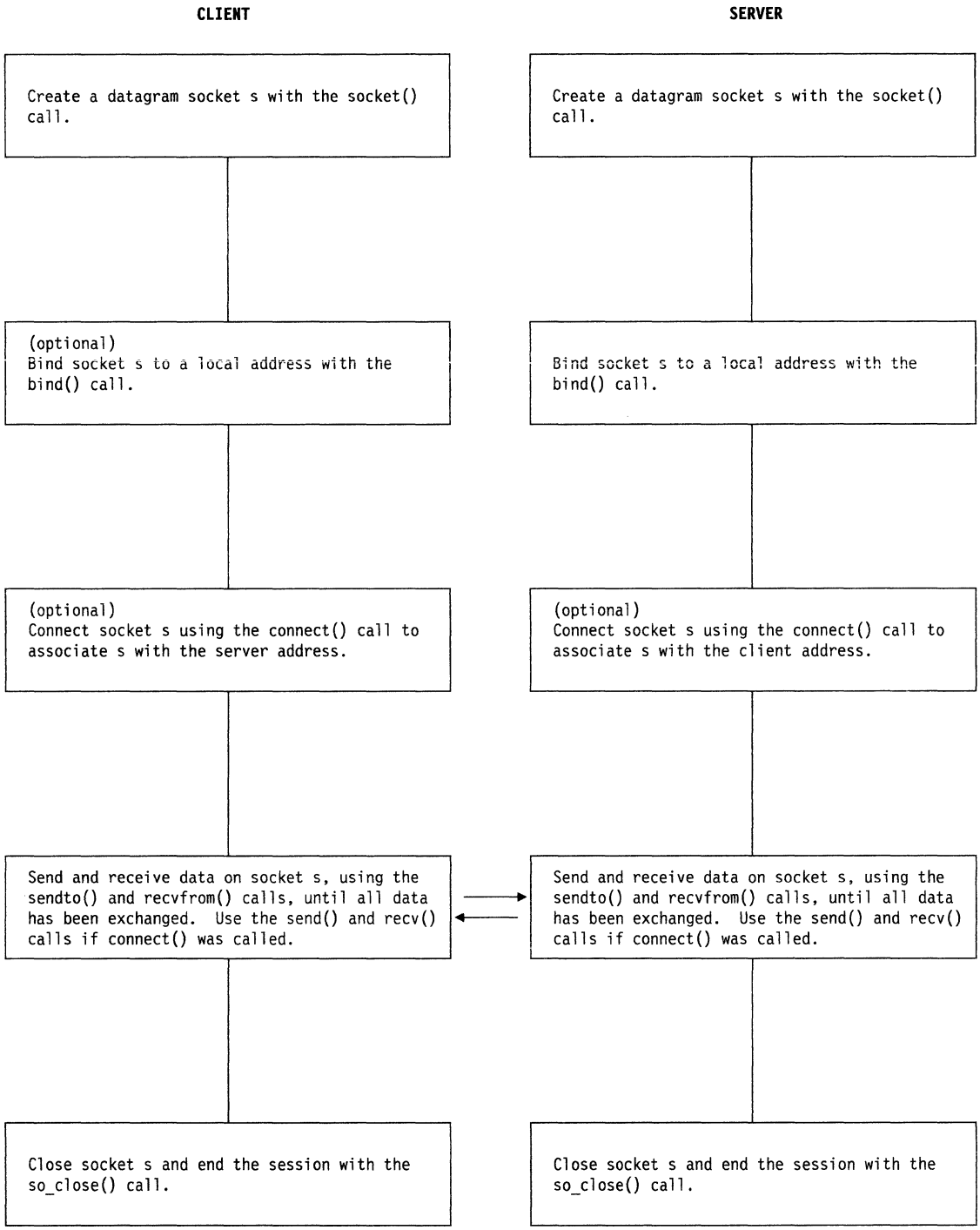


Figure 22. A Typical UDP Socket Session

Network Utility Routines

The DOS socket API also provides a set of network utility routines to perform useful tasks, such as internet address translation, domain name resolution, network byte order translation, and access to the database of useful network information. This section describes a few network utility routines.

Host Names Information: The following is a list of socket calls that provide host name information:

- gethostbyaddr()
- gethostbyname()
- sethostent()
- gethostent()
- endhostent().

The gethostbyname() call takes an internet host name and returns a *hostent* structure, which contains the name of the host, aliases, host address family, and host address. The *hostent* structure is defined in the <NETDB.H> header file. The call gethostbyaddr() maps the internet host address into a *hostent* structure.

The database for these calls is provided by the name server or <*TCPBASE*>\ETC\HOSTS file if a name server is not present. Because of the differences in the databases and their access protocols, the information returned may differ.

The sethostent(), gethostent() and endhostent() calls provide sequential access to the <*TCPBASE*>\ETC\HOSTS file.

Network Names Information: The following is a list of socket calls that provide network names information:

- getnetbyaddr()
- getnetbyname()
- setnetent()
- getnetent()
- endnetent().

The getnetbyname() call takes a network name and returns a *netent* structure, which contains the name of the network, aliases, network address family, and network number. The *netent* structure is defined in the <NETDB.H> header file. The getnetbyaddr() call maps the network number into a *netent* structure.

The database for these calls is provided by the <*TCPBASE*>\ETC\NETWORKS file.

The setnetent(), getnetent(), and endnetent() calls provide sequential access to the <*TCPBASE*>\ETC\NETWORKS file.

Protocol Names Information: The following is a list of socket calls that provide protocol names information:

- getprototbynumber()
- getprototbyname()
- setprotoent()
- getprotoent()
- endprotoent().

The getprotobyname() call takes the protocol name and returns a *protoent* structure, which contains the name of the protocol, aliases, and protocol number. The

protoent structure is defined in the <NETDB.H> header file. The getprotobynumber() call maps the protocol number into a *protoent* structure.

The database for these calls is provided by the <*TCPBASE*>\ETC\PROTOCOL file.

The setprotoent(), getprotoent(), and endprotoent() calls provide sequential access to the <*TCPBASE*>\ETC\PROTOCOL file.

Service Names Information: The following is a list of socket calls that provide service names information:

- getservbyname()
- getservtbyport()
- setservent()
- getservent()
- endservent().

The getservbyname() call takes the service name and protocol, and returns a *servent* structure, which contains the name of the service, aliases, port number, and protocol. The *servent* structure is defined in the <NETDB.H> header file. The getservbyport() call maps the port number and protocol into a *servent* structure.

The database for these calls is provided by <*TCPBASE*>\ETC\SERVICES file.

The setservent(), getservent(), and endservent() calls provide sequential access to the <*TCPBASE*>\ETC\SERVICES file.

Network Byte Order Translation: Ports and addresses are usually specified to calls using the network byte ordering convention. The following calls translate integers from network to host byte order and from host to network byte order.

Call	Function
htonl()	Translates host to network, long integer (32-bit)
htons()	Translates host to network, short integer (16-bit)
ntohl()	Translates network to host, long integer (32-bit)
ntohs()	Translates network to host, short integer (16-bit).

Internet Address Manipulation: The following calls convert internet addresses and decimal notation, and manipulate the network number and local network address portions of an internet address.

Call	Function
inet_addr()	Translates dotted decimal notation to a 32-bit internet address (network byte order).
inet_network()	Translates dotted decimal notation to a network number (host byte order), and zeros in the host part.
inet_ntoa()	Translates 32-bit internet address (network byte order) to dotted decimal notation.
inet_netof()	Extracts network number (host byte order) from 32-bit internet address (network byte order).
inet_lnaof()	Extracts local network address (host byte order) from 32-bit internet address (network byte order).
inet_makeaddr()	Constructs internet address (network byte order) from network number and local network address.

Domain Name Resolution: Resolver calls are used to resolve the symbolic host name into an internet address and to extract more information about the host from the database.

The resolver calls determine whether the name server is present or not present by referencing the custom structure.

To resolve a name with no name server present, the resolver calls check the <*TCPBASE*>\ETC\HOSTS file for an entry that maps the name to an address.

To resolve a name in a name server network, the resolver calls query the domain name server database. If this query fails, the calls then check for an entry in the local <*TCPBASE*>\ETC\HOSTS file.

Socket Library

To use the socket routines described in this chapter, you must have the TCPIP.LIB library file in the <*TCPBASE*>\LIB directory. Also, the following header files must be contained in the <*TCPBASE*>\INCLUDE directory, available on your system.

Socket	Description
TCPERRNO.H	Contains network error definitions.
NETDB.H	Contains data definitions for network utility calls.
TYPES.H	Contains data type definitions.
NETINET\IN.H	Contains definition for Internet constants and structures.
SYS\SOCKET.H	Contains data definitions and socket structure.
SYS\TIME.H	Contains definition of timeval structure.

Porting

The IBM DOS socket implementation differs from the Berkeley socket implementation. The following list summarizes the differences between the IBM DOS socket implementation and the Berkeley implementation:

- Sockets are not DOS files or devices. Socket numbers have no relationship to DOS file handles. Therefore, read(), write(), and close() do not work for sockets. Using read(), write(), or close() gives incorrect results. The recv(), send(), and so_close() functions must be used instead.
- Some socket calls require that the sock_init() routine or the dosip_init() routine, be invoked before the socket calls can be run. Therefore, you should always call either sock_init() or dosip_init(), at the beginning of programs using the socket interface.
- You must make reference to the additional header file <TCPERRNO.H> if you want to reference the networking errors other than those described in the compiler-supplied <ERRNO.H> file.

Compiling and Linking

The following steps describe how to compile and link programs using the Sockets APIs with Microsoft C Version 5.10.

Note: In the following examples, *model* refers to the memory model you use to compile your program: L for large model, S for small model, M for medium model, or C for compact model.

1. Include the <*TCPBASE*>\INCLUDE directory at the beginning of the INCLUDE environment variable so that the C compiler finds the appropriate header files. You can set this interactively or you can include it in the AUTOEXEC.BAT file.

   For example, if the INCLUDE environment variable previously read:

   ```
   SET INCLUDE=C:\MSC\INCLUDE
   ```

   You would change it to read:

   ```
   SET INCLUDE=<TCPBASE>\INCLUDE;C:\MSC\INCLUDE
   ```

2. To compile your program, enter the command:

   ```
   cl /c /J /Fs /Oars /FPc /Zp2 /Amodel myprog.c
   ```

3. To create an executable program, enter the following command:

   ```
   link /noi /stack:6144 /seg:200 myproj.obj,,,
   <TCPBASE>\LIB\model\tcpip.lib;
   ```

Socket Calls

This section provides the C programming language syntax, parameters, and other appropriate information for each socket call supported by TCP/IP for DOS.

accept()

```
#include <types.h>
#include <sys\socket.h>

int accept(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
s	The socket descriptor.
name	The socket address of the connecting client that is filled by accept() before it returns. The format of *name* is determined by the domain in which the client resides. This parameter can be NULL if the caller is not interested in the client address.
namelen	Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. On return, that integer contains the size of the data returned in the storage pointed to by *name*. If *name* is NULL, then *namelen* is ignored and can be NULL.

Description: The accept() call is used by a server to accept a connection request from a client. The call accepts the first connection on its queue of pending connections. The accept() call creates a new socket descriptor with the same properties as *s* and returns it to the caller. If the queue has no pending connection requests, accept() blocks the caller unless *s* is in nonblocking mode. If no connection requests are queued and *s* is in nonblocking mode, accept() returns −1 and sets *errno* to EWOULDBLOCK. The new socket descriptor cannot be used to accept new connections. The original socket, *s*, remains available to accept more connection requests.

The *s* parameter is a stream socket descriptor created with the socket() call. It is usually bound to an address with the bind() call, and can accept connections with the listen() call. The listen() call marks the socket as one that accepts connections and allocates a queue to hold pending connection requests. The listen() call allows the caller to place an upper boundary on the size of the queue.

The *name* parameter is a pointer to a buffer into which the connection requester's address is placed. The *name* parameter is optional and can be set to be the NULL pointer. If set to NULL, the requester's address is not copied into the buffer. The exact format of *name* depends on the addressing domain from which the communication request originated. For example, if the connection request originated in the AF_INET domain, *name* points to a *sockaddr_in* structure as defined in the header file <NETINET\IN.H>. The *namelen* parameter is used only if *name* is not NULL. Before calling accept(), you must set the integer pointed to by *namelen* to the size, in bytes, of the buffer pointed to by *name*. On successful return, the integer pointed to by *namelen* contains the actual number of bytes copied into the buffer. If the buffer is not large enough to hold the address, up to *namelen* bytes of the requester's address are copied.

This call is used only with SOCK_STREAM sockets. There is no way to screen requesters without calling accept(). The application cannot tell the system from which requesters it will accept connections. The caller can, however, choose to close a connection immediately after discovering the identity of the requester.

A socket can be checked for incoming connection requests using the select() call and setting the bit in the read descriptor array.

Return Values and Errno Values: A non-negative socket descriptor indicates success, the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno Value	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space into which information cannot be written.
EOPNOTSUPP	Listen() was not called for socket *s*.
ENOBUFS	Insufficient buffer space available to create the new socket.
EOPNOTSUPP	The *s* parameter is not of type SOCK_STREAM.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no connections are on the queue.

Examples: The following are two examples of the accept() call. In the first, the caller wishes to have the requester's address returned. In the second, the caller does not wish to have the requester's address returned.

```
int clientsocket;
int s;
struct sockaddr clientaddress;
int addrlen;
int accept(int s, struct sockaddr *addr, int *addrlen);
/* socket(), bind(), and listen() have been called */
/* EXAMPLE 1: I want the address now */

addrlen = sizeof(clientaddress);
clientsocket = accept(s, &clientaddress, &addrlen);
/* EXAMPLE 2: I can get the address later using getpeername() */
addrlen = 0;
clientsocket = accept(s, (struct sockaddr *) 0, (int *) 0);
```

See Also: bind(), connect(), getpeername(), listen(), socket().

bind()

```
#include <types.h>
#include <sys\socket.h>

int bind(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Parameter	Description
s	The socket descriptor returned by a previous socket() call.
name	Points to a *sockaddr* structure containing the name that is to be bound to *s*.
namelen	The size of *name* in bytes.

Description: The bind() call binds a unique local name to the socket with descriptor *s*. After calling socket(), a descriptor does not have a name associated with it. However, it does belong to a particular addressing family as specified when socket() is called. The exact format of a name depends on the addressing family. The bind() procedure also allows servers to specify from which network interfaces they wish to receive UDP packets and TCP connection requests.

The *s* parameter is a socket descriptor of any type created by calling socket().

The *name* parameter is a pointer to a buffer containing the name to be bound to *s*. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

Because *s* was created in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in* as defined in the header file <NETINET\IN.H>:

```
struct in_addr
{
        u_long s_addr;
};

struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET. The *sin_port* field is set to the port to which the application must bind. It must be specified in network byte order. If *sin_port* is set to 0, the caller leaves it to the system to assign an available port. The application can call getsockname() to discover the port number assigned. The *sin_addr* field is set to the internet address and must be specified in network byte order. On hosts with more than one network interface (called multi-homed hosts), a caller can select the interface with which it is to bind.

Subsequently, only UDP packets and TCP connection requests from this interface (which match the bound name) are routed to the application. If this field is set to the constant INADDR_ANY, as defined in <NETINET\IN.H>, the caller is requesting that the socket be bound to all network interfaces on the host. Subsequently, UDP packets and TCP connections from all interfaces (which match the bound name) are

routed to the application. This becomes important when a server offers a service to multiple networks. By leaving the address unspecified, the server can accept all UDP packets and TCP connection requests made for its port, regardless of the network interface on which the requests arrived. The *sin_zero* field is not used and must be set to all zeros.

Return Values and Errno Values: The value 0 indicates success, the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
EADDRINUSE	The address is already in use. See the SO_REUSEADDR option described under "getsockopt()" on page 64 and the SO_REUSEADDR option described under the "setsockopt()" on page 88.
EADDRNOTAVAIL	The address specified is not valid on this host. For example, if the internet address does not specify a valid network interface.
EAFNOSUPPORT	The address family is not supported.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a non-writable portion of the caller's address space.
EOPNOTSUPP	The socket is already bound to an address. For example, trying to bind a name to a socket that is in the connected state. This value is also returned if *namelen* is not the expected length.

Examples: The following are examples of the bind() call. Several things should be noted about the examples. The internet address and port must be in network byte order. To put the port into network byte order, a utility routine, htons(), is called to convert a short integer from host byte order to network byte order. The *address* field is set using another utility routine, inet_addr(), which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order. Finally, note that it is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. See "connect()" on page 41 for examples of how a client might connect to servers.

```
int rc;
int s;
struct sockaddr_in myname;
int bind(int s, struct sockaddr *name, int namelen);

/* Bind to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to all network interfaces in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr.s_addr = INADDR_ANY; /* all interfaces */
myname.sin_port = htons(1024);
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
/* Bind to a specific interface in the internet domain.
   Let the system choose a port                          */
/* make sure the sin_zero field is cleared */
memset(&myname, 0, sizeof(myname));
myname.sin_family = AF_INET;
myname.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
myname.sin_port = 0;
⋮
rc = bind(s, (struct sockaddr *) &myname, sizeof(myname));
```

The binding of a stream socket is not complete until a successful call to bind(), listen(), or connect() is made. Applications using stream sockets should check the return values of bind(), listen(), and connect() before using any function that requires a bound stream socket.

See Also: connect(), gethostbyname(), getsockname(), htons(), inet_addr(), listen(), socket().

connect()

```
#include <types.h>
#include <sys\socket.h>

int connect(s, name, namelen)
int s;
struct sockaddr *name;
int namelen;
```

Parameter	Description
s	The socket descriptor.
name	The pointer to a *socket address* structure containing the address of the socket to which a connection will be attempted.
namelen	The size of the *socket address* pointed to by *name* in bytes.

Description: For stream sockets, the connect() call attempts to establish a connection between two sockets. For UDP sockets, the connect() call specifies the peer for a socket. The *s* parameter is the socket used to originate the connection request. The connect() call performs two tasks when called for a stream socket. First, it completes the binding necessary for a stream socket (in case it has not been previously bound using the bind() call). Second, it attempts to make a connection to another socket.

The connect() call on a stream socket is used by the client application to establish a connection to a server. The server must have a passive open pending. If the server is using sockets, this means the server must successfully call bind() and listen() before a connection can be accepted by the server with accept(). Otherwise, connect() returns −1 and *errno* is set to ECONNREFUSED.

If *s* is in blocking mode, the connect() call blocks the caller until the connection is set up, or until an error is received. If the socket is in nonblocking mode then connect() returns −1 with *errno* set to EINPROGRESS if the connection can be initiated (no other errors occurred). The caller can test the completion of the connection setup by calling select() and testing for the ability to write to the socket.

When called for a datagram, connect() specifies the peer with which this socket is associated. This gives the application the ability to use data transfer calls reserved for sockets that are in the connected state. In this case, send(), recv(), sendto(), and recvfrom() are available. Stream sockets can call connect() only once, but datagram sockets can call connect() multiple times to change their association. Datagram sockets can dissolve their association by connecting to an invalid address such as the null address (all fields zeroed).

The *name* parameter is a pointer to a buffer containing the name of the peer to which the application needs to connect. The *namelen* parameter is the size, in bytes, of the buffer pointed to by *name*.

If the server is in the AF_INET domain, the format of the name buffer is expected to be *sockaddr_in*, as defined in the header file <NETINET/IN.H.>

```
struct in_addr
{
        u_long s_addr;
};

struct sockaddr_in
{
        short   sin_family;
        u_short sin_port;
        struct  in_addr sin_addr;
        char    sin_zero[8];
};
```

The *sin_family* field must be set to AF_INET. The *sin_port* field is set to the port to which the server is bound. It must be specified in network byte order. The *sin_zero* field is not used and must be set to all zeros.

Return Values and Errno Values: The value 0 indicates success, the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
EADDRNOTAVAIL	The calling host cannot reach the specified destination.
EAFNOSUPPORT	The address family is not supported.
EALREADY	The socket *s* is marked nonblocking, and a previous connection attempt has not completed.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
ECONNREFUSED	The connection request was rejected by the destination host.
EFAULT	Using *name* and *namelen* would result in an attempt to copy the address into a portion of the caller's address space to which data cannot be written.
EINPROGRESS	The socket *s* is marked nonblocking, and the connection cannot be completed immediately. The EINPROGRESS value does not indicate an error condition.
EOPNOTSUPP	The *namelen* parameter is not a valid length.
EISCONN	The socket *s* is already connected.
ENETUNREACH	The network cannot be reached from this host.
ETIMEDOUT	The connection establishment timed out before a connection was made.

Examples: The following are examples of the connect() call. Several things should be noted about the examples. The internet address and port must be in network byte order. To put the port into network byte order a utility routine, htons(), is called to convert a short integer from host byte order to network byte order. The address field is set using another utility routine, inet_addr(), which takes a character string representing the dotted decimal address of an interface and returns the binary internet address representation in network byte order. It is a good idea to zero the structure before using it to ensure that the name requested does not set any reserved fields. These examples could be used to connect to the servers shown in the examples listed with the call, "bind()" on page 38.

```
int s;
struct sockaddr_in servername;
int rc;
int connect(int s, struct sockaddr *name, int namelen);

/* Connect to server bound to a specific interface in the internet domain */
/* make sure the sin_zero field is cleared */
memset(&servername, 0, sizeof(servername));
servername.sin_family = AF_INET;
servername.sin_addr = inet_addr("129.5.24.1"); /* specific interface */
servername.sin_port = htons(1024);
:
rc = connect(s, (struct sockaddr *) &servername, sizeof(servername));
```

See Also: accept(), bind(), htons(), inet_addr(), listen(), select(), socket().

dosip_init()

```
u_int dosip_init(flags)
u_int flags;
```

Parameter	Description	
flags	If dosip_init fails, the *flags* parameter indicates whether to exit from the calling program with an error message indicating the cause of the error, or return to the calling program with a nonzero return code.	
	NIF_COMPLAIN	Exit from the program.
	NIF_NOCOMPLAIN	Return to the program with a nonzero return code.

Description: The dosip_init() call initializes the socket data structures and checks whether INET.EXE is running or not running. Therefore, either dosip_init() or sock_init() should be called at the beginning of each program that uses socket calls.

Note: Calling dosip_init() with the NIF_COMPLAIN flag is the same as calling sock_init().

Warning: If any socket function is called after the failure of dosip_init, unpredictable results can occur.

See Also: sock_init().

endhostent()

```
void endhostent()
```

Description: The endhostent() call closes the <*TCPBASE*>\ETC\HOSTS file, which contains information about known hosts.

See Also: gethostbyaddr(), gethostbyname(), gethostent(), sethostent().

endnetent()

```
void endnetent()
```

Description: The endnetent() call closes the <*TCPBASE*>\ETC\NETWORKS file, which contains information about known networks.

See Also: getnetbyaddr(), getnetbyname(), getnetent(), setnetent().

endprotoent()

```
void endprotoent()
```

Description: The endprotoent() call closes the <*TCPBASE*>\ETC\PROTOCOL file, which contains information about known protocols.

See Also: getprotobyname(), getprotobynumber(), getprotoent(), setprotoent().

endservent()

```
void endservent()
```

Description: The endservent() call closes the <*TCPBASE*>\ETC\SERVICES file, which contains information about known services.

See Also: getservbyname(), getservbyport(), getservent(), setservent().

gethostbyaddr()

```
#include <netdb.h>

struct hostent *gethostbyaddr(addr, addrlen, domain)
char *addr;
int addrlen;
int domain;
```

Parameter	Description
addr	A pointer to a 32-bit internet address in network byte order.
addrlen	The size of *addr* in bytes.
domain	The address domain supported (AF_INET).

Description: The gethostbyaddr() call tries to resolve the host name through a name server, if one is present. If a name server is not present, gethostbyaddr() sequentially searches the <*TCPBASE*>\ETC\HOSTS file until a matching host address is found or an EOF marker is reached.

The gethostbyaddr() call returns a pointer to a *hostent* structure for the host address specified on the call.

The <NETDB.H> header file defines the *hostent* structure, and contains the following elements:

Element	Description
h_name	Official name of the host.
h_aliases	A zero-terminated array of alternative names for the host.
h_addrtype	The type of address being returned; currently, always set to AF_INET.
h_length	The length of the address in bytes.
h_addr	A pointer to the network address of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endhostent(), gethostbyname(), gethostent(), sethostent().

gethostbyname()

```
#include <netdb.h>

struct hostent *gethostbyname(name)
char *name;
```

Parameter	Description
name	The name of the host being queried.

Description: The gethostbyname() call tries to resolve the host name through a name server, if one is present. If a name server is not present, gethostbyname() searches the <*TCPBASE*>\ETC\HOSTS file until a matching host name is found or an EOF marker is reached.

The gethostbyname() call returns a pointer to a *hostent* structure for the host name specified on the call.

The <NETDB.H> header file defines the *hostent* structure, and contains the following elements:

Element	Description
h_name	Official name of the host.
h_aliases	A zero-terminated array of alternative names for the host.
h_addrtype	The type of address being returned; currently, always set to AF_INET.
h_length	The length of the address in bytes.
h_addr	A pointer to the network address of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endhostent(), gethostbyaddr(), gethostent(), sethostent().

gethostent()

```
#include <netdb.h>

struct hostent *gethostent()
```

Description: The gethostent() call reads the next line of the <*TCPBASE*>\ETC\HOSTS file.

The gethostent() call returns a pointer to the next entry in the HOSTS file.

The <NETDB.H> header file defines the *hostent* structure, and contains the following elements:

Element	Description
h_name	Official name of the host.
h_aliases	A zero-terminated array of alternative names for host.
h_addrtype	The type of address being returned; currently, always set to AF_INET.
h_length	The length of the address in bytes.
h_addr	A pointer to the network address of the host.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *hostent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endhostent(), gethostbyaddr(), gethostbyname(), sethostent().

gethostid()

```
#include <types.h>
#include <sys\socket.h>

u_long gethostid()
```

Description: The gethostid() call gets the unique 32-bit identifier for the current host.

Return Values: The gethostid() call returns the 32-bit identifier of the current host, which should be unique across all hosts.

getnetbyaddr()

```
#include <netdb.h>

struct netent *getnetbyaddr(net, type)
u_long net;
int type;
```

Parameter	Description
net	The network address.
type	The address domain supported (AF_INET).

Description: The getnetbyaddr() call searches the <*TCPBASE*>\ETC\NETWORKS file for the specified network address. See Appendix J, "NETWORKS File Structure," for the format of the NETWORKS file.

The *netent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
n_name	Official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address being returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endnetent(), getnetbyname(), getnetent(), setnetent().

getnetbyname()

```
#include <netdb.h>

struct netent *getnetbyname(name)
char *name;
```

Parameter	Description
name	The pointer to a network name.

Description: The getnetbyname() call searches the <*TCPBASE*>\ETC\NETWORKS file for the specified network name. See Appendix J, "NETWORKS File Structure," for the format of the NETWORKS file.

The getnetbyname() call returns a pointer to a *netent* structure for the network name specified on the call.

The *netent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
n_name	Official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address being returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endnetent(), getnetbyaddr(), getnetent(), setnetent().

getnetent()

```
#include <netdb.h>

struct netent *getnetent()
```

Description: The getnetent() call reads the next entry of the <*TCPBASE*>\ETC\NETWORKS file. See Appendix J, "NETWORKS File Structure," for the format of the NETWORKS file.

The getnetent() call returns a pointer to the next entry in the NETWORKS file.

The *netent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
n_name	Official name of the network.
n_aliases	An array, terminated with a NULL pointer, of alternative names for the network.
n_addrtype	The type of network address being returned. The call always sets this value to AF_INET.
n_net	The network number, returned in host byte order.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *netent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endnetent(), getnetbyaddr(), getnetbyname(), setnetent().

getpeername()

```
#include <types.h>
#include <sys\socket.h>

int getpeername(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
s	The socket descriptor.
name	The internet address of the connected socket that is filled by getpeername() before it returns. The exact format of *name* is determined by the domain in which communication occurs.
namelen	The size of the address structure pointed to by *name* in bytes.

Description: The getpeername() call returns the name of the peer connected to socket *s*. *namelen* must be initialized to indicate the size of the space pointed to by *name* and is set to the number of bytes copied into the space before the call returns. The size of the peer name is returned in bytes. If the buffer of the local host is too small, the peer name is truncated.

Return Values and Errno Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.
ENOTCONN	The socket is not in the connected state.

See Also: accept(), connect(), getsockname(), socket().

getprotobyname()

```
#include <netdb.h>

struct protoent *getprotobyname(name)
char *name;
```

Parameter	Description
name	A pointer to the specified protocol.

Description: The getprotobyname() call searches the <*TCPBASE*>\ETC\PROTOCOL file for the specified protocol name.

The getprotobyname() call returns a pointer to a *protoent* structure for the network protocol specified on the call.

The *protoent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
p_name	Official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endprotoent(), getprotobynumber(), getprotoent(), setprotoent().

getprotobynumber()

```
#include <netdb.h>

struct protoent *getprotobynumber(proto)
int proto;
```

Parameter	Description
proto	The specified protocol number.

Description: The getprotobynumber() call searches the <*TCPBASE*>\ETC\PROTOCOL file for the specified protocol number.

The getprotobynumber() call returns a pointer to a *protoent* structure for the network protocol specified on the call.

The *protoent* structure is defined in the <NETDB.H> header file and contains the following elements:

Element	Description
p_name	The official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endprotoent(), getprotobyname(), getprotoent(), setprotoent().

getprotoent()

```
#include <netdb.h>

struct protoent *getprotoent()
```

Description: The getprotoent() call searches the <*TCPBASE*>\ETC\PROTOCOL file in the directory.

The getprotoent() call returns a pointer to the next entry in the PROTOCOL file.

The *protoent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
p_name	Official name of the protocol.
p_aliases	An array, terminated with a NULL pointer, of alternative names for the protocol.
p_proto	The protocol number.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *protoent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endprotoent(), getprotobyname(), getprotobynumber(), setprotoent().

getservbyname()

```
#include <netdb.h>

struct servent *getservbyname(name, proto)
char *name;
char *proto;
```

Parameter	Description
name	A pointer to the specified service name.
proto	A pointer to the specified protocol.

Description: The getservbyname() call searches the <*TCPBASE*>\ETC\SERVICES file for the specified service name. Searches for a service name must match the protocol if a protocol is stated.

The getservbyname() call returns a pointer to a *servent* structure for the network service specified on the call.

The *servent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
s_name	Official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The required protocol to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endservent(), getservbyport(), getservent(), setservent().

getservbyport()

```
#include <netdb.h>

struct servent *getservbyport(port, proto)
int *port;
char *proto;
```

Parameter	Description
port	The specified port.
proto	A pointer to the specified protocol.

Description: The getservbyport() call sequentially searches the *<TCPBASE>*\ETC\SERVICES file for the specified port number. Searches for a port number must match the protocol if a protocol is stated.

The getservbyport() call returns a pointer to a *servent* structure for the port number specified on the call.

The *servent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
s_name	Official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The required protocol to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endservent(), getservbyname(), getservent(), setservent().

getservent()

```
#include <netdb.h>

struct servent *getservent()
```

Description: The getservent() searches for the next line in the <*TCPBASE*>\ETC\SERVICES file.

The getservent() call returns a pointer to the next entry in the SERVICES file.

The *servent* structure is defined in the <NETDB.H> header file, and contains the following elements:

Element	Description
s_name	Official name of the service.
s_aliases	An array, terminated with a NULL pointer, of alternative names for the service.
s_port	The port number of the service.
s_proto	The required protocol to contact the service.

Return Values: The return value points to static data that is overwritten by subsequent calls. A pointer to a *servent* structure indicates success. A NULL pointer indicates an error or end-of-file.

See Also: endservent(), getservbyname(), getservbyport(), setservent().

getsockname()

```
#include <types.h>
#include <sys\socket.h>

int getsockname(s, name, namelen)
int s;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
s	The socket descriptor.
name	The address of the buffer into which getsockname() copies the name of *s*.
namelen	Must initially point to an integer that contains the size in bytes of the storage pointed to by *name*. Upon return, that integer contains the size of the data returned in the storage pointed to by *name*.

Description: The getsockname() call stores the current name for the socket specified by the *s* parameter into the structure pointed to by the *name* parameter. It returns the address to the socket that has been bound. If the socket is not bound to an address, the call returns with the family set and the rest of the structure is set to zero. For example, an inbound socket in the internet domain would cause the name to point to a *sockaddr_in* structure with the *sin_family* field set to AF_INET and all other fields zeroed.

Stream sockets are not assigned a name, until after a successful call to either bind(), connect(), or accept().

The getsockname() call is often used to discover the port assigned to a socket after the socket has been implicitly bound to a port. For example, an application can call connect() without previously calling bind(). In this case, the connect() call completes the binding necessary by assigning a port to the socket. This assignment can be discovered with a call to getsockname().

Return Values and Errno Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *name* and *namelen* parameters as specified would result in an attempt to access storage outside of the caller's address space.

See Also: accept(), bind(), connect(), getpeername(), socket().

getsockopt()

```
#include <types.h>
#include <sys\socket.h>

int getsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int *optlen;
```

Parameter	Description
s	The socket descriptor.
level	The level for which the option is set. Only SOL_SOCKET is supported.
optname	The name of a specified socket option.
optval	Points to option data.
optlen	Points to the length of the option data.

Description: The getsockopt() call gets options associated with a socket. It can be called only for sockets in the AF_INET domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET as defined in <SYS\SOCKET.H>. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET level is supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *optval* and *optlen* parameters are used to return data used by the particular get command. The *optval* parameter points to a buffer that is to receive the data requested by the get command. The *optlen* parameter points to the size of the buffer pointed to by the *optval* parameter. It must be initially set to the size of the buffer before calling getsockopt(). On return it is set to the actual size of the data returned.

All of the socket level options expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled.

The following options are recognized at the socket level:

Option	Description
SO_BROADCAST	Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_DEBUG	Toggles recording of debugging information.
SO_DONTBLOCK	Sets the socket to nonblocking.

SO_KEEPALIVE	Toggles keep connection alive. TCP uses a timer called the keepalive timer. This timer is used to monitor idle connections that might have been disconnected because of a peer crash or time-out. If this option is toggled, a keepalive packet is periodically sent to the peer. This is used mainly to allow servers to close connections that have already disappeared as a result of clients going away without closing connections. This option has meaning only for stream sockets.
SO_RCVBUF	Gets buffer size for input. This option gets the size of the receiving buffer from the buffer pointed to by *optval*. This allows the buffer size to be tailored for specific application needs, such as increasing the buffer size for high-volume connections.
SO_REUSEADDR	Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the bind() call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error EADDRINUSE is returned if the association already exists.
SO_SNDBUF	Gets buffer size for output. This option gets the size of the sending buffer from the buffer pointed to by *optval*. This allows the send buffer size to be tailored for specific application needs such as increasing the buffer size for high volume connections.
SO_SNDTIMEO	Sends time-out.

Return Values and Errno Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
EADDRINUSE	The address is already in use.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *optval* and *optlen* parameters would result in an attempt to access memory outside the caller's address space.
ENOPROTOOPT	The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Examples: The following are examples of the getsockopt() call. See "setsockopt()" on page 88 for examples of how the setsockopt() call options are set.

```
int rc;
int s;
int optval;
int optlen;
int getsockopt(int s, int level, int optname, char *optval, int *optlen);

:
/* Get the size of the sending buffer */
optlen = sizeof(int);
rc = getsockopt(
        s, SOL_SOCKET, SO_SNDBUF, (char *) &optval, &optlen);
if (rc == 0)
{
        printf("send buffer size = %\n," optval);
}
```

See Also: getprotobyname(), setsockopt(), socket().

htonl()

```
#include <types.h>
#include <netinet\in.h>

u_long htonl(a)
u_long a;
```

Parameter	Description
a	The unsigned long integer to be put into network byte order.

Description: The htonl() call translates a long integer from host byte order to network byte order.

Return Values: Returns the translated long integer.

See Also: htons(), ntohl(), ntohs().

htons()

```
#include <types.h>
#include <netinet\in.h>

u_short htons(a)
u_short a;
```

Parameter	Description
a	The unsigned short integer to be put into network byte order.

Description: The htons() call translates a short integer from host byte order to network byte order.

Return Values: Returns the translated short integer.

See Also: htonl(), ntohl(), ntohs().

inet_addr()

```
#include <types.h>
#include <netinet\in.h>

u_long inet_addr(cp)
char *cp;
```

Parameter	Description
cp	A character string in standard '.' notation.

Description: The inet_addr() call interprets character strings representing numbers expressed in standard '.' notation and returns numbers suitable for use as an internet address.

Values specified in standard '.' notation take one of the following forms:

```
a.b.c.d
a.b.c
a.b
a
```

When a four-part address is specified, each part is interpreted as a byte of data and assigned, from left to right, to one of the four bytes of an internet address.

When a three-part address is specified, the last part is interpreted as a 16-bit quantity and placed in the two rightmost bytes of the network address. This makes the three-part address format convenient for specifying Class B network addresses as 128.net.host.

When a two-part address is specified, the last part is interpreted as a 24-bit quantity and placed in the three rightmost bytes of the network address. This makes the two-part address format convenient for specifying Class A network addresses as net.host.

When a one-part address is specified, the value is stored directly in the network address space without any rearrangement of its bytes.

Numbers supplied as address parts in standard '.' notation can be decimal, hexadecimal, or octal. Numbers are interpreted in C language syntax. A leading 0x implies hexadecimal; a leading 0 implies octal. A number without a leading 0 implies decimal.

Return Values: The internet address is returned in network byte order.

See Also: inet_lnaof(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa().

inet_lnaof()

```
#include <types.h>
#include <netinet\in.h>

u_long inet_lnaof(in)
struct in_addr in;
```

Parameter	Description
in	The host internet address.

Description: The inet_lnaof() call breaks apart the internet host address and returns the local network address portion.

Return Values: The local network address is returned in host byte order.

See Also: inet_addr(), inet_makeaddr(), inet_netof(), inet_network(), inet_ntoa().

inet_makeaddr()

```
#include <types.h>
#include <netinet\in.h>

struct in_addr inet_makeaddr(net, lna)
u_long net;
u_long lna;
```

Parameter	Description
net	The network number.
lna	The local network address.

Description: The inet_makeaddr() call takes a network number and a local network address and constructs an internet address.

Return Values: The internet address is returned in network byte order.

See Also: inet_addr(), inet_lnaof(), inet_netof(), inet_network(), inet_ntoa().

inet_netof()

```
#include <types.h>
#include <netinet\in.h>

u_long inet_netof(in)
struct in_addr in;
```

Parameter	Description
in	The internet address in network byte order.

Description: The inet_netof() call breaks apart the internet host address and returns the network number portion.

Return Values: The network number is returned in host byte order.

See Also: inet_addr(), inet_lnaof(), inet_makeaddr(), inet_network(), inet_ntoa().

inet_network()

```
#include <types.h>
#include <netinet\in.h>

u_long inet_network(cp)
char *cp;
```

Parameter	Description
cp	A character string in standard '.' notation.

Description: The inet_network() call interprets character strings representing numbers expressed in standard '.' notation and returns numbers suitable for use as a network number.

Return Values: The network number is returned in host byte order.

See Also: inet_addr(), inet_lnaof(), inet_makeaddr(), inet_netof(), inet_ntoa().

inet_ntoa()

```
#include <types.h>
#include <netinet\in.h>

char *inet_ntoa(in)
struct in_addr in;
```

Parameter	Description
in	The host internet address.

Description: The inet_ntoa() call returns a pointer to a string expressed in the dotted-decimal notation. inet_ntoa() accepts an internet address expressed as a 32-bit quantity in network byte order and returns a string expressed in dotted-decimal notation.

Return Values: Returns a pointer to the internet address expressed in dotted-decimal notation.

See Also: inet_addr(), inet_lnaof(), inet_makeaddr(), inet_network(), inet_ntoa().

listen()

```
#include <types.h>
#include <sys\socket.h>

int listen(s, backlog)
int s;
int backlog;
```

Parameter	Description
s	The socket descriptor.
backlog	Defines the maximum length for the queue of pending connections.

Description: The listen() call applies only to stream sockets. It performs two tasks: it completes the binding necessary for a socket *s*, if bind() has not been called for *s*, and it creates a connection request queue of length *backlog* to queue incoming connection requests. Once full, additional connection requests are ignored.

The listen() call indicates a readiness to accept client connection requests. It transforms an active socket into a passive socket. Once called, *s* can never be used as an active socket to initiate connection requests. Calling listen() is the third of four steps that a server performs to accept a connection. It is called after allocating a stream socket with socket(), and after binding a name to *s* with bind(). It must be called before calling accept().

If the backlog is less than 0, *backlog* is set to 0. If the backlog is greater than SOMAXCONN, as defined in <SYS\SOCKET.H>, *backlog* is set to SOMAXCONN.

Return Values and Errno Values: The value 0 indicates success, the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EOPNOTSUPP	The *s* parameter is not a socket descriptor that supports the listen() call.

See Also: accept(), bind(), connect(), socket().

ntohl()

```
#include <types.h>
#include <netinet.\in.h>

u_long ntohl(a)
u_long a;
```

Parameter	Description
a	The unsigned long integer to be put into host byte order.

Description: The ntohl() call translates a long integer from network byte order to host byte order.

Return Values: Returns the translated long integer.

See Also: htonl(), htons(), ntohs().

ntohs()

```
#include <types.h>
#include <netinet\in.h>

u_short ntohs(a)
u_short a;
```

Parameter	Description
a	The unsigned short integer to be put into host byte order.

Description: The ntohs() call translates a short integer from network byte order to host byte order.

Return Values: Returns the translated short integer.

See Also: htonl(), htons(), ntohl().

recv()

```
#include <types.h>
#include <sys\socket.h>

int recv(s, buf, len, flags)
int s;
char *buf;
int len;
int flags;
```

Parameter	Description
s	The socket descriptor.
buf	The pointer to the buffer that receives the data.
len	The length in bytes of the buffer pointed to by the *buf* parameter.
flags	The *flags* parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain.
	MSG_OOB — Reads any out-of-band data on the socket.
	MSG_PEEK — Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.

Description: The recv() call receives data on a socket with descriptor *s* and stores it in a buffer. The recv() call applies only to connected sockets.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available at the socket with descriptor *s*, the recv() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode.

Return Values and Errno Values: If successful, the length, in bytes, of the message or datagram is returned. The value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: connect(), getsockopt(), recvfrom(), select(), send(), sendto(), setsockopt(), socket().

recvfrom()

```
#include <types.h>
#include <sys\socket.h>

int recvfrom(s, buf, len, flags, name, namelen)
int s;
char *buf;
int len;
int flags;
struct sockaddr *name;
int *namelen;
```

Parameter	Description
s	The socket descriptor.
buf	The pointer to the buffer that receives the data.
len	The length in bytes of the buffer pointed to by the *buf* parameter.
flags	A parameter that can be set to 0 or MSG_PEEK. Setting this parameter is supported only for sockets in the AF_INET domain.
	MSG_OOB — Reads any out-of-band data on the socket.
	MSG_PEEK — Peeks at the data present on the socket; the data is returned but not consumed, so that a subsequent receive operation sees the same data.
name	A pointer to a socket address structure from which data is received. If *name* is a nonzero value, the source address is returned.
namelen	The size of *name* in bytes.

Description: The recvfrom() call receives data on a socket with descriptor *s* and stores it in a buffer. The recvfrom() call applies to any datagram socket, whether connected or unconnected.

If *name* is nonzero, the source address of the message is filled. *namelen* must first be initialized to the size of the buffer associated with *name*, and is then modified on return to indicate the actual size of the address stored there.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If datagram packets are not available at the socket with descriptor *s*, the recvfrom() call waits for a message to arrive and blocks the caller, unless the socket is in non-blocking mode.

Return Values and Errno Values: If successful, the length, in bytes, of the message or datagram is returned. The value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: getsockopt(), recv(), select(), send(), sendto(), setsockopt(), socket().

select()

```
#include <types.h>
#include <sys\socket.h>
#include <sys\time.h>

int select(nfds, readfds, writefds, exceptfds, timeout)
int nfds;
fd_set *readfds;
fd_set *writefds;
fd_set *exceptfds;
struct timeval *timeout;
```

Parameter	Description
nfds	The number of socket descriptors to check.
readfds	Points to a bit mask of descriptors to check for reading.
writefds	Points to a bit mask of descriptors to check for writing.
exceptfds	Points to a bit mask of descriptors to be checked for exceptional pending conditions.
timeout	Points to the time to wait for the select() call to complete.

Description: The select() call monitors activity on a set of different sockets until a timeout expires, to see if any sockets are ready for reading or writing, or if any exceptional conditions are pending. The bit mask is made up of an array of integers. Macros are provided to manipulate the bit masks.

Macro	Description
FD_SET(*socket, bit_mask_address*)	Sets the bit for the socket in the bit mask pointed to by *bit_mask_address*.
FD_CLR(*socket, bit_mask_address*)	Clears the bit.
FD_ISSET(*socket, bit_mask_address*)	Returns true if the bit is set for this socket descriptor; otherwise, it returns false.
FD_ZERO	Clears the entire bit mask for all socket descriptors.

Note: For macros FD_SET, FD_CLR, and FD_ISSET, the parameters *socket* and *bit_mask_address* should be defined in the following manner:

```
int socket;
struct fd_set *bit_mask_address, bit_mask_address;
```

The first nfds descriptors in each bit mask are tested for the specified condition.

Socket descriptors are specified by setting bits in a bit mask. If timeout is a NULL pointer, the call blocks indefinitely until one of the requested conditions is satisfied. If timeout is non-NULL, it specifies the maximum time to wait for the call to complete. To poll a set of sockets, the timeout pointer should point to a zeroed *timeval* structure. The *timeval* structure is defined in the <SYS\TIME.H> header file and contains the following fields:

Field	Description
tv_sec	The number of seconds.
tv_usec	The number of microseconds.

Setting any of the descriptor pointers to zero indicates that no checks are to be made for the conditions. For example, setting *exceptfds* to be a NULL pointer causes the select call to check for only read and write conditions.

Return Values and Errno Values: The total number of ready sockets (in all bit masks) is returned. The value −1 indicates an error. The value 0 indicates an expired time limit. If the return value is greater than 0, the socket descriptors in each bit mask that are ready are set to 1. All others are set to 0. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	One of the descriptor sets specified an invalid descriptor.
EFAULT	One of the parameters pointed to a value outside the caller's address space.
EINVAL	One of the fields in the *timeval* structure is invalid.

See Also: accept(), connect(), recv(), send().

send()

```
#include <types.h>
#include <sys\socket.h>

int send(s, msg, len, flags)
int s;
char *msg;
int len;
int flags;
```

Parameter	Description
s	The socket descriptor.
msg	Points to the buffer containing the message to transmit.
len	The length of the message pointed to by the *msg* parameter.
flags	The *flags* parameter is set by specifying one or more of the following flags. If more than one flag is specified, the logical OR operator (|) must be used to separate them. Setting this parameter is supported only for sockets in the AF_INET domain.

MSG_OOB	Sends out-of-band data on sockets that support this notion. Only SOCK_STREAM sockets created in the AF_INET address family support out-of-band data.
MSG_FLUSH	This option flushes the data on send().

Description: The send() call sends packets on the socket with descriptor *s*. The send() call applies to all connected sockets.

If buffer space is not available at the socket to hold the message to be transmitted, the send() call normally blocks, unless the socket is placed in nonblocking mode. The select() call can be used to determine when it is possible to send more data.

Return Values and Errno Values: No indication of failure to deliver is implicit in a send() routine. The value −1 indicates locally detected errors. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
ENOBUFS	No buffer space is available to send the message.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: connect(), getsockopt(), recv(), recvfrom(), select(), sendto(), socket().

sendto()

```
#include <types.h>
#include <sys\socket.h>

int sendto(s, msg, len, flags, to, tolen)
int s;
char *msg;
int len;
int flags;
struct sockaddr *to;
int tolen;
```

Parameter	Description
s	The socket descriptor.
msg	The pointer to the buffer containing the message to transmit.
len	The length of the message in the buffer pointed to by the *msg* parameter.
flags	A parameter that can be set to 0. Setting this parameter is supported only for sockets in the AF_INET domain.
	MSG_FLUSH The option flushes the data on send.
to	The address of the target.
tolen	The size of the address pointed to by *to*.

Description: The sendto() call sends packets on the socket with descriptor *s*. The sendto() call applies to any datagram socket, whether connected or unconnected.

Return Values and Errno Values: If successful, the number of characters sent is returned. The value −1 indicates an error. The value of *errno* indicates the specific error.

No indication of failure to deliver is implied in the return value of this call when used with datagram sockets.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EINVAL	*tolen* is not the size of a valid address for the specified address family.
EMSGSIZE	The message was too big to be sent as a single datagram. The default is 8192, and the maximum is 32 767.
ENOBUFS	No buffer space is available to send the message.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: recv(), recvfrom(), send(), select(), socket().

sethostent()

```
int sethostent(stayopen)
int stayopen;
```

Parameter	Description
stayopen	Tells the HOST file whether to remain open after each call.

Description: The sethostent() call opens and rewinds the <*TCPBASE*>\ETC\HOSTS file. If the *stayopen* flag is nonzero, the HOSTS file remains open after each call.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

See Also: endhostent(), gethostbyaddr(), gethostbyname(), gethostent().

setnetent()

```
int setnetent(stayopen)
int stayopen;
```

Parameter	Description
stayopen	Tells the NETWORKS file whether to remain open after each call.

Description: The setnetent() call opens and rewinds the <*TCPBASE*>\ETC\NETWORKS file which contains information about known networks. If *stayopen* is nonzero the NETWORKS file remains open after each call. See Appendix J, "NETWORKS File Structure," for the format of the NETWORKS file.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

See Also: endnetent(), getnetbyaddr(), getnetbyname(), getnetent().

setprotoent()

```
int setprotoent(stayopen)
int stayopen;
```

Parameter	Description
stayopen	Tells the PROTOCOL file whether to remain open after each call.

Description: The setprotoent() call opens and rewinds the <*TCPBASE*>\ETC\PROTOCOL file, which contains information about known protocols. If the *stayopen* flag is nonzero, PROTOCOL file remains open after each call.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

See also: endprotoent(), getprotobyname(), getprotobynumber(), getprotoent().

setservent()

```
int setservent(stayopen)
int stayopen;
```

Parameter	Description
stayopen	Tells the SERVICES file whether to remain open after each call.

Description: The setservent() call opens and rewinds the <*TCPBASE*>\ETC\SERVICES file, which contains information about known services and well-ports. See "Ports," for more information on the SERVICES file.

Return Values: The value 0 indicates success, the value −1 indicates an error. The value of *errno* indicates the specific error.

See Also: endservent(), getservbyname(), getservbyport(), getservent().

setsockopt()

```
#include <types.h>
#include <sys\socket.h>

int setsockopt(s, level, optname, optval, optlen)
int s;
int level;
int optname;
char *optval;
int optlen;
```

Parameter	Description
s	The socket descriptor.
level	The level for which the option is being set. Only SOL_SOCKET is supported.
optname	The name of a specified socket option.
optval	Points to option data.
optlen	Specifies the length of the option data.

Description: The setsockopt() call sets options associated with a socket. It can be called only for sockets in the AF_INET domain. Options can exist at multiple protocol levels; they are always present at the highest socket level.

When manipulating socket options, you must specify the level at which the option resides and the name of the option. To manipulate options at the socket level, the *level* parameter must be set to SOL_SOCKET, as defined in <SYS\SOCKET.H>. To manipulate options at any other level, such as the TCP or IP level, supply the appropriate protocol number for the protocol controlling the option. Currently, only the SOL_SOCKET level is supported. The getprotobyname() call can be used to return the protocol number for a named protocol.

The *optval* and *optlen* parameters are used to pass data used by the particular set command. The *optval* parameter points to a buffer containing the data needed by the set command. The *optval* parameter is optional and can be set to the NULL pointer, if data is not needed by the command. The *optlen* parameter must be set to the size of the data pointed to by *optval*.

All of the socket level options expect *optval* to point to an integer and *optlen* to be set to the size of an integer. When the integer is nonzero, the option is enabled. When it is zero, the option is disabled.

The following options are recognized at the socket level:

Option	Description
SO_BROADCAST	Toggles the ability to broadcast messages. If this option is enabled, it allows the application to send broadcast messages over *s*, if the interface specified in the destination supports broadcasting of packets. This option has no meaning for stream sockets.
SO_DONTBLOCK	Sets sockets to nonblocking.

SO_KEEPALIVE	Toggles keep connection alive. TCP uses a timer called the keepalive timer. This timer is used to monitor idle connections that might have been disconnected because of a peer crash or time-out. If this option is toggled, a keepalive packet is periodically sent to the peer. This is used mainly to allow servers to close connections that have already disappeared as a result of clients going away without closing connections. This option only has meaning for stream sockets.
SO_RCVBUF	Sets buffer size for input. This option sets the size of the receive buffer to the value contained in the buffer pointed to by *optval*. This allows the buffer size to be tailored for specific application needs, such as increasing the buffer size for high-volume connections.
SO_REUSEADDR	Toggles local address reuse. When enabled, this option allows local addresses that are already in use to be bound. This alters the normal algorithm used in the bind() call. The system checks at connect time to be sure that no local address and port have the same foreign address and port. The error EADDRINUSE is returned if the association already exists.
SO_SNDBUF	Sets buffer size for output. This option sets the size of the send buffer to the value contained in the buffer pointed to by *optval* This allows the send buffer size to be tailored for specific application needs, such as increasing the buffer size for high-volume connections.
SO_SNDTIMEO	Sends time-out.

Return Values and Errno Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
EADDRINUSE	The address is already in use.
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using *optval* and *optlen* parameters would result in an attempt to access memory outside the caller's address space.
ENOPROTOOPT	The *optname* parameter is unrecognized, or the *level* parameter is not SOL_SOCKET.

Examples: The following are examples of the setsockopt() call. See "getsockopt()" on page 64 for examples of how the getsockopt() options set are queried.

```
int rc;
int s;
int optval;
int setsockopt(int s, int level, int optname, char *optval, int optlen)
:
/* Set the send buffer size */
optval = 16384;
rc = setsockopt(s, SOL_SOCKET, SO_SNDBUF, (char *) &optval, sizeof(int));
```

See Also: getprotobyname(), getsockopt(), socket().

shutdown()

```
int shutdown(s, how)
int s;
int how;
```

Parameter	Description
s	The socket descriptor.
how	The condition of the shutdown. The values 0, 1, or 2 set the condition.

Description: The shutdown() call shuts down all or part of a duplex connection. *how* sets the condition for shutting down the connection to socket *s*.

how can have a value of 0, 1, or 2, where:

- 0 ends communication from socket *s*.
- 1 ends communication to socket *s*.
- 2 ends communication both to and from socket *s*.

Return Values and Errno Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EINVAL	The *how* parameter was not set to one of the valid values. Valid values are 0, 1, and 2.

See Also: accept(), connect(), socket(), so_close().

sock_init()

```
int sock_init()
```

Description: There are no parameters associated with this call. The sock_init() call initializes the socket data structures and checks whether or not INET.EXE is running. Therefore, sock_init() should be called at the beginning of each program that uses socket().

Return Values: The value 0 indicates success, the value 1 indicates an error.

See Also: dosip_init().

socket()

```
#include <types.h>
#include <sys\socket.h>

int socket(domain, type, protocol)
int domain;
int type;
int protocol;
```

Parameter	Description
domain	The address domain requested. It must be AF_INET.
type	The type of socket created, either SOCK_STREAM or SOCK_DGRAM.
protocol	The protocol requested. Some possible values are 0, IPPROTO_UDP, or IPPROTO_TCP.

Description: The socket() call creates an endpoint for communication and returns a socket descriptor representing the endpoint. Different types of sockets provide different communication services.

The *domain* parameter specifies a communication domain within which communication is to take place. This parameter selects the address family (format of addresses within a domain) that is used. The only family supported is AF_INET, which is the internet domain. This constant is defined in the <SYS\SOCKET.H> header file.

The *type* parameter specifies the type of socket created. The type is analogous with the semantics of the communication requested. These socket type constants are defined in the <SYS\SOCKET.H> header file. The types supported are:

Socket Type	Description
SOCK_STREAM	Provides sequenced, two-way byte streams that are reliable and connection-oriented. They support a mechanism for out-of-band data.
SOCK_DGRAM	Provides datagrams, which are connectionless messages of a fixed maximum length whose reliability is not guaranteed. Datagrams can be corrupted, received out of order, lost, or delivered multiple times.

The *protocol* parameter specifies a particular protocol to be used with the socket. In most cases, a single protocol exists to support a particular type of socket in a particular addressing family. If the *protocol* field is set to 0, the system selects the default protocol number for the domain and socket type requested. Protocol numbers are found in the <*TCPBASE*>\ETC\PROTOCOL file. Alternatively, the getprotobyname() call can be used to get the protocol number for a protocol with a known name. Currently, *protocol* defaults are TCP for stream sockets and UDP for datagram sockets.

SOCK_STREAM sockets model duplex byte streams. They provide reliable, flow-controlled connections between peer applications. Stream sockets are either active or passive. Active sockets are used by clients who initiate connection requests with connect(). By default, socket() creates active sockets. Passive sockets are used by servers to accept connection requests with the connect() call. An active socket is transformed into a passive socket by binding a name to the socket with the bind() call and by indicating a willingness to accept connections with

the listen() call. Once a socket is passive, it cannot be used to initiate connection requests.

In the AF_INET domain, the bind() call applied to a stream socket lets the application specify the networks from which it is willing to accept connection requests. The application can fully specify the network interface by setting the *internet address* field in the *address* structure to the internet address of a network interface. Alternatively, the application can use a wildcard to specify that it wants to receive connection requests from any network. This is done by setting the *internet address* field in the *address* structure to the constant INADDR_ANY as defined in <SYS\SOCKET.H>.

Once a connection has been established between stream sockets, any of the data transfer calls can be used: send(), recv(), sendto(), recvfrom(). Usually, a send-recv pair is used for sending data on stream sockets.

SOCK_DGRAM sockets model datagrams. They provide connectionless message exchange with no guarantees on reliability. Messages sent have a maximum size.

There is no active or passive analogy to stream sockets with datagram sockets. Servers must still call bind() to name a socket and to specify from which network interfaces it wishes to receive packets. Wildcard addressing, as described for stream sockets, applies for datagram sockets also. Because datagram sockets are connectionless, the listen() call has no meaning for them and must not be used with them.

Once an application has received a datagram socket it can exchange datagrams using the sendto() and recvfrom() calls. If the application goes one step further by calling connect() and fully specifying the name of the peer with which all messages will be exchanged, then the other data transfer calls send() and recv() can also be used. For more information about placing a socket into the connected state, see "connect()" on page 41.

Datagram sockets allow messages to be broadcast to multiple recipients. Setting the destination address to be a broadcast address is network interface dependent (depends on class of address and whether sub-nets are being used). The constant INADDR_BROADCAST, defined in <SYS\SOCKET.H> can be used to broadcast to the primary network if the primary network configured supports broadcast.

Outgoing packets have an IP header prefixed to them. IP options can be set and inspected using the setsockopt() and getsockopt() calls respectively. Incoming packets are received with the IP header and options intact.

Sockets are deallocated with the so_close() call.

Return Values and Errno Values: A non-negative socket descriptor indicates success. The value −1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
EPROTONOSUPPORT	The *protocol* is not supported in this *domain* or this *protocol* is not supported for this socket *type*.

Examples: The following are examples of the socket() call.

```
int s;
struct protoent *p;
struct protoent *getprotobyname(char *name);
int socket(int domain, int type, int protocol);
:
/* Get stream socket in internet domain with default protocol */
s = socket(AF_INET, SOCK_STREAM, 0);
:
/* Get datagram socket in internet domain for UDP protocol */
p = getprotobyname("udp");
s = socket(AF_INET, p->p_proto);
```

See Also: accept(), bind(), connect(), getprotobyname(), getsockname(), getsockopt(), recv(), recvfrom(), select(), send(), sendto(), shutdown(), so_close().

so_close()

```
#include <types.h>
#include <sys\socket.h>

int so_close(s)
int s;
```

Parameter	Description
s	The descriptor of the socket to discard.

Description: The so_close() call shuts down the socket associated with the socket descriptor *s*, and frees resources allocated to the socket. If *s* refers to an open TCP connection, the connection is closed.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *errno* indicates the specific error.

See Also: accept(), socket().

so_flush()

```
#include <types.h>
#include <sys\socket.h>

int so_flush(s)
int s;
```

Parameter	Description
s	The socket descriptor.

Description: The so_flush() call flushes the packet with descriptor *s*.

so_read()

```
#include <types.h>
#include <sys\socket.h>

int read(s, buf, len)
int s;
char *buf;
int len;
```

Parameter	Description
s	The socket descriptor.
buf	The pointer to the buffer that receives the data.
len	The length in bytes of the buffer pointed to by the *buf* parameter.

Description: The so_read() call receives data on a socket with descriptor *s* and stores it in a buffer. The so_read() call applies only to connected sockets.

This call returns the length of the incoming message or data. If a datagram packet is too long to fit in the supplied buffer, datagram sockets discard excess bytes. If data is not available at the socket with descriptor *s*, the so_read() call waits for a message to arrive and blocks the caller, unless the socket is in nonblocking mode.

Return Values and Errno Values: If successful, the length, in bytes, of the message or datagram is returned. The value –1 indicates an error. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *buf* and *len* parameters would result in an attempt to access memory outside the caller's address space.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: connect(), getsockopt(), recvfrom(), select(), send(), sendto(), setsockopt(), socket().

so_write()

```
#include <types.h>
#include <sys\socket.h>

int so_write(s, msg, len)
int s;
char *msg;
int len;
```

Parameter	Description
s	The socket descriptor.
msg	Points to the buffer containing the message to transmit.
len	The length of the message pointed to by the *msg* parameter.

Description: The so_write() call sends packets on the socket with descriptor *s*. The so_write() call applies to all connected sockets.

If buffer space is not available at the socket to hold the message to be transmitted, the so_write() call normally blocks, unless the socket is placed in nonblocking mode. The select() call can be used to determine when it is possible to send more data.

Return Values and Errno Values: No indication of failure to deliver is implicit in a so_write() routine. The value −1 indicates locally detected errors. The value of *errno* indicates the specific error.

Errno	Description
ENOTSOCK	The *s* parameter is not a valid socket descriptor.
EFAULT	Using the *msg* and *len* parameters would result in an attempt to access memory outside the caller's address space.
ENOBUFS	No buffer space is available to send the message.
EWOULDBLOCK	The *s* parameter is in nonblocking mode and no data is available to read.

See Also: connect(), getsockopt(), recv(), recvfrom(), select(), sendto(), socket().

Chapter 4. Remote Procedure Calls (RPCs)

Chapter 4. Remote Procedure Calls (RPCs)

This chapter describes the high-level remote procedure calls (RPCs) implemented in TCP/IP for DOS, including the RPC programming interface to the C language, and communication between processes.

TCP/IP for DOS does not support RPCs with raw sockets, a local portmapper, or a RPC server on the DOS machine. Applications with client/server functions must run the server routines on a machine using *TCP/IP Version 1.2 for OS/2*, *TCP/IP Version 2.0 for VM*, *TCP/IP Version 2.0 for MVS* or AIX* WorkStation.

The RPC protocol enables users to work with remote procedures as if the procedures were local. The remote procedure calls are defined through routines contained in the RPC protocol. Each call message is matched with a reply message. The RPC protocol is a message-passing protocol that implements other non-RPC protocols such as batching and broadcasting remote calls. The RPC protocol also supports callback procedures and the select subroutines on the server side.

RPC provides an authentication process that identifies the server and client to each other. RPC includes a slot for the authentication parameters on every remote procedure call so that the caller can identify itself to the server. The client package generates and returns authentication parameters. RPC supports various types of authentication, such as the UNIX** systems.

In RPC, each server supplies a program that is a set of procedures. The combination of a host address, a program number, and a procedure number specifies one remote service procedure. In the RPC model, the client makes a procedure call to send a data packet to the server. When the packet arrives, the server calls a dispatch routine, performs whatever service is requested, and sends a reply back to the client. The procedure call then returns to the client.

RPC is divided into three layers: highest, intermediate, and lowest. The RPC interface is generally used to communicate between processes on different workstations in a network. However, RPC works just as well for communication between different processes on the same workstation.

The port mapper program maps RPC program and version numbers to a transport-specific port number. The port mapper program makes dynamic binding of remote programs possible.

To use the RPC protocol, you must be familiar with C language programming and have a working knowledge of networking concepts.

For more information on the RPC and XDR protocols, see the Sun Microsystems publication, *Networking on the Sun Workstation: Remote Procedure Call Programming Guide*.

The RPC Interface

The RPC interface enables programmers to write distributed applications using high-level RPCs rather than lower-level calls based on sockets.

When you use RPCs, the client communicates with a server. The client invokes a procedure to send a call message to the server. When the message arrives, the

server calls a dispatch routine, and performs the requested service. The server sends back a reply message, after which the original procedure call returns to the client program with a value derived from the reply message.

See Appendix C, "Sample RPC Programs," for sample RPC client and server programs. Figure 23, and Figure 24 on page 103, provide an overview of the high-level RPC client and server processes from initialization through cleanup.

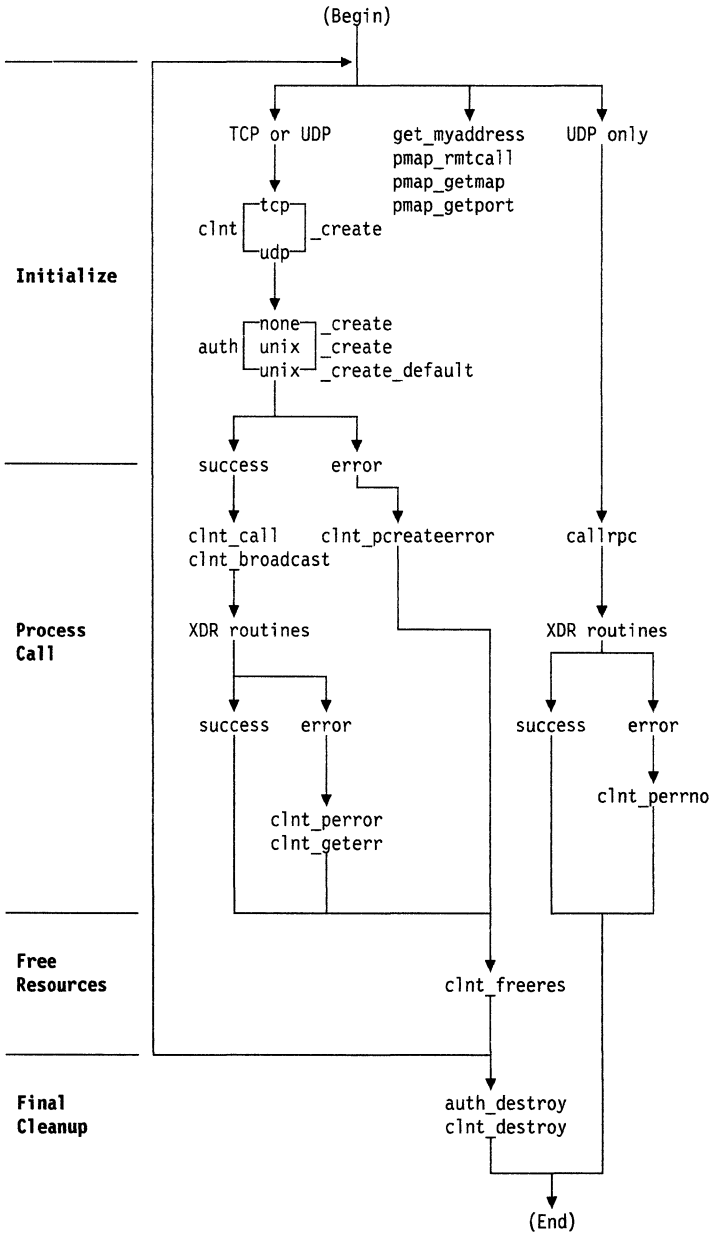


Figure 23. Remote Procedure Call (Client)

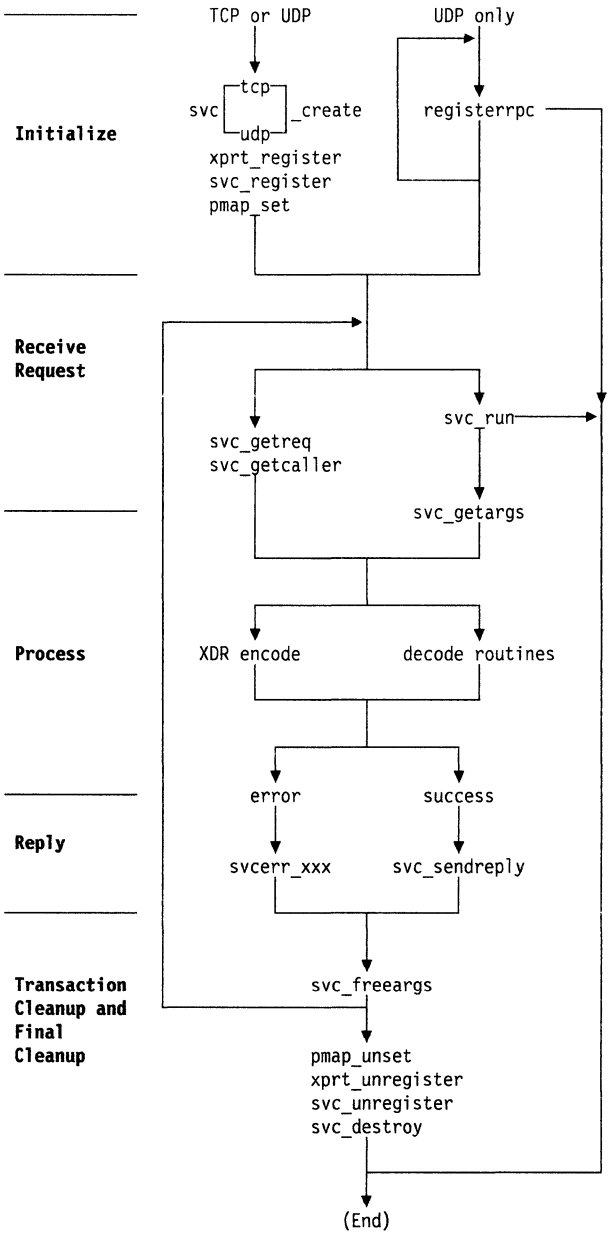


Figure 24. Remote Procedure Call (Server)

RPC Support for DOS

The RPC protocol permits remote execution of subroutines across a TCP/IP network. RPC, together with the eXternal Data Representation (XDR) protocol, defines a standard for representing data that is independent of internal protocols or formatting. RPCs can communicate between processes on the same or different hosts.

TCP/IP for DOS does not support RPCs with raw sockets, a local portmapper, or a RPC server on the DOS machine. Applications with client/server functions must run the server routines on a machine using *TCP/IP Version 1.2 for OS/2*, *TCP/IP Version 2.0 for VM*, *TCP/IP Version 2.0 for MVS* or AIX WorkStation.

RPC Client Calls

The following is a list of RPC client calls supported by TCP/IP for DOS.

auth_destroy()
authunix_create()
callrpc()
clnt_call()
clnt_freeres()
clnt_pcreateerror()
clnt_perror()
clntudp_create()
pmap_getmaps()
pmap_rmtcall()
xdr_accepted_reply()
xdr_array()
xdr_bytes()
xdr_callmsg()
xdr_enum()
xdr_inline()
xdr_long()
xdr_opaque_auth()
xdr_pmaplist()
xdr_rejected_reply()
xdr_short()
xdr_u_int()
xdr_u_short()
xdr_void()
xdrmem_create()
xdrrec_endofrecord()
xdrrec_skiprecord()
xprt_register()
authnone_create()
authunix_create_default()
clnt_broadcast()
clnt_destroy()
clnt_geterr()
clnt_perrno()
clnttcp_create()
get_myaddress()
pmap_getport()
rpc_createerr
xdr_authunix_parms()
xdr_bool()
xdr_callhdr()
xdr_double()
xdr_float()
xdr_int()
xdr_opaque()
xdr_pmap()
xdr_reference()
xdr_replymsg()
xdr_string()
xdr_u_long()
xdr_union()
xdr_wrapstring()
xdrrec_create()
xdrrec_eof()
xdrstdio_create()
xprt_unregister()

RPC Server Calls

The following is a list of RPC server calls which are not supported by TCP/IP for DOS.

pmap_set()
registerrpc()
svc_fds
svc_getargs()
svc_getreq()
svc_run()
svc_unregister()
pmap_unset()
svc_destroy()
svc_freeargs()
svc_getcaller()
svc_register()
svc_sendreply()
svcerr_auth()

svcerr_decode()
svcerr_noprog()
svcerr_systemerr()
svctcp_create()
svcerr_noproc()
svcerr_progvers()
svcerr_weakauth()
svcudp_create()

Portmapper

Portmapper is the software that supplies client programs with the port numbers of server programs.

You can communicate between different computer operating systems when messages are directed to port numbers rather than to targeted remote programs. Clients contact server programs by sending messages to the port numbers where receiving processes receive the message. Because you make requests to the port number of a server rather than directly to a server program, client programs need a way to find the port number of the server programs they wish to call. Portmapper standardizes the way clients locate the port number of the server programs supported on a network.

Portmapper resides on all hosts on well-known port 111. See Appendix A, "Well-Known Port Assignments," for other well-known TCP and UDP port assignments.

The port-to-program information maintained by Portmapper is called the portmap. Clients ask Portmapper about entries for servers on the network. Servers contact Portmapper to add or update entries to the portmap.

Contacting Portmapper

To find the port of a remote program, the client sends an RPC to well-known port 111 of the server's host. If Portmapper has a portmap entry for the remote program, Portmapper provides the port number in a return RPC. The client then requests the remote program by sending an RPC to the port number provided by Portmapper.

Clients can save port numbers of recently called remote programs to avoid having to contact Portmapper for each request to a server.

Target Assistance

Portmapper offers a program to assist clients in contacting server programs. If the client sends Portmapper an RPC with the target program number, version number, procedure number, and arguments, Portmapper searches the portmap for an entry, and passes the client's message to the server. When the target server returns the information to Portmapper, the information is passed to the client, along with the port number of the remote program. The client can then contact the server directly.

enum clnt_stat Structure

The enum clnt_stat structure is defined in the <RPC\CLNT.H> file.

RPCs frequently return enum clnt_stat information. The following is the format of the enum clnt_stat structure:

```
enum clnt_stat  {
   RPC_SUCCESS=0,             /* call succeeded */
   /*
    * local errors
    */
   RPC_CANTENCODEARGS=1,      /* can't encode arguments */
   RPC_CANTDECODERES=2,       /* can't decode results */
   RPC_CANTSEND=3,            /* failure in sending call */
   RPC_CANTRECV=4,            /* failure in receiving result */
   RPC_TIMEDOUT=5,            /* call timed out */
   /*
    * remote errors
    */
   RPC_VERSMISMATCH=6,        /* RPC versions not compatible */
   RPC_AUTHERROR=7,           /* authentication error */
   RPC_PROGUNAVAIL=8,         /* program not available */
   RPC_PROGVERSMISMATCH=9,    /* program version mismatched */
   RPC_PROCUNAVAIL=10,        /* procedure unavailable */
   RPC_CANTDECODEARGS=11,     /* decode arguments error */
   RPC_SYSTEMERROR=12,        /* generic "other problem" */
   /*
    * callrpc errors
    */
   RPC_UNKNOWNHOST=13,        /* unknown host name */
   /*
    * create errors
    */
   RPC_PMAPFAILURE=14,         /* the pmapper failed in its call */
   RPC_PROGNOTREGISTERED=15,  /* remote program is not registered */
   /*
    * unspecified error
    */
   RPC_FAILED=16
              };
```

Remote Procedure Call Library

To use the RPCs described in this chapter, you must have the following header files, contained in the <*TCPBASE*>\INCLUDE directory, available on your system:

- RPC\AUTH.H
- RPC\A_UNIX.H
- RPC\CLNT.H
- RPC\P_CLNT.H
- RPC\P_PROT.H
- RPC\P_RMT.H
- RPC\RPC.H
- RPC\R_MSG.H
- RPC\TYPES.H
- RPC\SVC.H
- RPC\SVC_AUTH.H
- RPC\XDR.H

The RPC routines are contained in the SUNRPC.LIB file in the <*TCPBASE*>\LIB directory. You must also have the TCPIP.LIB file in your <*TCPBASE*>\LIB directory.

You should put the following statement at the top of any file using RPC code:

```
#include <rpc\rpc.h>
```

For a summary of each remote procedure call supported by TCP/IP for DOS, see Appendix F, "Remote Procedure Call Quick Reference."

Porting

The IBM DOS RPC implementation differs from the Sun Microsystems RPC implementation, because functions that rely on file descriptor structures are not supported by the IBM DOS RPC implementation.

Compiling and Linking

The following steps describe how to compile and link programs using the RPC API's with Microsoft C Version 5.10.

Note: In the following examples, *model* refers to the memory model you will use to compile your program: L for large model, S for small model, M for medium model, or C for compact model.

1. Include the <*TCPBASE*>\INCLUDE directory at the beginning of the INCLUDE environment variable so that the C compiler finds the appropriate header files. You can set this interactively or you can include it in the AUTOEXEC.BAT file.

   For example, if the INCLUDE environment variable previously read:

   ```
   SET INCLUDE=C:\MSC\INCLUDE
   ```

   you would change it to read:

   ```
   SET INCLUDE=<TCPBASE>\INCLUDE;C:\MSC\INCLUDE
   ```

2. To compile your program, enter the following command:

   ```
   cl \c \Amodel myprog.c
   ```

3. To create an executable program, enter the following command:

   ```
   link /stack:6144 myproj.obj,,,<TCPBASE>\LIB\model\sunrpc.lib+
   <TCPBASE>\LIB\model\tcpip.lib;
   ```

Remote Procedure and eXternal Data Representation Calls

This section provides the syntax, parameters, and other appropriate information for each remote procedure and external data representation call supported by TCP/IP for DOS.

auth_destroy()

```
#include <rpc\rpc.h>

void
auth_destroy(auth)
AUTH *auth;
```

Parameter	Description
auth	A pointer to authentication information.

Description: The auth_destroy() call deletes the authentication information for *auth*. Once this procedure is called, *auth* is undefined.

See Also: authnone_create(), authunix_create(), authunix_create_default().

authnone_create()

```
#include <rpc\rpc.h>

AUTH *
authnone_create()
```

Description: The authnone_create() call creates and returns an RPC authentication handle. The handle passes the NULL authentication on each call.

See Also: auth_destroy(), authunix_create(), authunix_create_default().

authunix_create()

```
#include <rpc\rpc.h>

AUTH *
authunix_create(host, uid, gid, len, aup_gids)
char *host;
int uid;
int gid;
int len;
int *aup_gids;
```

Parameter	Description
host	A pointer to the symbolic name of the host where the desired server is located.
uid	The user's user ID.
gid	The user's group ID.
len	The length of the information pointed to by *aup_gids*.
aup_gids	A pointer to an array of groups to which the user belongs.

Description: The authunix_create() call creates and returns an authentication handle that contains UNIX-based authentication information.

See Also: auth_destroy(), authnone_create(), authunix_create_default().

authunix_create_default()

```
#include <rpc\rpc.h>

AUTH *
authunix_create_default()
```

Description: The authunix_create_default() call calls authunix_create() with default parameters.

See Also: auth_destroy(), authnone_create(), authunix_create().

callrpc()

```
#include <rpc\rpc.h>

enum clnt_stat
callrpc(host, prognum, versnum, procnum, inproc, in, outproc, out)
char *host;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
```

Parameter	Description
host	A pointer to the symbolic name of the host where the desired server is located.
prognum	Identifies the program number of the remote procedure.
versnum	Identifies the version number of the remote procedure.
procnum	Identifies the procedure number of the remote procedure.
inproc	The XDR procedure used to encode the arguments of the remote procedure.
in	A pointer to the arguments of the remote procedure.
outproc	The XDR procedure used to decode the results of the remote procedure.
out	A pointer to the results of the remote procedure.

Description: The callrpc() call calls the remote procedure described by *prognum*, *versnum*, and *procnum* running on the *host* system. callrpc() encodes and decodes the parameters for transfer.

Notes:

1. clnt_perrno() can be used to translate the return code into messages.
2. callrpc() cannot call the procedure xdr_enum. See "xdr_enum()" on page 159 for more information.
3. This procedure uses UDP as its transport layer. See "clntudp_create()" on page 123 for more information.

Return Values: RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

See Also: clnt_call(), clnt_perrno(), clntudp_create().

clnt_broadcast()

```
#include <rpc\rpc.h>

enum clnt_stat
clnt_broadcast(prognum, versnum, procnum, inproc, in, outproc, out, eachresult)
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
caddr_t in;
xdrproc_t outproc;
caddr_t out;
resultproc_t eachresult;
```

Parameter	Description
prognum	Identifies the program number of the remote procedure.
versnum	Identifies the version number of the remote procedure.
procnum	Identifies the procedure number of the remote procedure.
inproc	The XDR procedure used to encode the arguments of the remote procedure.
in	A pointer to the arguments of the remote procedure.
outproc	The XDR procedure used to decode the results of the remote procedure.
out	A pointer to the results of the remote procedure.
eachresult	The procedure called after each response. **Note:** resultproc_t is a type definition: `typedef bool_t (*resultproc_t) ();`

Description: The clnt_broadcast() call broadcasts the remote procedure described by *prognum*, *versnum*, and *procnum* to all locally connected broadcast networks. Each time clnt_broadcast() receives a response it calls eachresult(). The format of eachresult() is:

```
#include <in.h>
#include <rpc\types.h>

bool_t eachresult(out, addr)
char *out;
struct sockaddr_in *addr;
```

Parameter	Description
out	Has the same function as it does for clnt_broadcast(), except that the output of the remote procedure is decoded.
addr	Points to the address of the machine that sent the results.

Return Values: If eachresult() returns 0, clnt_broadcast() waits for more replies; otherwise, eachresult() returns the appropriate status.

Note: Broadcast sockets are limited in size to the maximum transfer unit of the data link.

See Also: callrpc(), clnt_call().

clnt_call()

```
#include <rpc\rpc.h>

enum clnt_stat
clnt_call(clnt, procnum, inproc, in, outproc, out, tout)
CLIENT *clnt;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
```

Parameter	Description
clnt	Points to a client handle that was previously obtained using clnttcp_create(), or clntudp_create().
procnum	Identifies the remote procedure number.
inproc	The XDR procedure used to encode *procnum*'s arguments.
in	Points to the remote procedure's arguments.
outproc	The XDR procedure used to decode the remote procedure's results.
out	Points to the remote procedure's results.
tout	The time allowed for the server to respond in units of 0.1 seconds.

Description: The clnt_call() call calls the remote procedure (*procnum*) associated with the client handle (*clnt*).

Return Values: RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

See Also: callrpc(), clnt_perror(), clnttcp_create(), clntudp_create().

clnt_destroy()

```
#include <rpc\rpc.h>

void
clnt_destroy(clnt)
CLIENT *clnt;
```

Parameter	Description
clnt	Points to a client handle that was previously created using clntudp_create() or clnttcp_create().

Description: The clnt_destroy() call deletes a client RPC transport handle. This procedure involves the deallocation of private data resources, including *clnt*. Once this procedure is used, *clnt* is undefined. Open sockets associated with *clnt* must be closed.

See Also: clnttcp_create(), clntudp_create().

clnt_freeres()

```
#include <rpc\rpc.h>

bool_t
clnt_freeres(clnt, outproc, out)
CLIENT *clnt;
xdrproc_t outproc;
char *out;
```

Parameter	Description
clnt	Points to a client handle that was previously obtained using clnttcp_create() or clntudp_create().
outproc	The XDR procedure used to decode the remote procedure's results.
out	Points to the results of the remote procedure.

Description: The clnt_freeres() call deallocates any resources that were assigned by the system to decode the results of an RPC.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: clnttcp_create(), clntudp_create().

clnt_geterr()

```
#include <rpc\rpc.h>

void
clnt_geterr(clnt, errp)
CLIENT *clnt;
struct rpc_err *errp;
```

Parameter	Description
clnt	Points to a client handle that was previously obtained using clnttcp_create() or clntudp_create().
errp	Points to the address into which the error structure is copied.

Description: The clnt_geterr() call copies the error structure from the client handle to the structure at address *errp*.

See Also: clnt_call(), clnt_pcreateerror(), clnt_perrno(), clnt_perror(), clnttcp_create(), clntudp_create().

clnt_pcreateerror()

```
#include <rpc\rpc.h>

void
clnt_pcreateerror(s)
char *s;
```

Parameter	Description
s	Points to a string that is to be printed in front of the message. The string is followed by a colon.

Description: The clnt_pcreateerror() call writes a message to the standard error device, indicating why a client handle cannot be created. This procedure is used after the clnttcp_create() or clntudp_create() calls fail.

See Also: clnt_geterr(), clnt_perrno(), clnt_perror(), clnttcp_create(), clntudp_create().

clnt_perrno()

```
#include <rpc\rpc.h>

void
clnt_perrno(stat)
enum clnt_stat stat;
```

Parameter	Description
stat	The client status.

Description: The clnt_perrno() call writes a message to the standard error device corresponding to the condition indicated by *stat*. This procedure should be used after callrpc() if there is an error.

See Also: callrpc(), clnt_geterr(), clnt_pcreateerror(), clnt_perror().

clnt_perror()

```
#include <rpc\rpc.h>

void
clnt_perror(clnt, s)
CLIENT *clnt;
char *s;
```

Parameter	Description
clnt	Points to a client handle that was previously obtained using clntudp_create() or clnttcp_create().
s	Points to a string that is to be printed in front of the message. The string is followed by a colon.

Description: The clnt_perror() call writes a message to the standard error device, indicating why an RPC failed. This procedure should be used after clnt_call() if there is an error.

See Also: clnt_call(), clnt_geterr(), clnt_pcreateerror(), clnt_perrno(), clnttcp_create(), clntudp_create().

clnttcp_create()

```
#include <rpc\rpc.h>

CLIENT *
clnttcp_create(addr, prognum, versnum, sockp, sendsz, recvsz)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
int *sockp;
u_int sendsz;
u_int recvsz;
```

Parameter	Description
addr	Points to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving.
prognum	The remote program number.
versnum	The version number of the remote program.
sockp	Points to the socket. If *sockp* is RPC_ANYSOCK, then this routine opens a new socket and sets *sockp*.
sendsz	The size of the send buffer. Specify 0 to choose the default.
recvsz	The size of the receive buffer. Specify 0 to choose the default.

Description: The clnttcp_create() call creates an RPC client transport handle for the remote program specified by (*prognum, versnum*). The client uses TCP as the transport layer.

Return Values: NULL indicates failure.

See Also: clnt_destroy(), clnt_pcreateerror(), clntudp_create().

clntudp_create()

```
#include <rpc\rpc.h>
#include <netdb.h>
#include <netlib.h>

CLIENT *
clntudp_create(addr, prognum, versnum, wait, sockp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
struct timeval wait;
int *sockp;
```

Parameter	Description
addr	Points to the internet address of the remote program. If *addr* points to a port number of 0, *addr* is set to the port on which the remote program is receiving. The remote portmap service is used for this.
prognum	The remote program number.
versnum	The version number of the remote program.
wait	UDP resends the call request at intervals of *wait* time, until either a response is received or the call times out. The time-out length is set using the clnt_call() procedure.
sockp	Points to the socket. If *sockp* is RPC_ANYSOCK, this routine opens a new socket and sets *sockp*.

Description: The clntudp_create() call creates a client transport handle for the remote program (*prognum*) with version (*versnum*). UDP is used as the transport layer.

Note: This procedure should not be used with procedures that use large arguments or return large results. UDP RPC messages can only contain 2 KB of encoded data.

Return Values: NULL indicates failure.

See Also: clnt_destroy(), clnt_pcreateerror(), clnttcp_create().

get_myaddress()

```
#include <rpc\rpc.h>

void
get_myaddress(addr)
struct sockaddr_in *addr;
```

Parameter	Description
addr	Points to the location where the local internet address is placed.

Description: The get_myaddress() call puts the local host's internet address into *addr*. The port number (*addr—>sin_port*) is set to htons (PMAPPORT), which is 111.

pmap_getmaps()

```
#include <rpc\rpc.h>

struct pmaplist *
pmap_getmaps(addr)
struct sockaddr_in *addr;
```

Parameter	Description
addr	Points to the internet address of the foreign host.

Description: The pmap_getmaps() call returns a list of current program-to-port mappings on the foreign host specified by *addr*.

See Also: pmap_getport(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_getport()

```
#include <rpc\rpc.h>

u_short
pmap_getport(addr, prognum, versnum, protocol)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_int protocol;
```

Parameter	Description
addr	Points to the internet address of the foreign host.
prognum	The program number to be mapped.
versnum	The version number of the program to be mapped.
protocol	The transport protocol used by the program.

Description: The pmap_getport() call returns the port number associated with the remote program (*prognum*), the version (*versnum*), and the transport protocol (*protocol*).

Return Values: The value 0 indicates that the mapping does not exist or that the remote portmap could not be contacted. If Portmapper cannot be contacted, *rpc_createerr* contains the RPC status.

See Also: pmap_getmaps(), pmap_rmtcall(), pmap_set(), pmap_unset().

pmap_rmtcall()

```
#include <rpc\rpc.h>
#include <netdb.h>
#include <netlib.h>

enum clnt_stat
pmap_rmtcall(addr, prognum, versnum, procnum, inproc, in, outproc, out, tout, portp)
struct sockaddr_in *addr;
u_long prognum;
u_long versnum;
u_long procnum;
xdrproc_t inproc;
char *in;
xdrproc_t outproc;
char *out;
struct timeval tout;
u_long *portp;
```

Parameter	Description
addr	Points to the internet address of the foreign host.
prognum	The remote program number.
versnum	The version number of the remote program.
procnum	Identifies the procedure to be called.
inproc	The XDR procedure used to encode the arguments of the remote procedure.
in	Points to the arguments of the remote procedure.
outproc	The XDR procedure used to decode the results of the remote procedure.
out	Points to the results of the remote procedure.
tout	The time-out period for the remote request.
portp	If the call from the remote portmap service is successful, *portp* contains the port number of the triple (*prognum*, *versnum*, *procnum*).

Description: The pmap_rmtcall() call instructs Portmapper to make an RPC call to a procedure on that host, on your behalf. This procedure should be used only for ping-type functions.

Return Values: RPC_SUCCESS indicates success; otherwise, an error has occurred. The results of the remote procedure call are returned to *out*.

See Also: pmap_getmaps(), pmap_getport(), pmap_set(), pmap_unset().

pmap_set()

```
#include <rpc\rpc.h>

bool_t
pmap_set(prognum, versnum, protocol, port)
u_long prognum;
u_long versnum;
u_int protocol;
u_short port;
```

Parameter	Description
prognum	The local program number.
versnum	The version number of the local program.
protocol	The transport protocol used by the local program.
port	The port to which the local program is mapped.

Description: The pmap_set() call sets the mapping of the program (specified by *prognum*, *versnum*, and *protocol*) to *port* on the local machine. This procedure is automatically called by the svc_register() procedure.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: pmap_getmaps(), pmap_getport(), pmap_rmtcall(), pmap_unset().

pmap_unset()

```
#include <rpc\rpc.h>

bool_t
pmap_unset(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameter	Description
prognum	The local program number.
versnum	The version number of the local program.

Description: The pmap_unset() call removes the mappings associated with *prognum* and *versnum* on the local machine. All ports for each transport protocol currently mapping the *prognum* and *versnum* are removed from the portmap service.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: pmap_getmaps(), pmap_getport(), pmap_rmtcall(), pmap_set().

registerrpc()

```
#include <rpc\rpc.h>

int
registerrpc(prognum, versnum, procnum, procname, inproc, outproc)
u_long prognum;
u_long versnum;
u_long procnum;
char *(*procname) ();
xdrproc_t inproc;
xdrproc_t outproc;
```

Parameter	Description
prognum	The program number to register.
versnum	The version number to register.
procnum	The procedure number to register.
procname	The procedure that is called when the registered program is requested. *procname* must accept a pointer to its arguments, and return a static pointer to its results.
inproc	The XDR routine used to decode the arguments.
outproc	The XDR routine that encodes the results.

Description: The registerrpc() call registers a procedure (*prognum*, *versnum*, *procnum*) with the local Portmapper, and creates a control structure to remember the server procedure and its XDR routine. The control structure is used by svc_run(). When a request arrives for the program (*prognum, versnum, procnum*), the procedure *procname* is called. Procedures registered using registerrpc() are accessed using the UDP transport layer.

Note: xdr_enum() cannot be used as an argument to registerrpc(). See "xdr_enum()" on page 159 for more information.

Return Values: The value 0 indicates success; the value –1 indicates an error.

See Also: svc_register(), svc_run().

rpc_createerr

```
#include <rpc\rpc.h>

struct  rpc_createerr rpc_createerr;
```

Description: A global variable that is set when any RPC client creation routine fails. Use clnt_pcreateerror() to print the message.

svc_destroy()

```
#include <rpc\rpc.h>

void
svc_destroy(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The svc_destroy() call deletes the RPC service transport handle *xprt*, which becomes undefined after this routine is called.

See Also: svctcp_create(), svcudp_create().

svc_fds

```
int svc_fds;
```

Description: A global variable reflecting the RPC service-side read file descriptor bit mask, but limited to 16 descriptors.

See Also: svc_getreq().

svc_freeargs()

```
#include <rpc\rpc.h>

bool_t
svc_freeargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameter	Description
xprt	Points to the service transport handle.
inproc	The XDR routine used to decode the arguments.
in	Points to the input arguments.

Description: The svc_freeargs() call frees storage allocated to decode the arguments received by svc_getargs().

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: svc_getargs().

svc_getargs()

```
#include <rpc\rpc.h>

bool_t
svc_getargs(xprt, inproc, in)
SVCXPRT *xprt;
xdrproc_t inproc;
char *in;
```

Parameter	Description
xprt	Points to the service transport handle.
inproc	The XDR routine used to decode the arguments.
in	Points to the decoded arguments.

Description: The svc_getargs() call uses the XDR routine *inproc* to decode the arguments of an RPC request associated with the RPC service transport handle *xprt*. The results are placed at address *in*.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: svc_freeargs().

svc_getcaller()

```
#include <rpc\rpc.h>

struct sockaddr_in *
svc_getcaller(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: This macro obtains the socket address of the client associated with the service transport handle *xprt*.

svc_getreq()

```
#include <rpc\rpc.h>

void
svc_getreq(svc_fds)
int svc_fds;
```

Parameter	Description
svc_fds	Service side read file descriptor bit mask.

Description: The svc_getreq() call is used rather than svc_run() to implement asynchronous event processing. The routine returns control to the program when all sockets in the *socks* array have been serviced.

See Also: svc_run().

svc_register()

```
#include <rpc\rpc.h>
#include <netdb.h>
#include <netlib.h>

bool_t
svc_register(xprt, prognum, versnum, dispatch, protocol)
SVCXPRT *xprt;
u_long prognum;
u_long versnum;
void (*dispatch) ();
int protocol;
```

Parameter	Description
xprt	Points to the service transport handle.
prognum	The program number to be registered.
versnum	The version number of the program to be registered.

dispatch	The dispatch routine associated with *prognum* and *versnum*.
	
	The structure of the dispatch routine:
	
	`dispatch(request, xprt)`
	`struct svc_req *request;`
	`SVCXPRT *xprt;`

protocol	The protocol used. The value is generally one of the following: • 0 (zero) • IPPROTO_UDP • IPPROTO_TCP When a value of 0 is used, the service is not registered with Portmapper.

Description: The svc_register() call associates the program described by (*prognum, versnum*) with the service dispatch routine *dispatch*.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: registerrpc(), svc_unregister(), xprt_register().

svc_run()

```
#include <rpc\rpc.h>

void
svc_run()
```

Description: The svc_run() call does not return control. It accepts RPC requests, and calls the appropriate service using svc_getreq().

See Also: registerrpc(), svc_getreq().

svc_sendreply()

```
#include <rpc\rpc.h>

bool_t
svc_sendreply(xprt, outproc, out)
SVCXPRT *xprt;
xdrproc_t outproc;
char *out;
```

Parameter	Description
xprt	Points to the caller's transport handle.
outproc	The XDR procedure used to encode the results.
out	Points to the results.

Description: The svc_sendreply() call is called by the service dispatch routine to send the results of the call to the caller.

Return Values: The value 1 indicates success; the value 0 indicates an error.

svc_unregister()

```
#include <rpc\rpc.h>

void
svc_unregister(prognum, versnum)
u_long prognum;
u_long versnum;
```

Parameter	Description
prognum	The program number that is removed.
versnum	The version number of the program that is removed.

Description: The svc_unregister() call removes all local mappings of (*prognum*, *versnum*) to dispatch routines and (*prognum*, *versnum*, *) to port numbers.

See Also: svc_register().

svcerr_auth()

```
#include <rpc\rpc.h>

void
svcerr_auth(xprt, why)
SVCXPRT *xprt;
enum auth_stat why;
```

Parameter	Description
xprt	Points to the service transport handle.
why	The reason the call is refused.

Description: The svcerr_auth() call is called by a service dispatch routine that refuses to execute an RPC request, because of authentication errors.

See Also: svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_decode()

```
#include <rpc\rpc.h>

void
svcerr_decode(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The svcerr_decode() call is called by a service dispatch routine that cannot decode its parameters.

See Also: svcerr_auth(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_noproc()

```
#include <rpc\rpc.h>

void
svcerr_noproc(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The svcerr_noproc() call is called by a service dispatch routine that does not implement the requested procedure.

See Also: svcerr_auth(), svcerr_decode(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_noprog()

```
#include <rpc\rpc.h>

void
svcerr_noprog(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The svcerr_noprog() call is used when the desired program is not registered.

See Also: svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_progvers()

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt, low_vers, high_vers)
SVCXPRT *xprt;
u_long low_vers;
u_long high_vers;
```

Parameter	Description
xprt	Points to the service transport handle.
low_vers	The low version number that did not match.
high_vers	The high version number that did not match.

Description: The svcerr_progvers() call is called when the version numbers of two RPC programs do not match. The low version number and the high version number are the two version numbers that do not match. One number is the version number of the client. The other number is the version number of the server.

See Also: svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr(), svcerr_weakauth().

svcerr_systemerr()

```
#include <rpc\rpc.h>

void
svcerr_systemerr(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The svcerr_systemerr() call is called by a service dispatch routine when it detects a system error that is not handled by the protocol.

See Also: svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_weakauth().

svcerr_weakauth()

```
#include <rpc\rpc.h>

void
svcerr_progvers(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Note: This is the equivalent of: `svcerr_auth(xprt, AUTH_TOOWEAK)`.

Description: The svcerr_weakauth() call is called by a service dispatch routine that cannot execute an RPC, because of correct but weak authentication parameters.

See Also: svcerr_auth(), svcerr_decode(), svcerr_noproc(), svcerr_noprog(), svcerr_progvers(), svcerr_systemerr().

svctcp_create()

```
#include <rpc\rpc.h>

SVCXPRT *
svctcp_create(sock, send_buf_size, recv_buf_size)
int sock;
u_int send_buf_size;
u_int recv_buf_size;
```

Parameter	Description
sock	The socket descriptor. If *sock* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local TCP port, it is bound to an arbitrary port.
send_buf_size	The size of the send buffer. Specify 0 to choose the default.
recv_buf_size	The size of the receive buffer. Specify 0 to choose the default.

Description: The svctcp_create() call creates a TCP-based service transport to which it returns a pointer. xprt→xp_sock contains the transport's socket descriptor. xprt→xp_port contains the transport's port number.

Return Values: NULL indicates failure.

See Also: svc_destroy(), svcudp_create().

svcudp_create()

```
#include <rpc\rpc.h>

SVCXPRT *
svcudp_create(sockp)
int sockp;
```

Parameter	Description
sockp	Points to the socket associated with the service transport handle. If *sockp* is RPC_ANYSOCK, a new socket is created. If the socket is not bound to a local UDP port, it is bound to an arbitrary port.

Warning: UDP can only transmit 2 KB of data for each packet.

Description: The svcudp_create() call creates a UDP-based service transport to which it returns a pointer. xprt→xp_sock contains the transport's socket descriptor. xprt→xp_port contains the transport's port number.

Return Values: NULL indicates failure.

See Also: svc_destroy(), svctcp_create().

xdr_accepted_reply()

```
#include <rpc\rpc.h>

bool_t
xdr_accepted_reply(xdrs, ar)
XDR *xdrs;
struct accepted_reply *ar;
```

Parameter	Description
xdrs	Points to an XDR stream.
ar	Points to the reply to be represented.

Description: The xdr_accepted_reply() call translates RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_array()

```
#include <rpc\rpc.h>

bool_t
xdr_array(xdrs, arrp, sizep, maxsize, elsize, elproc)
XDR *xdrs;
char **arrp;
u_int *sizep;
u_int maxsize;
u_int elsize;
xdrproc_t elproc;
```

Parameter	Description
xdrs	Points to an XDR stream.
arrp	The address of the pointer to the array.
sizep	Points to the element count of the array.
maxsize	The maximum number of elements accepted.
elsize	The size of each of the array's elements, found using sizeof().
elproc	The XDR routine that translates an individual array element.

Description: The xdr_array() call translates between an array and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_authunix_parms()

```
#include <rpc\rpc.h>

bool_t
xdr_authunix_parms(xdrs, aupp)
XDR *xdrs;
struct authunix_parms *aupp;
```

Parameter	Description
xdrs	Points to an XDR stream.
aupp	Points to the authentication information.

Description: The xdr_authunix_parms() call translates UNIX-based authentication information.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_bool()

```
#include <rpc\rpc.h>

bool_t
xdr_bool(xdrs, bp)
XDR *xdrs;
bool_t *bp;
```

Parameter	Description
xdrs	Points to an XDR stream.
bp	Points to the boolean.

Description: The xdr_bool() call translates between booleans and their external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_bytes()

```
#include <rpc\rpc.h>

bool_t
xdr_bytes(xdrs, sp, sizep, maxsize)
XDR *xdrs;
char **sp;
u_int *sizep;
u_int maxsize;
```

Parameter	Description
xdrs	Points to an XDR stream.
sp	Points to a pointer to the byte string.
sizep	Points to the byte string size.
maxsize	The maximum size of the byte string.

Description: The xdr_bytes() call translates between byte strings and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_callhdr()

```
#include <rpc\rpc.h>

void
xdr_callhdr(xdrs, chdr)
XDR *xdrs;
struct rpc_msg *chdr;
```

Parameter	Description
xdrs	Points to an XDR stream.
chdr	Points to the call header.

Description: The xdr_callhdr() call translates an RPC message header into XDR format.

xdr_callmsg()

```
#include <rpc\rpc.h>

bool_t
xdr_callmsg(xdrs, cmsg)
XDR *xdrs;
struct rpc_msg *cmsg;
```

Parameter	Description
xdrs	Points to an XDR stream.
cmsg	Points to the call message.

Description: The xdr_callmsg() call translates RPC messages (header and authentication; not argument data).

xdr_double()

```
#include <rpc\rpc.h>

bool_t
xdr_double(xdrs, dp)
XDR *xdrs;
double *dp;
```

Parameter	Description
xdrs	Points to an XDR stream.
dp	Points to a double-precision number.

Description: The xdr_double() call translates between C language double-precision numbers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_enum()

```
#include <rpc\rpc.h>

bool_t
xdr_enum(xdrs, ep)
XDR *xdrs;
enum_t *ep;
```

Parameter	Description
xdrs	Points to an XDR stream.
ep	Points to the enumerated number.

Description: The xdr_enum() call translates between C language enumerated groups and their external representation. When calling the procedures callrpc() and registerrpc(), a stub procedure must be created for both the server and the client before the procedure of the application program using xdr_enum(). This procedure should look like the following:

```
#include <rpc\rpc.h>

void
static xdr_enum_t(xdrs, ep)
XDR *xdrs;
enum_t *ep;
{
        xdr_enum(xdrs, ep)
}
```

The xdr_enum_t procedure is used as the *inproc* and *outproc* in both the client and server RPCs.

For example, an RPC client would contain the following lines:

```
.
.
.
error =
callrpc(argv[1],ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,&innumber,xdr_enum_t,
                &outnumber);
.
.
.
```

An RPC server would contain the following line:

```
.
.
.
registerrpc(ENUMRCVPROG,VERSION,ENUMRCVPROC,xdr_enum_t,xdr_enum_t);
.
.
.
```

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_float()

```
#include <rpc\rpc.h>

bool_t
xdr_float(xdrs, fp)
XDR *xdrs;
float *fp;
```

Parameter	Description
xdrs	Points to an XDR stream.
fp	Points to the floating-point number.

Description: The xdr_float() call translates between C language floating-point numbers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_inline()

```
#include <rpc\rpc.h>

long *
xdr_inline(xdrs, len)
XDR *xdrs;
int len;
```

Parameter	Description
xdrs	Points to an XDR stream.
len	The byte length of the desired buffer.

Description: The xdr_inline() call returns a pointer to a continuous piece of the XDR stream's buffer. The value is `long *` rather than `char *`, because the external data representation of any object is always an integer multiple of 32 bits.

Return Values: The value 1 indicates success; the value 0 indicates an error.

Note: xdr_inline() may return NULL if there is not sufficient space in the stream buffer to satisfy the request.

xdr_int()

```
#include <rpc\rpc.h>

bool_t
xdr_int(xdrs, ip)
XDR *xdrs;
int *ip;
```

Parameter	Description
xdrs	Points to an XDR stream.
ip	Points to the integer.

Description: The xdr_int() call translates between C language integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_long()

```
#include <rpc\rpc.h>

bool_t
xdr_long(xdrs, lp)
XDR *xdrs;
long *lp;
```

Parameter	Description
xdrs	Points to an XDR stream.
lp	Points to the long integer.

Description: The xdr_long() call translates between C language long integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_opaque()

```
#include <rpc\rpc.h>

bool_t
xdr_opaque(xdrs, cp, cnt)
XDR *xdrs;
char *cp;
u_int cnt;
```

Parameter	Description
xdrs	Points to an XDR stream.
cp	Points to the opaque object.
cnt	The size of the opaque object.

Description: The xdr_opaque() call translates between fixed-size opaque data and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_opaque_auth()

```
#include <rpc\rpc.h>

bool_t
xdr_opaque_auth(xdrs, ap)
XDR *xdrs;
struct opaque_auth *ap;
```

Parameter	Description
xdrs	Points to an XDR stream.
ap	Points to the opaque authentication information.

Description: The xdr_opaque_auth() call translates RPC message authentications.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_pmap()

```
#include <rpc\rpc.h>

bool_t
xdr_pmap(xdrs, regs)
XDR *xdrs;
struct pmap *regs;
```

Parameter	Description
xdrs	Points to an XDR stream.
regs	Points to the portmap parameters.

Description: The xdr_pmap() call translates an RPC procedure identification, such as is used in calls to Portmapper.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_pmaplist()

```
#include <rpc\rpc.h>

bool_t
xdr_pmaplist(xdrs, rp)
XDR *xdrs;
struct pmaplist **rp;
```

Parameter	Description
xdrs	Points to an XDR stream.
rp	Points to a pointer to the portmap data array.

Description: The xdr_pmaplist() call translates a variable number of RPC procedure identifications, such as Portmapper creates.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_reference()

```
#include <rpc\rpc.h>

bool_t
xdr_reference(xdrs, pp, size, proc)
XDR *xdrs;
char **pp;
u_int size;
xdrproc_t proc;
```

Parameter	Description
xdrs	Points to an XDR stream.
pp	Points to a pointer.
size	The size of the target.
proc	The XDR procedure that translates an individual element of the type addressed by the pointer.

Description: The xdr_reference() call provides pointer chasing within structures.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_rejected_reply()

```
#include <rpc\rpc.h>

bool_t
xdr_rejected_reply(xdrs, rr)
XDR *xdrs;
struct rejected_reply *rr;
```

Parameter	Description
xdrs	Points to an XDR stream.
rr	Points to the rejected reply.

Description: The xdr_rejected_reply() call translates RPC reply messages.

xdr_replymsg()

```
#include <rpc\rpc.h>

bool_t
xdr_replymsg(xdrs, rmsg)
XDR *xdrs;
struct rpc_msg *rmsg;
```

Parameter	Description
xdrs	Points to an XDR stream.
rmsg	Points to the reply message.

Description: The xdr_replymsg() call translates RPC reply messages.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_short()

```
#include <rpc\rpc.h>

bool_t
xdr_short(xdrs, sp)
XDR *xdrs;
short *sp;
```

Parameter	Description
xdrs	Points to an XDR stream.
sp	Points to the short integer.

Description: The xdr_short() call translates between C language short integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_string()

```
#include <rpc\rpc.h>

bool_t
xdr_string(xdrs, sp, maxsize)
XDR *xdrs;
char **sp;
u_int maxsize;
```

Parameter	Description
xdrs	Points to an XDR stream.
sp	Points to a pointer to the string.
maxsize	The maximum size of the string.

Description: The xdr_string() call translates between C language strings and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_u_int()

```
#include <rpc\rpc.h>

bool_t
xdr_u_int(xdrs, up)
XDR *xdrs;
unsigned *up;
```

Parameter	Description
xdrs	Points to an XDR stream.
up	Points to the unsigned integer.

Description: The xdr_u_int() call translates between C language unsigned integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_u_long()

```
#include <rpc\rpc.h>

bool_t
xdr_u_long(xdrs, ulp)
XDR *xdrs;
u_long *ulp;
```

Parameter	Description
xdrs	Points to an XDR stream.
ulp	Points to the unsigned long integer.

Description: The xdr_u_long() call translates between C language unsigned long integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_u_short()

```
#include <rpc\rpc.h>

bool_t
xdr_u_short(xdrs, usp)
XDR *xdrs;
u_short *usp;
```

Parameter	Description
xdrs	Points to an XDR stream.
usp	Points to the unsigned short integer.

Description: The xdr_u_short() call translates between C language unsigned short integers and their external representations.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_union()

```
#include <rpc\rpc.h>

bool_t
xdr_union(xdrs, dscmp, unp, choices, dfault)
XDR *xdrs;
int *dscmp;
char *unp;
struct xdr_discrim *choices;
xdrproc_t dfault;
```

Parameter	Description
xdrs	Points to an XDR stream.
dscmp	Points to the union's discriminant.
unp	Points to the union.
choices	Points to an array detailing the XDR procedure to use on each arm of the union.
dfault	The default XDR procedure to use.

Description: The xdr_union() call translates between a discriminated C language union and its external representation.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdr_void()

```
#include <rpc\rpc.h>

bool_t
xdr_void()
```

Description: The xdr_void() call may be passed to RPC routines that require a function parameter. It does not translate external representation and no function is performed.

Return Values: Always a value of 1.

xdr_wrapstring()

```
#include <rpc\rpc.h>

bool_t
xdr_wrapstring(xdrs, sp)
XDR *xdrs;
char **sp;
```

Parameter	Description
xdrs	Points to an XDR stream.
sp	Points to a pointer to the string.

Description: The xdr_wrapstring() call is the same as calling xdr_string() with the maximum size of an unsigned integer. It is useful because many RPC procedures implicitly invoke two-parameter XDR routines, and xdr_string() is a three-parameter routine.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdrmem_create()

```
#include <rpc\rpc.h>

void
xdrmem_create(xdrs, addr, size, op)
XDR *xdrs;
char *addr;
u_int size;
enum xdr_op op;
```

Parameter	Description
xdrs	Points to an XDR stream.
addr	Points to the memory location.
size	The maximum size of *addr*.
op	Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Description: The xdrmem_create() call initializes the XDR stream pointed to by *xdrs*. Data is written to, or read from, *addr*.

xdrrec_create()

```
#include <rpc\rpc.h>

void
xdrrec_create(xdrs, sendsize, recvsize, handle, readit, writeit)
XDR *xdrs;
u_int sendsize;
u_int recvsize;
char *handle;
int (*readit) ();
int (*writeit) ();
```

Parameter	Description
xdrs	Points to an XDR stream.
sendsize	The size of the send buffer. Specify 0 to choose the default.
recvsize	The size of the receive buffer. Specify 0 to choose the default.
handle	The first parameter passed to *readit*() and *writeit*().
readit()	Called when a stream's input buffer is empty.
writeit()	Called when a stream's output buffer is full.

Description: The xdrrec_create() call initializes the XDR stream pointed to by *xdrs*.

Note: The *op* field must be set by the caller.

Warning: This XDR procedure implements an intermediate record string. Additional bytes in the XDR stream provide record boundary information.

xdrrec_endofrecord()

```
#include <rpc\rpc.h>

bool_t
xdrrec_endofrecord(xdrs, sendnow)
XDR *xdrs;
int sendnow;
```

Parameter	Description
xdrs	Points to an XDR stream.
sendnow	Specifies nonzero to write out data in the output buffer.

Description: The xdrrec_endofrecord() call can be invoked only on streams created by xdrrec_create(). Data in the output buffer is marked as a complete record.

Return Values: The value 1 indicates success; the value 0 indicates an error.

xdrrec_eof()

```
#include <rpc\rpc.h>

bool_t
xdrrec_eof(xdrs)
XDR *xdrs;
int empty;
```

Parameter	Description
xdrs	Points to an XDR stream.

Description: The xdrrec_eof() call can be invoked only on streams created by xdrrec_create().

Return Values: The value 1 indicates the current record has been consumed; the value 0 indicates continued input on the stream.

xdrrec_skiprecord()

```
#include <rpc\rpc.h>

bool_t
xdrrec_skiprecord(xdrs)
XDR *xdrs;
```

Parameter	Description
xdrs	Points to an XDR stream.

Description: The xdrrec_skiprecord() call can be invoked only on streams created by xdrrec_create(). The XDR implementation is instructed to discard the remaining data in the input buffer.

Return Values: The value 1 indicates success; the value 0 indicates an error.

See Also: xdrrec_create().

xdrstdio_create()

```
#include <rpc\rpc.h>
#include <stdio.h>

void
xdrstdio_create(xdrs, file, op)
XDR *xdrs;
FILE *file;
enum xdr_op op;
```

Parameter	Description
xdrs	Points to an XDR stream.
file	The file name for the input and output stream.
op	Determines the direction of the XDR stream (either XDR_ENCODE, XDR_DECODE, or XDR_FREE).

Description: The xdrstdio_create() call initializes the XDR stream pointed to by *xdrs*. Data is written to, or read from, *file*.

xprt_register()

```
#include <rpc\rpc.h>

void
xprt_register(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The xprt_register() call registers service transport handles with the RPC service package. This routine also modifies the global variable *svc_fds*.

See Also: svc_register().

xprt_unregister()

```
#include <rpc\rpc.h>

void
xprt_unregister(xprt)
SVCXPRT *xprt;
```

Parameter	Description
xprt	Points to the service transport handle.

Description: The xprt_unregister() call unregisters an RPC service transport handle. A transport handle should be unregistered with the RPC service package before it is destroyed. This routine also modifies the global variable *svc_fds*.

Chapter 5. File Transfer Protocol Application Programming Interface

Chapter 5. File Transfer Protocol Application Programming Interface

The File Transfer Protocol (FTP) Application Programming Interface (API) allows applications to have a client interface for file transfer. Applications written to this interface can communicate with multiple FTP servers at the same time. A maximum of 256 simultaneous connections are supported. The interface also allows third-party proxy transfers between pairs of FTP servers. Consecutive third-party transfers are allowed between any sequence of pairs of FTP servers.

The API tracks the servers to which an application is currently connected. When a new request for FTP service is requested, API checks whether there is a connection to the server. If the connection does not exist, it is established. If the server has dropped the connection since last use, it is reestablished.

FTP API Call Library

To use the FTP APIs described in this chapter, you must have the <FTPAPI.H> header file, contained in the <*TCPBASE*>\INCLUDE directory, available on your system.

The FTP API routines are contained in the FTPAPI.LIB file in the <*TCPBASE*>\LIB directory. You must also have the TCPIP.LIB file in your <*TCPBASE*>\LIB directory.

You should put the following statement at the top of any file using FTP API code:

```
#include <ftpapi.h>
```

For a summary of each FTP API call supported by TCP/IP for DOS, see Appendix G, "FTP API Quick Reference."

Compiling and Linking

The following steps describe how to compile and link programs using the FTP API's with Microsoft C Version 5.10.

Note: In the following examples, *model* refers to the memory model you use to compile your program: L for large model, S for small model, M for medium model, or C for compact model.

1. Include the <*TCPBASE*>\INCLUDE directory at the beginning of the INCLUDE environment variable so that the C compiler finds the appropriate header files. You can set this interactively or you can include it in the AUTOEXEC.BAT file.

For example, if the INCLUDE environment variable previously read:

```
SET INCLUDE=C:\MSC\INCLUDE
```

You would change it to read:

```
SET INCLUDE=<TCPBASE>\INCLUDE;C:\MSC\INCLUDE
```

2. To compile your program, enter the command:

```
cl /c /Os /Amodel myprog.c
```

3. To create an executable program, enter the following command:

```
link /stack:8192 /seg:200 myproj.obj,,,<TCPBASE>\LIB\model\ftpapi.lib+
<TCPBASE>\LIB\model\tcpip.lib;
```

Return Values (ftperrno)

Most functions return a value of −1 to indicate failure and a value of 0 to indicate success. Two functions do not return 0 and −1 values: ftplogoff(), which is of type void, and ping(), which returns an error code rather than storing the return value in *ftperrno*. When the value is −1, the global integer variable *ftperrno* is set to one of the following codes:

Return Value Code	Description
FTPSERVICE	Unknown service
FTPHOST	Unknown host
FTPSOCKET	Unable to obtain socket
FTPCONNECT	Unable to connect to server
FTPLOGIN	Login failed
FTPABORT	Transfer aborted
FTPLOCALFILE	Problem opening the local file
FTPDATACONN	Problem initializing data connection
FPTCOMMAND	Command failed
FTPPROXYTHIRD	Proxy server does not support third party
FTPNOPRIMARY	No primary connection for proxy transfer.

FTP API Calls

This section provides the syntax, parameters, and other appropriate information for each FTP API call supported by TCP/IP for DOS.

ftpappend()

```
#include <ftpapi.h>

int ftpappend(host, userid, passwd, acct, local, remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The local file name.
remote	The remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description: The ftpappend() call appends information to a remote file.

Example: The following is an example of the ftpappend() call.

```
int rc;
rc=ftpappend("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

In this example, the local ASCII file, `abc.doc`, is appended to the file, `new.doc`, in the current working directory at the host, `conypc`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpcd()

```
#include <ftpapi.h>

int ftpcd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct,
char *dir;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
dir	Specifies the new working directory.

Description: The ftpcd() call changes the current working directory.

Example: The following is an example of the ftpcd() call.

```
int rc;
rc=ftpcd("conypc","jason","ehgr1",NULL,"mydir");
```

In this example, the current working directory is changed to `mydir` on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpdelete()

```
#include <ftpapi.h>

int ftpdelete(host, userid, passwd, acct, name)
char *host;
char *userid;
char *passwd;
char *acct;
char *name;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
name	The file to be deleted.

Description: The ftpdelete() call deletes a file on a host.

Example: The following is an example of the ftpdelete() call.

```
int rc;
rc=ftpdelete("conypc","jason","ehgr1",NULL,"abc.1");
```

In this example, the file, `abc.1`, is deleted on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpdir()

```
#include <ftpapi.h>

int ftpdir(host, userid, passwd, acct, local, pattern,)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The local file name.
pattern	Specifies the file name or pattern of the files to be deleted on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning: * The asterisk shows that any character or group of characters can occupy that position in the pattern. ? The question mark shows that any single character can occupy that position in the pattern.

Description: The ftpdir() call gets a directory from a host in wide format.

Example: The following is an example of the ftpdir() call.

```
int rc;
rc=ftpdir("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

In this example, a directory is obtained using a wide format of `*.c` files, and the directory is placed in a local file, `conypc.dir`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpget()

```
#include <ftpapi.h>

int ftpget(host, userid, passwd, acct, local, remote, mode, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
char *mode;
int transfertype;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The local file name.
remote	The remote file name.
mode	Either *w* for write or *a* for append.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description: The ftpget() call gets a file from an FTP server.

Example: The following is an example of the ftpget() call.

```
int rc;
rc=ftpget("conypc","jason","ehgr1",NULL,"new.doc","abc.doc","w",T_ASCII);
```

In this example, the ASCII file, `abc.doc`, on the host, `conypc`, is copied into the local current working directory as the file, `new.doc`. If the file, `new.doc`, already exists in the local current working directory, the file, `new.doc`, is overwritten with the contents of the file, `abc.doc`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftplogoff()

```
#include <ftpapi.h>

void ftplogoff()
```

Description: The ftplogoff() call closes all current connections. This call must be called by an application before terminating.

ftpls()

```
#include <ftpapi.h>

int ftpls(host, userid, passwd, acct, local, pattern)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *pattern;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The name of the local file to which the information is placed.
pattern	Specifies the file name or pattern of the files to be deleted on the foreign host. Patterns are any combination of ASCII characters. The following two characters have special meaning: * The asterisk shows that any character or group of characters can occupy that position in the pattern. ? The question mark shows that any single character can occupy that position in the pattern.

Description: The ftpls() call retrieves directory information from a host in short format and writes it to a local file.

Example: The following is an example of the ftpls() call.

```
int rc;
rc=ftpls("conypc","jason","ehgr1",NULL,"conypc.dir","*.c");
```

In this example, a directory is obtained using the short format of `*.c` files and the directory information is placed in the local file, `conypc.dir`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpmkd()

```
#include <ftpapi.h>

int ftpmkd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
dir	The name of the directory to be created.

Description: The ftpmkd() call creates a new directory on a host.

Example: The following is an example of the ftpmkd() call.

```
int rc;
rc=ftpmkd("conypc","jason","ehgr1",NULL,"mydir");
```

In this example, the directory, `mydir`, is created on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpping()

```
#include <ftpapi.h>

int ftpping(host, len, addr)
char *host;
int len;
unsigned long *addr;
```

Parameter	Description
host	The host running the FTP server.
len	The length of the ping packets.
addr	The buffer in which to return the internet address of the host.

Description: The ftpping() attempts to resolve the host name through a name server. If the name server is not present, ftpping() searches the HOSTS file in the ETC directory for a matching host name. Unlike the ping() call, the ftpping() call could take several seconds because it must resolve the host name before it sends a ping. For this reason, use the ftpping() call only in the initial attempt to determine if the host is alive. The ftpping() call sets the addr parameter to the internet address of the host. After the initial attempt, use this address value to call ping.

Example: The following is an example of the ftpping() call:

```
int rc;
unsigned long addr;

rc = ftpping("conypc", 256, &addr);
```

Return Values: If the ftpping() return value is positive, the return value is the number of milliseconds it took for the echo to return. If the return value is negative, it contains an error code. The following list contains ftpping() call return codes and their corresponding descriptions.

Return Code	Description
PINGREPLY	Host does not reply.
PINGSOCKET	Unable to obtain socket.
PINGPROTO	Unknown protocol ICMP.
PINGSEND	Send failed.
PINGRECV	Recv failed.
PINGHOST	Unknown host.

ftpproxy()

```
#include <ftpapi.h>

int ftpproxy(host1, userid1, passwd1, acct1, host2, userid2, passwd2, acct2, fn1, fn2,
        transfertype)
char *host1;
char *userid1;
char *passwd1;
char *acct1;
char *host2;
char *userid2;
char *passwd2;
char *acct2;
char *fn1;
char *fn2;
int transfertype;
```

Parameter	Description
host1	The target host running the FTP server.
userid1	The user ID used for logon on host 1.
passwd1	The password of the user ID on host 1.
acct1	The account for host 1, when needed, can be NULL.
host2	The source host running the FTP server.
userid2	The user ID used for logon on host 2.
passwd2	The password of the user ID on host 2.
acct2	The account for host 2, when needed, can be NULL.
fn1	The file to be written on host 1.
fn2	The file to be copied from host 2.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description: The ftpproxy() call copies a file on a specified source host directly to a specified target host, without involving the requesting host in the file transfer. This call is functionally equivalent to the FTP client subcommand *proxy put.*

Note: Both the source and the target hosts must be running the FTP servers for the ftpproxy() to complete successfully.

Example: The following is an example of the ftpproxy() call.

```
int rc;
rc=ftpproxy("pc1","oleg","erst",NULL,  /* target host information*/
            "pc2","yan", "dssa1", NULL, /* source host information*/
            "\tmp\newdoc.1",            /* target file name */
            "\tmp\doc.1",               /* source file name */
            T_ASCII);                   /* ascii transfer */
```

In this example, the ASCII file, `\tmp\doc.1`, on the host `pc2`, is copied to host `pc1` as the file, `\tmp\newdoc.1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftppwd()

```
#include <ftpapi.h>

int ftppwd(host, userid, passwd, acct, buf, buflen)
char *host;
char *userid;
char *passwd;
char *acct,
char *buf;
int buflen;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
buf	The buffer to store the string returned by the FTP server.
buflen	The length of *buf*.

Description: The ftppwd() call stores the string containing the FTP server description of the current working directory on the host to the buffer *buf*. The string describing the current working directory is truncated to fit *buf* if it is longer than *buflen*. The returned string is always null-terminated.

Example: The following is an example of the ftppwd() call.

```
int rc;
rc=ftppwd("conypc","jason","ehgr1","dirbuf", sizeof dirbuf);
```

After the ftppwd() call the buffer `dirbuf` contains the following:

```
"C:\" is the current directory.
```

In this example, the server reply describing the current working directory on host `conypc`, using user ID `jason` with password `eghr1`, is stored to `dirbuf`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpput()

```
#include <ftpapi.h>

int ftpput(host, userid, passwd, acct, local, remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The local file name.
remote	The remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description: The ftpput() call transfers a file to an FTP server.

Example: The following is an example of the ftpput() call.

```
int rc;
rc=ftpput("conypc","jason","ehgr1",NULL,"abc.doc","new.doc",T_ASCII);
```

In this example, the ASCII file, `abc.doc`, on the local current working directory is copied to the current working directory of the host, `conypc`, as file `new.doc`. If the file, `new.doc`, already exists, the contents of the file, `new.doc`, is overwritten with the contents of the file, `abc.doc`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpputunique()

```
#include <ftpapi.h>

int ftpputunique(host, userid, passwd, acct, local, remote, transfertype)
char *host;
char *userid;
char *passwd;
char *acct;
char *local;
char *remote;
int transfertype;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
local	The local file name.
remote	The remote file name.
transfertype	Specifies a binary or ASCII transfer. T_ASCII is for ASCII, T_BINARY is for binary.

Description: The ftpputunique() call copies a local file to a file on a specified host and guarantees that the new file has a unique name and that the new file does not overwrite a file with the same name. If the file already exists on the host, a new and unique file name is created and used as the target of the file transfer.

Example: The following is an example of the ftpputunique() call.

```
int rc;
rc=ftpputunique(
       "conypc","jason","ehgr1",NULL,"abc.doc", "new.doc",T_ASCII);
```

In this example, the ASCII file, `abc.doc`, is copied to the current working directory of the host, `conypc`, as file `new.doc` unless the file, `new.doc`, already exists. If the file, `new.doc`, already exists, the file, `new.doc`, is given a new name unique within the current working directory on the host, `conypc`. The name of the new file is displayed on successful completion of the file transfer.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpquote()

```
#include <ftpapi.h>

int ftpquote(host, userid, passwd, acct, quotestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *quotestr;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
quotestr	The quote string to be passed to the FTP server verbatim.

Description: The ftpquote() call sends a string to the server verbatim.

Example: The following is an example of the ftpquote() call.

```
int rc;
rc=ftpquote("conypc","jason","ehgr1",NULL,"site idle 2000");
```

In this example, the idle is set to time out in 2000 seconds. Your server may not support that amount of idle time.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftprename()

```
#include <ftpapi.h>

int ftprename(host, userid, passwd, acct, namefrom, nameto)
char *host;
char *userid;
char *passwd;
char *acct;
char *namefrom;
char *nameto;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
namefrom	The original file name.
nameto	The new file name.

Description: The ftprename() call renames a file on a host.

Example: The following is an example of the ftprename() call.

```
int rc;
rc=ftprename("conypc","jason","ehgr1",NULL,"abc.1","cd.fg");
```

In this example, the file, `abc.1`, is renamed to `cd.fg` on a host, `conypc`, using user ID, `jason`, with password, `ehgr1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftprmd()

```
#include <ftpapi.h>

int ftprmd(host, userid, passwd, acct, dir)
char *host;
char *userid;
char *passwd;
char *acct;
char *dir;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
dir	The directory to be removed.

Description: The ftprmd() call removes a directory on a host.

Example: The following is an example of the ftprmd() call.

```
int rc;
rc=ftprmd("conypc","jason","ehgr1",NULL,"mydir");
```

In this example, the directory, `mydir`, is removed on the host, `conypc`, using the user ID, `jason`, and the password, `ehgr1`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpsite()

```
#include <ftpapi.h>

int ftpsite(host, userid, passwd, acct, sitestr)
char *host;
char *userid;
char *passwd;
char *acct;
char *sitestr;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
sitestr	The site string to be executed.

Description: The ftpsite() call executes the site command.

Example: The following is an example of the ftpsite() call.

```
int rc;
rc=ftpsite("conypc","jason","ehgr1",NULL,"idle 2000");
```

In this example, the idle time-out is set to 2000 seconds. Your server may not support that amount of idle time.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ftpsys()

```
#include <ftpapi.h>

int ftpsys(host, userid, passwd, acct, buf, buflen)
char *host;
char *userid;
char *passwd;
char *acct,
char *buf;
int buflen;
```

Parameter	Description
host	The host running the FTP server.
userid	The user ID used for logon.
passwd	The password of the user ID.
acct	The account, when needed, can be NULL.
buf	The buffer to store the string returned by the FTP server.
buflen	The length of *buf*.

Description: The ftpsys() call stores the string containing the FTP server description of the operating system running on the host to the buffer *buf*. The string describing the operating system of the host is truncated to fit *buf* if it is longer than *buflen*. The returned string is always null-terminated.

Example: The following is an example of the ftpsys() call.

```
int rc;
rc=ftpsys("ralvmm","jason","ehgr1",hostsysbuf, sizeof hostsysbuf);
```

After the ftpsys() call the buffer `hostsysbuf` contains the following:

```
VM is the operating system of this server.
```

In this example, the FTP server reply describing the operating system of host `ralvmm` using user ID `jason` with password `eghr1` is stored to `hostsysbuf`.

Return Values: The value 0 indicates success; the value −1 indicates an error. The value of *ftperrno* indicates the specific error.

ping()

```
#include <ftpapi.h>

ping(addr, len)
u_long addr;
int len;
```

Parameter	Description
addr	The internet address of the host in network byte order.
len	The length of the ping packets.

Description: The ping() call sends a ping to the host with ICMP Echo Request. The ping() call is useful to determine whether the host is alive before attempting FTP transfers, because time-out on regular connections is more than a minute. The ping() call returns within 3 seconds, at most, if the host is not responding. If the return value is positive, the return value is the number of milliseconds it took for the echo to return. If the return value is negative, it contains an error code. The parameter *len* specifies the length of the ping packet(s).

Example: The following is an example of the ping() call.

```
#include <stdio.h>
#include <netdb.h>
#include <ftpapi.h>

struct hostent *hp;     /* Pointer to host info */

main(argc, argv, envp)
   int argc;
   char *argv[];
   char *envp[];

{
   int i;
   unsigned long addr;

   if (argc!=2) {
       printf("Usage: p <host>\n");
       exit(1);
   }

   hp = gethostbyname(argv[1]);
   if (hp) {
           memcpy( (char *)&addr, hp->h_addr, hp->h_length);
           i = ping(addr,256);
           printf("ping reply in %d milliseconds\n",i);
   } else {
           printf("unknown host\n");
           exit(2);
   }
   ftplogoff(); /* close all connections */
}
```

Return Values: The following are ping() call return codes and their corresponding descriptions:

Return Code	Description
PINGREPLY	Host does not reply.
PINGSOCKET	Unable to obtain socket.
PINGPROTO	Unknown protocol ICMP.
PINGSEND	Send failed.
PINGRECV	Recv failed.

Chapter 6. Timer Routines

Chapter 6. Timer Routines

This chapter describes the timer routines in the Application Programming Interface (API) for TCP/IP for DOS. Use the timer routines to create, set, clear, and remove timers in your application programs.

Ensure that you have initialized the TCPIP library with the sock_init() or dosip_init() routines before you call any of the timer routines. See Chapter 3, "Sockets" for more information about the sock_init() and dosip_init() initialization routines.

Timers and the Timer Task

The sock_init() and dosip_init() calls start a task called the timer task. The timer task contains timers, which are counters that count down from an initial value to zero. Think of a timer as an alarm clock. Specifying the initial value sets the timer (alarm clock). The value of the counter is the time left on the timer. A timer that is counting down is ticking. When a timer reaches zero, it goes off. After a timer goes off, it is dormant unless you set it again. If a timer is ticking, stopping the timer and making it dormant is referred to as clearing the timer.

A List of Timer Routines

TCP/IP for DOS provides routines for creating, setting, clearing, and removing timers. The following is a list of timer routines and their functions.

Timer Routine	Function
tm_alloc()	Creates a timer.
tm_tset()	Sets a timer to go off in a specific number of ticks.
tm_set()	Sets a timer to go off in a specific number of seconds.
tm_mset()	Sets a timer to go off in a specific number of milliseconds.
tm_retset()	Changes the time left on a timer that is already ticking. You can specify the new time in ticks.
tm_reset()	Changes the time left on a timer that is already ticking. You can specify the new time in seconds.
tm_remset()	Changes the time left on a timer that is already ticking. You can specify the new time in milliseconds.
tm_clear()	Clears a timer.
tm_free()	Removes a timer from the list of timers. Because timers use system resources, always remove timers that you no longer need.

tm_alloc()

```
#include <timer.h>

timer *tm_alloc();
```

Description: The tm_alloc() routine creates a new timer in the timing task.

Return Values: If tm_alloc() is successful, it returns a pointer to the timer it created. If tm_alloc() is unsuccessful, it returns a NULL pointer.

tm_tset(), tm_set(), and tm_mset()

```
#include <timer.h>

void tm_tset(ticks, handler, arg, tm);
int ticks;
int (*handler)();
char *arg;
timer *tm;

void tm_set(secs, handler, arg, tm);
int secs;
int (*handler)();
char *arg;
timer *tm;

void tm_mset(msecs, handler, arg, tm);
int msecs;
int (*handler)();
char *arg;
timer *tm;
```

Description: The tm_tset(), tm_set() and tm_mset() routines set the *tm* timer to an initial value of either *ticks* ticks, *secs* seconds, or *msecs* milliseconds. When the *tm* timer goes off, the timer task calls the *handler* routine with the *arg* argument. You are responsible for writing the *handler* routine.

tm_retset(), tm_reset(), and tm_remset()

```
#include <timer.h>

int tm_retset(ticks, tm);
int ticks;
timer *tm;

int tm_reset(secs, tm);
int secs;
timer *tm;

int tm_remset(msecs, tm);
int msecs;
timer *tm;
```

Description: The tm_retset(), tm_reset() and tm_remset() routines change the time left on the *tm* timer to either *ticks* ticks, *secs* seconds, or *msecs* milliseconds.

Return Values: If tm_retset(), tm_reset(), or tm_remset() is successful, it returns the value of 1 (TRUE). If tm_retset(), tm_reset(), or tm_remset() is unsuccessful because the *tm* timer is dormant or does not exist, the routine returns the value of 0 (FALSE).

tm_clear()

```
#include <timer.h>

int tm_clear(tm);
timer *tm;
```

Description: The tm_clear() routine stops the *tm* timer from ticking and makes the timer dormant.

Return Values: If tm_clear() is successful, it returns the value of 1 (TRUE). If tm_clear() is unsuccessful because the *tm* timer is already dormant or does not exist, the routine returns the value of 0 (FALSE).

tm_free()

```
#include <timer.h>

int tm_free(tm);
timer *tm;
```

Description: The tm_free() routine removes the *tm* timer from the timer task. Ensure that the *tm* timer is dormant before you remove it.

Note: Because timers use system resources, always remove timers that you no longer need.

Return Values: If tm_free() is successful, it returns the value of 1 (TRUE). If tm_free() is unsuccessful because the *tm* timer is ticking (not dormant), the routine returns the value of 0 (FALSE).

Chapter 7. Tasking Routines

Chapter 7. Tasking Routines

This chapter describes the tasking routines in the Application Programming Interface (API) for TCP/IP for DOS. Using the tasking routines allows your DOS system to behave as though it were running a list of tasks simultaneously. For example, if you write a server application program that waits for requests, you can have this program running on your computer at the same time that you are using your computer to write a letter.

Ensure that you have initialized the TCPIP library with the sock_init() or dosip_init() routines before you call any of the tasking routines. See Chapter 3, "Sockets" for more information about the sock_init() and dosip_init() initialization routines.

Tasking and the Scheduler

In the UTIL there is a scheduler that maintains a circular list of tasks and schedules time for each task to run. The circular list of tasks is called the tasking ring. Each task is an independent program that runs once every time the scheduler loops through the tasking ring (round-robin tasking). Each task runs until it decides to stop and let the next task run (nonpreemptive, cooperative tasking). The task that is running at a particular time is called the current task. Each task takes its turn at being the current task.

All programs in TCP/IP for DOS, including all user application programs, are tasks that voluntarily stop running and let the next task run. The only exception is DOS itself. When DOS is running, the scheduler must preempt DOS to let other tasks run.

The scheduler preempts DOS once every tasking interval. The default value of the tasking interval is 15 ticks (18 ticks = 1 second). To change this value, use the quantum subcommand of the IFCONFIG command. For more information about the IFCONFIG command, see *IBM TCP/IP Version 2.0 for DOS: User's Guide*.

Tasks, Task State Vectors, and Task Status

A task is a program that runs once every cycle of the tasking ring.

In the ring, each task is represented by a task state vector. This vector contains the parameters needed to run the task, a unique task identification number, and two global variables defining the task status.

The identification number is called a Process IDentification number (PID). The scheduler stores the PID for the current task in the variable *tk_cur*.

The two global variables that define a task's status are the included and marked variables for removal flag, and the wake counter. The included and marked variables for removal flag indicates whether a task is included in the tasking ring, or the task is about to be removed from the tasking ring.

The wake counter indicates whether the task is going to run, or pass up its chance to run, the next time it is scheduled. If a task is going to run next time it is scheduled, the task is awake and the wake counter is greater than zero. If a task is going to pass up its chance to run next time it is scheduled, the task is asleep and the wake counter is less than or equal to zero.

The current task can change its own status, or change the status of another task. For example, the current task can keep itself awake, put itself to sleep, or remove itself from the tasking ring. The current task can also include a new task in the tasking ring, wake another task, put another task to sleep, or mark another task to be removed from the tasking ring.

The Wake Counter

The wake counter indicates whether a task is awake or asleep. Consider a task (the service task) that provides a service to other tasks (the requesting tasks). When the service task runs, it looks in a queue and provides the service to the first requesting task in the queue. Next time the service task runs, it provides the service to the next requesting task in the queue. This continues until the queue is empty. The service task does not run again until there is at least one requesting task in the queue.

The scheduler does not know anything about the specifics of the tasks it schedules, so how does the scheduler know when to run or not run the service task? The answer is in the wake counter. When a requesting task runs, and wants to request the service, that task puts its request in the queue, and increments the wake count of the service task. Several requesting tasks may run, and make requests, before the service task runs.

When the service task is scheduled, the scheduler decrements the wake count of the service task by one, and the service task provides the service to one of the requesting tasks. This continues until the wake count reaches zero, and the service task has provided the service to all the requesting tasks. The service task is always awake (wake count greater than zero) when one or more requesting tasks are waiting in the queue, and the service is always asleep (wake count less than or equal to zero) when there are no requesting tasks in the queue.

A List of Tasking Routines

The following is a list of the tasking routines and their functions.

Tasking Routine	Function
tk_fork()	Includes a new task in the tasking ring.
tk_contract()	Registers a task in case that task leaves memory (exits) prematurely.
tk_yield()	Lets the next task run while keeping the current task awake.
tk_wake()	Wakes another task.
tk_block()	Lets the next task run while putting the current task to sleep.
tk_sleep()	Puts another task to sleep.
tk_shell()	Temporarily runs DOS from inside the current task.
tk_exit()	Removes the current task from the tasking ring.
tk_kill()	Marks another task for removal from the tasking ring.

tk_fork()

```
#include <task.h>
#include <types.h>

task_t tk_fork(start, stack, stksize, name, arg);
int (far *start)();
unsigned char near *stack;
int stksize;
unsigned char *name;
unsigned long arg;
```

Description: The tk_fork() routine includes a new task in the tasking ring. The new task is the program *start* with the argument *arg*. The new task has a stack *stack* of size *stksize*. The name of the new task, as a character string, is *name*. The new task starts out awake, with a wake count of 1.

Each time you include a new task in the tasking ring with tk_fork(), register the task with tk_contract(). Registering the task ensures that if the task leaves memory (exits) prematurely, it is immediately removed from the tasking ring. For more information, see "tk_contract()" on page 224.

Return Values: If tk_fork() is successful, it returns a nonzero positive integer PID for the new task. If tk_fork() is unsuccessful because there is not enough memory for the new stack, the routine returns the value of 0.

tk_contract()

```
#include <task.h>
#include <types.h>

int tk_contract(pid);
task_t pid;
```

Description: The tk_contract() routine registers the task with PID *pid*, so that in the event that the task leaves memory (exits) prematurely, the task is immediately removed from the tasking ring. Each time you include a new task in the tasking ring with tk_fork(), register the task with tk_contract().

tk_yield()

```
#include <task.h>
#include <types.h>

int tk_yield();
```

Description: The tk_yield() routine lets the next task run, while keeping the current task awake. This routine increments the wake counter for the current task, to ensure the current task stays awake.

Return Values: If the current task is still included in the tasking ring, when the the task is scheduled again, tk_yield() returns with the value of 0. If the current task has been marked for removal by another task, when the the task is scheduled again, tk_yield() returns with a nonzero error code. Release all memory and other resources allocated to the current task, because the scheduler is about to remove the current task from the tasking ring.

tk_wake()

```
#include <task.h>
#include <types.h>

int tk_wake(pid);
task_t pid;
```

Description: The tk_wake() routine wakes the task with PID *pid*, by incrementing the wake counter for that task.

Return Values: If tk_wake() is successful, it returns the value of 0. If tk_wake() is unsuccessful because *pid* is not a valid PID, the routine returns a nonzero integer.

tk_block()

```
#include <task.h>
#include <types.h>

int tk_block();
```

Description: The tk_block() routine lets the next task in the tasking ring run while putting the current task to sleep. This routine does not increment the wake counter for the current task, so the current task remains asleep until another task wakes it with tk_wake().

Return Values: If the current task is still included in the tasking ring, when the the task wakes up again, tk_block() returns with the value of 0. If the current task has been marked for removal by another task, when the the task wakes up again, tk_block() returns with a nonzero error code. Release all memory and other resources allocated to the current task, because the scheduler is about to remove the current task from the tasking ring.

tk_sleep()

```
#include <task.h>
#include <types.h>

int tk_sleep(pid);
task_t pid;
```

Description: The tk_sleep() routine puts the task with PID *pid* to sleep, by decrementing the wake counter for that task.

Return Values: If tk_sleep() is successful, it returns the value of 0. If tk_sleep() is unsuccessful because *pid* is not a valid PID, the routine returns a nonzero integer.

tk_shell()

```
#include <task.h>
#include <types.h>

void tk_shell(sigstop);
unsigned sigstop;
```

Description: The tk_shell() routine temporarily runs DOS from inside the current task. When DOS is running, you can enter DOS commands at the DOS prompt. To return to the task, type EXIT at the DOS prompt and then press **Enter**. Use this routine when you want to be able to temporarily leave a task, use DOS commands, and then return where you left off in the task.

If you call tk_shell with the value of 1 (TRUE) for *sigstop*, then the scheduler does not preempt DOS while DOS is temporarily running. If you call tk_shell with the value of 0 (FALSE) for *sigstop*, then the scheduler preempts DOS once every tasking interval as usual.

Return Values: If tk_shell() is successful, it returns the value of 0. If tk_shell() is unsuccessful because it cannot run DOS, the routine returns the value of −1 and the global variable ERRNO is set to the error code.

tk_exit()

```
#include <task.h>
#include <types.h>

void tk_exit();
```

Description: The tk_exit() routine removes the current task from the tasking ring. Release all memory and other resources allocated to the current task before using tk_exit() to remove the task from the tasking ring.

tk_kill()

```
#include <task.h>
#include <types.h>

int tk_kill(pid);
task_t pid;
```

Description: The tk_kill() routine marks the task with PID *pid* for removal from the tasking ring. The tasker schedules this task once more so you can release the memory and other resources allocated to that task.

Return Values: If tk_kill() is successful, it returns the value of 0. If tk_kill() is unsuccessful because *pid* is not a valid PID, the routine returns a nonzero integer.

Appendixes

Appendix A. Well-Known Port Assignments

This appendix lists the well-known port assignments for the TCP and UDP transport protocols, the port number, keyword, and a description of the reserved port assignment. You can also find a list of these well-known port numbers in the SERVICES file.

TCP Well-Known Port Assignments

Table 1 lists the well-known port assignments for TCP.

Table 1 (Page 1 of 2). TCP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	**rje**	remote job entry	remote job entry
7	**echo**	echo	echo
9	**discard**	discard	sink null
11	**systat**	active users	active users
13	**daytime**	daytime	daytime
15	**netstat**	Netstat	who is up or Netstat
19	**chargen**	ttytst source	character generator
21	**ftp**	FTP	File Transfer Protocol
23	**telnet**	Telnet	Telnet
25	**smtp**	mail	Simple Mail Transfer Protocol
37	**time**	timeserver	timeserver
39	**rlp**	resource	Resource Location Protocol
42	**nameserver**	name	host name server
43	**nicname**	NICNAME/WHOIS	NICNAME/WHOIS
53	**domain**	name server	domain name server
57	**mtp**	private terminal access	private terminal access
69	**tftp**	TFTP	Trivial File Transfer Protocol
77	**rje**	netrjs	any private RJE service
79	**finger**	finger	finger
87	**link**	ttylink	any private terminal link
95	**supdup**	supdup	SUPDUP Protocol
101	**hostname**	hostname	nic hostname server, usually from SRI-NIC
109	**pop**	postoffice	Post Office Protocol Version 2.0
111	**sunrpc**	sunrpc	Sun remote procedure call
113	**auth**	authentication	authentication service
115	**sftp**	sftp	Simple File Transfer Protocol
117	**uucp-path**	UUCP path service	UUCP path service
119	**untp**	readnews untp	USENET News Transfer Protocol

Table 1 (Page 2 of 2). TCP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
123	**ntp**	NTP	Network Time Protocol
160 – 223		reserved	
712	**vexec**	vice-exec	Andrew File System authenticated service
713	**vlogin**	vice-login	Andrew File System authenticated service
714	**vshell**	vice-shell	Andrew File System authenticated service
2001	**filesrv**		Andrew File System service
2106	**venus.itc**		Andrew File System service, for the Venus process

UDP Well-Known Port Assignments

Table 2 lists the well-known port assignments for UDP.

Table 2. UDP Well-Known Port Assignments

Port Number	Keyword	Reserved for	Services Description
0		reserved	
5	**rje**	remote job entry	remote job entry
7	**echo**	echo	echo
9	**discard**	discard	sink null
11	**users**	active users	active users
13	**daytime**	daytime	daytime
15	**netstat**	Netstat	Netstat
19	**chargen**	ttytst source	character generator
37	**time**	timeserver	timeserver
39	**rlp**	resource	Resource Location Protocol
42	**nameserver**	name	host name server
43	**nicname**	NICNAME/WHOIS	NICNAME/WHOIS
53	**domain**	name server	domain name server
67	**bootps**	bootps	bootp server
68	**bootpc**	bootpc	bootp client
69	**tftp**	TFTP	Trivial File Transfer Protocol
75			any private dial out service
77	**rje**	netrjs	any private RJE service
79	**finger**	finger	finger
111	**sunrpc**	sunrpc	Sun remote procedure call
123	**ntp**	NTP	Network Time Protocol
135	**llbd**	NCS LLBD	NCS local location broker daemon
160 – 223		reserved	
531	**rvd-control**		rvd control port
2001	**rauth2**		Andrew File System service, for the Venus process
2002	**rfilebulk**		Andrew File System service, for the Venus process
2003	**rfilesrv**		Andrew File System service, for the Venus process
2018	**console**		Andrew File System service
2115	**ropcons**		Andrew File System service, for the Venus process
2131	**rupdsrv**		assigned in pairs; bulk must be **srv** +1
2132	**rupdbulk**		assigned in pairs; bulk must be **srv** +1
2133	**rupdsrv1**		assigned in pairs; bulk must be **srv** +1
2134	**rupdbulk1**		assigned in pairs; bulk must be **srv** +1

Appendix B. Sample Socket Programs

This appendix provides examples of the following C language programs:

- Socket UDP client
- Socket UDP server
- Socket TCP client
- Socket TCP server.

Socket UDP Client

The following is an example of a socket UDP client program:

```
#include <stdlib.h>
#include <types.h>
#include <netinet/in.h>
#include <sys/socket.h>

main(argc, argv)
int argc;
char **argv;
{
   int s;
   unsigned short port;
   struct sockaddr_in server;
   char buf[32];
   /*
    * argv[1] is internet address of server argv[2] is port of server.
    * Convert the port from ascii to integer and then from host byte
    * order to network byte order.
    */
   if(argc != 3)
   {
      printf("Usage: %s <host address> <port> \n",argv[0]);
      exit(1);
   }
   port = htons(atoi(argv[2]));

   /* Initialize with sockets */
   sock_init();
   /*
    * Create a datagram socket in the internet domain and use the
    * default protocol (UDP).
    */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
       perror("socket()");
       exit(1);
   }

   /* Set up the server name */
   server.sin_family = AF_INET;            /* Internet Domain    */
   server.sin_port = port;                 /* Server Port        */
   server.sin_addr.s_addr = inet_addr(argv[1]);
                                                /* Server's Address   */

   strcpy(buf, "Hello");
```

```
    /* Send the message in buf to the server */
    if (sendto(s, buf, (strlen(buf)+1), 0, &server, sizeof(server)) < 0)
    {
        perror("sendto()");
        exit(2);
    }

    /* Deallocate the socket */
    so_close(s);
}
```

Socket UDP Server

The following is an example of a socket UDP server C language program:

```
#include <stdlib.h>
#include <types.h>
#include <netinet/in.h>
#include <sys/socket.h>

main()
{
   int s, namelen, client_address_size;
   struct sockaddr_in client, server;
   char buf[32];

   /*
    * Initialize with sockets.
    */
   sock_init();

   /*
    * Create a datagram socket in the internet domain and use the
    * default protocol (UDP).
    */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) < 0)
   {
       perror("socket()");
       exit(1);
   }

   /*
    * Bind my name to this socket so that clients on the network can
    * send me messages. (This allows the operating system to demultiplex
    * messages and get them to the correct server)
    *
    * Set up the server name. The internet address is specified as the
    * wildcard INADDR_ANY so that the server can get messages from any
    * of the physical internet connections on this host. (Otherwise we
    * would limit the server to messages from only one network interface)
    */
   server.sin_family = AF_INET;      /* Server is in Internet Domain */
   server.sin_port = 0;                 /* Use any available port      */
   server.sin_addr.s_addr = INADDR_ANY;/* Server's Internet Address    */

   if (bind(s, &server, sizeof(server)) < 0)
   {
       perror("bind()");
       exit(2);
   }

   /* Find out what port was really assigned and print it */
   namelen = sizeof(server);
   if (getsockname(s, (struct sockaddr *) &server, &namelen) < 0)
   {
       perror("getsockname()");
       exit(3);
   }

   printf("Port assigned is %d\n", ntohs(server.sin_port));
```

```
    /*
     * Receive a message on socket s in buf  of maximum size 32
     * from a client. Because the last two parameters
     * are not null, the name of the client will be placed into the
     * client data structure and the size of the client address will
     * be placed into client_address_size.
     */
    client_address_size = sizeof(client);

    if(recvfrom(s, buf, sizeof(buf), 0, (struct sockaddr *) &client,
            &client_address_size) <0)
    {
        perror("recvfrom()");
        exit(4);
    }
    /*
     * Print the message and the name of the client.
     * The domain should be the internet domain (AF_INET).
     * The port is received in network byte order, so we translate it to
     * host byte order before printing it.
     * The internet address is received as 32 bits in network byte order
     * so we use a utility that converts it to a string printed in
     * dotted decimal format for readability.
     */
    printf("Received message %s from domain %s port %d internet address %s\n",
        buf,
        (client.sin_family == AF_INET?"AF_INET":"UNKNOWN"),
        ntohs(client.sin_port),
        inet_ntoa(client.sin_addr));

    /*
     * Deallocate the socket.
     */
    so_close(s);
}
```

Socket TCP Client

The following is an example of a socket TCP client C language program:

```
/*
 * Include Files.
 */
#include <types.h>
#include <netinet\in.h>
#include <sys\socket.h>
#include <netdb.h>
#include <stdio.h>

/*
 * Client Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port client will connect to              */
    char buf[12];    /* data buffer for sending and receiving    */
    struct hostent *hostnm;    /* server host name information             */
    struct sockaddr_in server; /* server address                           */
    int s;                     /* client socket                            */

    /*
     * Check Arguments Passed. Should be hostname and port.
     */
    if (argc != 3)
    {
        fprintf(stderr, "Usage: %s hostname port\n", argv[0]);
        exit(1);
    }

    /*
     * Initialize with sockets.
     */
    sock_init();

    /*
     * The host name is the first argument. Get the server address.
     */
    hostnm = gethostbyname(argv[1]);
    if (hostnm == (struct hostent *) 0)
    {
        fprintf(stderr, "Gethostbyname failed\n");
        exit(2);
    }

    /*
     * The port is the second argument.
     */
    port = (unsigned short) atoi(argv[2]);

    /*
     * Put a message into the buffer.
     */
    strcpy(buf, "the message");
```

```
    /*
     * Put the server information into the server structure.
     * The port must be put into network byte order.
     */
    server.sin_family      = AF_INET;
    server.sin_port        = htons(port);
    server.sin_addr.s_addr = *((unsigned long *)hostnm->h_addr);

    /*
     * Get a stream socket.
     */
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket()");
        exit(3);
    }

    /*
     * Connect to the server.
     */
    if (connect(s, &server, sizeof(server)) < 0)
    {
        perror("Connect()");
        exit(4);
    }

    if (send(s, buf, sizeof(buf), 0) < 0)
    {
        perror("Send()");
        exit(5);
    }

    /*
     * The server sends back the same message. Receive it into the buffer.
     */
    if (recv(s, buf, sizeof(buf), 0) < 0)
    {
        perror("Recv()");
        exit(6);
    }

    /*
     * Close the socket.
     */
    so_close(s);

    printf("Client Ended Successfully\n");
    exit(0);

}
```

Socket TCP Server

The following is an example of a socket TCP server C language program:

```
/*
 * Include Files.
 */
#include <types.h>
#include <netinet\in.h>
#include <sys\socket.h>
#include <stdio.h>

/*
 * Server Main.
 */
main(argc, argv)
int argc;
char **argv;
{
    unsigned short port;       /* port server binds to                */
    char buf[12];              /* buffer for sending and receiving data */
    struct sockaddr_in client; /* client address information          */
    struct sockaddr_in server; /* server address information          */
    int s;                     /* socket for accepting connections    */
    int ns;                    /* socket connected to client          */
    int namelen;               /* length of client name               */

    /*
     * Check arguments. Should be only one: the port number to bind to.
     */

    if (argc != 2)
    {
        fprintf(stderr, "Usage: %s port\n", argv[0]);
        exit(1);
    }

    /*
     * Initialize with sockets.
     */
    sock_init();

    /*
     * First argument should be the port.
     */
    port = (unsigned short) atoi(argv[1]);

    /*
     * Get a socket for accepting connections.
     */
    if ((s = socket(AF_INET, SOCK_STREAM, 0)) < 0)
    {
        perror("Socket()");
        exit(2);
    }
```

```
    /*
     * Bind the socket to the server address.
     */
    server.sin_family = AF_INET;
    server.sin_port   = htons(port);
    server.sin_addr.s_addr = INADDR_ANY;

    if (bind(s, &server, sizeof(server)) < 0)
    {
        perror("Bind()");
        exit(3);
    }

    /*
     * Listen for connections. Specify the backlog as 1.
     */
    if (listen(s, 1) != 0)
    {
        perror("Listen()");
        exit(4);
    }

    /*
     * Accept a connection.
     */
    namelen = sizeof(client);
    if ((ns = accept(s, &client, &namelen)) == -1)
    {
        perror("Accept()");
        exit(5);
    }

    /*
     * Receive the message on the newly connected socket.
     */
    if (recv(ns, buf, sizeof(buf), 0) == -1)
    {
        perror("Recv()");
        exit(6);
    }

    /*
     * Send the message back to the client.
     */
    if (send(ns, buf, sizeof(buf), 0) < 0)
    {
        perror("Send()");
        exit(7);
    }

    so_close(ns);
    so_close(s);

    printf("Server ended successfully\n");
    exit(0);
}
```

Appendix C. Sample RPC Programs

This appendix provides examples of the following programs:

- RPC client
- RPC server.

RPC Client

The following is an example of an RPC client program:

```
/* GENERAL RPC CLIENT  */
/* Send an integer to the remote host and receive the integer back  */
/* PORTMAPPER AND REMOTE SERVER MUST BE RUNNING */

#include <stdio.h>
#include <rpc\rpc.h>
#include <sys\socket.h>
#define intrcvprog ((u_long)150000)
#define version    ((u_long)1)
#define intrcvproc ((u_long)1)

main(argc, argv)
   int argc;
   char *argv[];
{
   int innumber;
   int outnumber;
   int error;

   if (argc != 3)
   {
      fprintf(stderr,"usage: %s hostname integer\n", argv[0]);
      exit (-1);
   }  /* endif */

   innumber = atoi(argv[2]);
   /*
    * Send the integer to the server. The server should
    * return the same integer.
    */
   error = callrpc(argv[1],intrcvprog,version,intrcvproc,
           xdr_int, (char *)&innumber,xdr_int,(char *)&outnumber);

   if (error != 0)
   {
      fprintf(stderr,"error: callrpc failed: %d \n",error);
      fprintf(stderr,"intrcprog: %d version: %d intrcvproc: %d",
                     intrcvprog, version,intrcvproc);
      exit(1);
   }  /* endif */

   printf("value sent: %d   value received: %d\n", innumber, outnumber);
   exit(0);
}
```

RPC Server

The following is an example of an RPC server program:

```
/* GENERIC RPC SERVER */
/* RECEIVE AN INTEGER OR FLOAT AND RETURN THEM RESPECTIVELY */
/* PORTMAPPER MUST BE RUNNING */

#include <rpc\rpc.h>
#include <stdio.h>

#define intrcvprog ((u_long)150000)
#define fltrcvprog ((u_long)150102)
#define intvers    ((u_long)1)
#define intrcvproc ((u_long)1)
#define fltrcvproc ((u_long)1)
#define fltvers    ((u_long)1)

main()
{
   int *intrcv();
   float *floatrcv();

  /*REGISTER PROG, VERS AND PROC WITH THE PORTMAPPER*/

      /*FIRST PROGRAM*/
   registerrpc(intrcvprog,intvers,intrcvproc,intrcv,xdr_int,xdr_int);
   printf("Intrcv Registration with Port Mapper completed\n");

      /*OR MULTIPLE PROGRAMS*/
   registerrpc(fltrcvprog,fltvers,fltrcvproc,floatrcv,xdr_float,xdr_float);
   printf("Floatrcv Registration with Port Mapper completed\n");

   /*
    * svc_run will handle all requests for programs registered.
    */
   svc_run();
   printf("Error:svc_run returned!\n");
   exit(1);
}

/*
 * Procedure called by the server to receive and return an integer.
 */
int *
intrcv(in)
   int *in;
{
   int *out;

   printf("integer received: %d\n",*in);
   out = in;
   printf("integer being returned: %d\n",*out);
   return (out);
}
```

```
/*
 * Procedure called by the server to receive and return a float.
 */

float *
floatrcv(in)
   float *in;
{
   float *out;

   printf("float received: %e\n",*in);
   out=in;
   printf("float being returned: %e\n",*out);
   return(out);
}
```

Appendix D. Sample Tasking Program

This appendix provides examples of the following tasking functions:

- tk_fork()
- tk_yield().

Tasking Program

```
/* Illustrates multiple threads using functions:
 *      tk_fork()               tk_yield()
 *
 * Also the global variable:
 *      tk_cur
 *
 * This program requires the library TCPIP.LIB
 * cl -c -AL -J -Gs -Oars -FPc -Zp2 threads.c
 * link /noi /stack:16000 threads,,,%TCPBASE%\lib\l\tcpip.lib;
 */
#include <types.h>
#include <task.h>
#include <sys/socket.h>
#include <stdio.h>
#include <sys/timeb.h>
#include <stdlib.h>
#include <conio.h>
void far Bounce( int c );
void far CheckKey( void *dummy );

/* GetRandom returns a random integer between min and max. */
#define GetRandom( min, max ) ((rand() % (int)(((max)+1) - (min))) + (min))
#define STACK_SIZE   1024
#define BOOL    int
#define PCHAR   char *
#define CHAR    char

BOOL repeat = TRUE;         /* Global repeat flag and video variable */

struct {
   int col;
   int row;
   } vmi;

void VioWrtCellStr(char * cell,int len,int y,int x,int dummy )
{
   memcpy((void far *)(0xB8000000 + (((y * 80) + x)*2)),(char far *) cell, len);
}

void DosSleep(unsigned long duration)
{
   struct timeb time1;
   struct timeb time2;

   ftime(&time1);
   do
   {
      tk_yield();
```

```
      ftime(&time2);
   }
   while ((((time2.time-time1.time)*1000)+(time2.millitm-time1.millitm) <
duration) ); /* enddo */

}

void main()
{
    PCHAR   stack;
    CHAR    ch = 'A';

    sock_init();

    /* Get display screen's text row and column information. */
    vmi.col = 80;
    vmi.row = 25;

    /* Launch CheckKey thread to check for terminating keystroke. */
    tk_fork(CheckKey, (char near *)NULL, STACK_SIZE, "CheckKey", (u_long)0);
    tk_contract( tk_cur );

    /* Loop until CheckKey terminates program. */
    while( repeat )
    {
        /* On first loops, launch character threads. */
        if ( tk_fork(Bounce,
                     (char near *)NULL,
                     STACK_SIZE,
                     "Bounce",
                     (u_long)ch++) ) {

            tk_contract( tk_cur );
        } /* endif */

        /* Wait one second between loops. */
        DosSleep( 1000L );
    }
}

/* CheckKey - Thread to wait for a keystroke, then clear repeat flag. */
void CheckKey( void *dummy )
{
    while (!kbhit()) {
       tk_yield();
    } /* endwhile */
    tk_stats();
    getch();
    repeat = 0;      /* endthread implied */
    tk_exit();
}
```

```
/* Bounce - Thread to create and control a colored letter that moves
 * around on the screen.
 *
 * Params: ch - the letter to be moved
 */
void Bounce( int ch )
{
    /* Generate letter and color attribute from thread argument. */
    char      blankcell[2];
    char      blockcell[2];
    int       xold, xcur, yold, ycur;
    BOOL      first = TRUE;

    blankcell[0] = 0x20;
    blankcell[1] = 0x07;
    blockcell[0] = ch;
    blockcell[1] = (ch % 16) + 1;

    /* Seed random-number generator and get initial location. */
    srand( tk_cur );
    xcur = GetRandom( 0, vmi.col - 1 );
    ycur = GetRandom( 0, vmi.row - 1 );
    while( repeat )
    {
        /* Pause between loops. */
        DosSleep( 100L );

        /* Blank out our old position on the screen, and draw new letter. */
        if( first )
            first = FALSE;
        else
            VioWrtCellStr( blankcell, 2, yold, xold, 0 );
        VioWrtCellStr( blockcell, 2, ycur, xcur, 0 );

        /* Increment the coordinate for next placement of the block. */
        xold = xcur;
        yold = ycur;
        xcur += GetRandom( -1, 1 );
        ycur += GetRandom( -1, 1 );

        /* Correct placement (and beep) if about to go off the screen. */
        if( xcur < 0 )
            xcur = 1;
        else if( xcur == vmi.col )
            xcur = vmi.col - 2;
        else if( ycur < 0 )
            ycur = 1;
        else if( ycur == vmi.row )
            ycur = vmi.row - 2;

        /* If not at screen border, continue; otherwise beep. */
        else
            continue;
 /*     printf("%c",0x07); */  /* BEEP */
    }
    tk_exit();
}
```

Appendix E. Socket Quick Reference

Table 3 describes each socket call supported by TCP/IP for DOS, and identifies the page in the book where you can find more information.

Table 3 (Page 1 of 2). Socket Quick Reference

SocketCall	Description	Page
accept()	Accepts a connection request from a foreign host.	36
bind()	Assigns a local address to the socket.	38
connect()	Requests a connection to a foreign server.	41
dosip_init()	Initializes the socket data structures and checks whether INET.EXE is running.	44
endhostent()	Closes the HOSTS file.	45
endnetent()	Closes the NETWORKS file.	46
endprotoent()	Closes the PROTOCOL file.	47
endservent()	Closes the SERVICES file.	48
gethostbyaddr()	Returns information about a host specified by an address.	49
gethostbyname()	Returns information about a host specified by a name.	50
gethostent()	Returns the next entry in the HOSTS file.	51
gethostid()	Returns the unique identifier of the current host.	52
getnetbyaddr()	Returns the network entry specified by address.	53
getnetbyname()	Returns the network entry specified by name.	54
getnetent()	Returns the next entry in the NETWORKS file.	55
getpeername()	Returns the name of the peer connected to socket *s*.	56
getprotobyname()	Returns a protocol entry specified by name.	57
getprotobynumber()	Returns a protocol entry specified by number.	58
getprotoent()	Returns the next entry in the PROTOCOL file.	59
getservbyname()	Returns a service entry specified by name.	60
getservbyport()	Returns a service entry specified by port number.	61
getservent()	Returns the next entry in the SERVICES file.	62
getsockname()	Obtains local socket name.	63
getsockopt()	Returns values of options associated with a socket.	64
htonl()	Translates byte order from host to network for a long integer.	67
htons()	Translates byte order from host to network for a short integer.	68
inet_addr()	Constructs an internet address from character strings set in standard dotted-decimal notation.	69
inet_lnaof()	Returns the local network portion of an internet address.	70
inet_makeaddr()	Constructs an internet address from a network number and a local address.	71
inet_netof()	Returns the network portion of the internet address in network byte order.	72

Table 3 (Page 2 of 2). Socket Quick Reference

SocketCall	Description	Page
inet_network()	Constructs a network number from character strings set in standard dotted-decimal notation.	73
inet_ntoa()	Returns a pointer to a string in dotted-decimal notation.	74
listen()	Indicates that a stream socket is ready for a connection request from a foreign client.	75
ntohl()	Translates byte order from network to host for a long integer.	76
ntohs()	Translates byte order from network to host for a short integer.	77
recv()	Receives messages on a connected socket.	78
recvfrom()	Receives messages on a datagram socket, regardless of its connection status.	79
select()	Returns read, write, and exception status on a group of sockets.	80
send()	Sends packets on a connected socket.	82
sendto()	Sends packets on a datagram socket, regardless of its connection status.	83
sethostent()	Opens and rewinds the HOSTS file.	84
setnetent()	Opens and rewinds the NETWORKS file.	85
setprotoent()	Opens and rewinds the PROTOCOL file.	86
setservent()	Opens and rewinds the SERVICES file.	87
setsockopt()	Sets options associated with a socket.	88
shutdown()	Shuts down all or part of a full-duplex connection.	90
sock_init()	Initializes the socket data structures and checks whether or not INET.EXE is running.	91
socket()	Requests that a socket be created.	92
so_close()	Closes the socket associated with the descriptor *s*.	95
so_flush()	Clears the contents of the socket.	96
so_read()	Receives messages on a connected socket.	97
so_write()	Sends packets on a connected socket.	98

Appendix F. Remote Procedure Call Quick Reference

Table 4 describes each RPC supported by TCP/IP for DOS, and identifies the page in the book where you can find more information.

Table 4 (Page 1 of 3). Remote Procedure Call Quick Reference

Remote Procedure Call	Description	Page
auth_destroy()	Destroys authentication information.	109
authnone_create()	Creates and returns a NULL RPC authentication handle.	110
authunix_create()	Creates and returns a UNIX-based authentication handle.	111
authunix_create_default()	Calls authunix_create() with default parameters.	112
callrpc()	Calls remote procedures.	113
clnt_call()	Calls the remote procedure associated with the client handle.	115
clnt_broadcast()	Calls remote procedures and locally broadcasts messages.	114
clnt_destroy()	Destroys client's RPC handle.	116
clnt_freeres()	De-allocates resources assigned for decoding RPC.	117
clnt_geterr()	Copies the error structure from a client's handle to the local address.	118
clnt_pcreateerror()	Indicates why a client handle cannot be created.	119
clnt_perrno()	Writes error message indicating why RPC failed.	120
clnt_perror()	Writes error message indicating why RPC failed.	121
clnttcp_create()	Creates an RPC client for the remote program using TCP transport.	122
clntudp_create()	Creates an RPC client for the remote program using UDP transport.	123
get_myaddress()	Returns the local host's internet address.	124
pmap_getmaps()	Returns a list of current program to port mappings on a specified foreign host.	125
pmap_getport()	Returns a port number associated with a remote program.	126
pmap_rmtcall()	Instructs a foreign host to make an RPC call on the client's behalf.	127
pmap_set()	Sets the mapping of a server program to a port on local machine.	128
pmap_unset()	Resets the mappings on the local machine.	129
registerrpc()	Registers the procedure with the local RPC Portmapper.	130
rpc_createerr	Global variable set when any RPC client creation routine fails.	131
svc_destroy()	Destroys the RPC service transport handle.	132
svc_fds()	A global variable reflecting the RPC service-side read file descriptor bit mask.	133
svc_freeargs()	Frees storage allocated for arguments.	134
svc_getargs()	Decodes arguments from an RPC service transport handle.	135
svc_getcaller()	Obtains the network address of the client associated with the service transport handle.	136
svc_getreq()	Implements asynchronous event processing, and returns control to the program after all sockets have been serviced.	137

Table 4 (Page 2 of 3). Remote Procedure Call Quick Reference

Remote Procedure Call	Description	Page
svc_register()	Registers procedures on the Portmapper.	138
svc_run()	Accepts RPC requests, and calls the appropriate service.	139
svc_sendreply()	Sends the results of an RPC to caller.	140
svc_unregister()	Removes the local mapping.	141
svcerr_auth()	Returns an error reply when the service cannot execute RPC because of authentication errors.	142
svcerr_decode()	Returns an error reply when the service cannot decode its parameters.	143
svcerr_noproc()	Returns an error reply when the service cannot call procedure requested.	144
svcerr_noprog()	Returns an error reply when the service cannot call the program requested.	145
svcerr_progvers()	Returns an error reply when the service cannot call a version of the program requested.	146
svcerr_systemerr()	Returns an error reply when the service detects a system error that has not been handled.	147
svcerr_weakauth()	Returns an error reply when the service cannot execute an RPC because of weak authentication parameters.	148
svctcp_create()	Creates TCP-based service transport.	149
svcudp_create()	Creates UDP-based service transport.	150
xdr_accepted_reply()	Translates RPC reply messages.	151
xdr_array()	Translates an array to its external representation.	152
xdr_authunix_parms()	Translates UNIX-based authentication information.	153
xdr_bool()	Translates booleans to their external representations.	154
xdr_bytes()	Translates counted byte strings.	155
xdr_callhdr()	Translates an RPC call message header.	156
xdr_callmsg()	Translates RPC call messages.	157
xdr_double()	Translates C double-precision numbers to their external representations.	158
xdr_enum()	Translates C-enumerated numbers to their external representations.	159
xdr_float()	Translates C floating-point numbers to their external representations.	160
xdr_inline()	Invokes the inline routine, and returns a pointer to the continuous piece of XDR buffer.	161
xdr_int()	Translates C integers to their external representations.	162
xdr_long()	Translates C long integers to their external representations.	163
xdr_opaque()	Translates fixed-size opaque data to its external representation.	164
xdr_opaque_auth()	Translates RPC authentication data.	165
xdr_pmap()	Translates port map elements.	166
xdr_pmaplist()	Translates a list of port mappings.	167
xdr_reference()	Provides pointer chasing within structures.	168

Table 4 (Page 3 of 3). Remote Procedure Call Quick Reference

Remote Procedure Call	Description	Page
xdr_rejected_reply()	Translates rejected RPC reply messages.	169
xdr_replymsg()	Translates RPC reply messages.	170
xdr_short()	Translates between C short integers and their external representations.	171
xdr_string()	Translates between C strings and their external representations.	172
xdr_u_int()	Translates between C unsigned integers and their external representations.	173
xdr_u_long()	Translates between C unsigned long integers and their external representations.	174
xdr_u_short()	Translates between C unsigned short integers and their external representations.	175
xdr_union()	Translates between a discriminated C union and its external representations.	176
xdr_void()	Returns a value of 1.	177
xdr_wrapstring()	Translates strings to their external representation.	178
xdrmem_create()	Initializes the stream object that is pointed to by the XDRs; writes to or reads from memory.	179
xdrrec_create()	Initializes the stream object that is pointed to by the XDRs; writes to or reads from a buffer.	180
xdrrec_endofrecord()	Marks the data in the output buffer as a completed record.	181
xdrrec_eof()	Marks the end of the file, after using the rest of the current record in the XDR stream.	182
xdrrec_skiprecord()	Discards the rest of the XDR stream's current record in the input buffer.	183
xdrstdio_create()	Initializes the stream object that is pointed to by the XDRs; writes to or reads from the standard input-output stream file.	184
xprt_register()	Registers service transport handles with the RPC service package.	185
xprt_unregister()	Unregisters the RPC service transport handle before it is destroyed.	186

Appendix G. FTP API Quick Reference

Table 5 describes each FTP API routine supported by TCP/IP for DOS, and identifies the page in the book where you can find more information.

Table 5. FTP API Quick Reference

FTP API Call	Description	Page
ftpappend()	Appends information to a remote file.	191
ftpcd()	Changes the current working directory.	192
ftpdelete()	Deletes files on a remote machine.	193
ftpdir()	Obtains a directory from a remote machine in wide format.	194
ftpget()	Obtains a file from a remote server.	195
ftplogoff()	Closes all current connections.	196
ftpls()	Obtains a directory from a remote machine in short format.	197
ftpmkd()	Creates a new directory on a target machine.	198
ftpping()	Attempts to resolve the host name through a name server.	199
ftpproxy()	Transfers a file between two remote servers without sending the file to a local machine.	200
ftpput()	Transfers a file to a remote FTP server.	202
ftpputunique()	Transfers a file to a remote host and names it uniquely on a remote machine.	203
ftppwd()	Stores the string containing the FTP server description of the current working directory on the host to the buffer.	201
ftpquote()	Sends a string to the server verbatim.	204
ftprename()	Renames a file on a remote machine.	205
ftprmd()	Removes a directory on a target machine.	206
ftpsite()	Executes the site command.	207
ftpsys()	Stores the string containing the FTP server description of the operating system running on the host to the buffer.	208
ping()	Sends a ping to the remote host to determine if that host is alive.	209

Appendix H. Timer Quick Reference

Table 6 describes each TIMER ROUTINE supported by TCP/IP for DOS, and identifies the page in the book where more information is located.

Table 6. Timer Quick Reference

Timer Routines	Description	Page
tm_alloc()	Creates a timer.	214
tm_clear()	Clears a timer.	217
tm_free()	Removes a timer from the list of timers. Because timers use system resources, always remove timers that are no longer needed.	218
tm_mset()	Sets a timer to go off in a specific number of milliseconds.	215
tm_remset()	Changes the time left on a timer that is already ticking. The new time is specified in milliseconds.	216
tm_reset()	Changes the time left on a timer that is already ticking. The new time is specified in seconds.	216
tm_retset()	Changes the time left on a timer that is already ticking. The new time is specified in ticks.	216
tm_set()	Sets a timer to go off in a specific number of seconds.	215
tm_tset()	Sets a timer to go off in a specific number of ticks.	215

Appendix I. Tasking Quick Reference

Table 7 describes each TASKING ROUTINE supported by TCP/IP for DOS, and identifies the page in the book where more information is located.

Table 7. Tasking Quick Reference

Tasking Routines	Description	Page
tk_block()	Lets the next task run while putting the current task to sleep.	227
tk_contract()	Registers a task in case that task leaves memory (exits) prematurely.	224
tk_exit()	Removes the current task from the tasking ring.	230
tk_fork()	Includes a new task in the tasking ring.	223
tk_kill()	Marks another task for removal from the tasking ring.	231
tk_shell()	Temporarily runs DOS from inside the current task.	229
tk_sleep()	Puts another task to sleep.	228
tk_wake()	Wakes up another task.	226
tk_yield()	Lets the next task run while keeping the current task awake.	225

Appendix J. NETWORKS File Structure

The NETWORKS file contains the network name, number, and alias(es) of known networks. The NETWORKS file must reside in your <*TCPBASE*>\ETC directory, or in the directory specified by the ETC environment variable. The NETWORKS file is used only by the following socket calls:

- endnetent()
- getnetbyaddr()
- getnetbyname()
- getnetent()
- setnetent().

Table 8 provides examples of network names contained in the NETWORKS file.

Table 8. Name Structures of Known Networks

Name of File	Contents of File	Sample File Entries
NETWORKS	*official_network_name network_number alias(es)*	ne-region 128.1 classb.net1 at1-region 128.2 classb.net2 lab-net 192.5.1 classc.net5

Appendix K. Messages and Codes

TCP/IP for DOS displays the following types of messages:

- Informational messages
- Warning messages
- Error messages
- Internal error messages.

Informational messages provide general information. Warning messages signify an unusual occurrence that does not stop the module. Error messages occur when the module cannot continue. Internal error messages signify an unexpected error.

Each type of message shows the name of the module that generated the message. In warning, error, and internal error messages, the type of message follows the module name. Messages are grouped as follows:

- General Modules
  - Errors
  - Internal Errors
  - Warnings
- Generic Text Messages
- IFCONFIG Errors
- Name Server Messages
- NFS Errors
- TSR Errors.

Within each of these groups, messages appear in alphabetical order according to the message text. For example, the following message:

```
error: You must specify a host and command
```

appears before:

```
error: You must specify a name to query
```

Note: Some modules, for example FTP and TELNET, are interactive. These shell and menu-based modules provide some messages that are self-explanatory and are not described in this appendix.

General Module Errors

The following error messages can appear in different modules.

---

module: `error: A command must be supplied`

Module: REXEC RSH

Explanation: An attempt to execute the specified command failed, because no remote command was supplied.

User Response: Reissue the command and include a remote command to be executed.

---

module: `error: A local user name must be supplied`

Module: RSH

Explanation: An attempt to execute the specified command failed, because no local user name was supplied on the command line and none was provided interactively.

User Response: Reissue the command and either include the local user name on the command line or provide it when prompted.

---

module: `error: A login name must be supplied`

Module: REXEC RSH

Explanation: An attempt to execute the specified command failed, because no logon name was supplied. The logon name is your logon ID on the remote host.

User Response: Reissue the command and include the logon name.

---

module: `error: Closing service` *'service'*: *message*

Module: NETWATCH

Explanation: An attempt to close *service* failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: `error: Component file must be specified after '-c' option`

Module: COMPOSE

Explanation: The -c option requires a valid file name following it.

User Response: Reissue the COMPOSE command with a valid file name following the -c option.

---

module: `error: Conflicting options: '-a' and '-b'`

Module: TFTP

Explanation: An attempt to execute the specified command failed, because both the BINARY and ASCII modes of transfer were specified. These options are mutually exclusive.

User Response: Reissue the command, specifying only one of these options.

---

module: `error: Conflicting options: '-b' and '-f'`

Module: LPR

Explanation: An attempt to execute the specified command failed, because both local binary file printing and passing the file through the PR filter on a UNIX server. These options are mutually exclusive.

User Response: Reissue the command, specifying only one of these options.

---

module: `error: Conflicting options: '-h' and '-c'`

Module: FTP

Explanation: An attempt to execute the specified command failed, because both byte count and hash mark options were specified. These options are mutually exclusive.

User Response: Reissue the command, specifying only one of these options.

---

module: `error: Conflicting options: '-U' and '-T'`

Module: COOKIE

Explanation: An attempt to execute the specified command failed, because both the UDP and TCP protocols were specified. These options are mutually exclusive.

User Response: Reissue the command, specifying only one of these options.

---

module: `error: Conflicting options: 'Q' and 'h'`

Module: PREV NEXT SHOW

Explanation: The options are in conflict. Q means to be quite, but h means give information.

User Response: Reissue the command without either the Q or h parameter.

module: error: Consecutive delimiters not supported

Module: COOKIE COMPOSE CUSTOM FINGER FOLDER FTP HOST INC LPR NEXT NICNAME PING POST PREV REFILE REPL RMF RMM SCAN SEND SETCLOCK SHOW TFTP

Explanation: An attempt to execute the specified command failed, because two option delimiters (-) were specified on the command line without an intervening option.

User Response: Reissue the command making sure that an option letter follows each option delimiter.

---

module: error: Could not access mail file: *filename: message*

Module: POST

Explanation: The *message* should provide an explanation.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not allocate mail socket: *socket: message*

Module: POST

Explanation: The specified socket could not be allocated at this time.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not create output file *filename*: *message*

Module: NETWATCH

Explanation: An attempt to create the output file *filename* failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not establish telnet session with *host*

Module: TELNET

Explanation: An attempt to execute the specified command failed.

User Response: Make sure that *host* can be resolved into an internet address and that it is active.

---

module: error: Could not find file *filename*: *message*

Module: LPR

Explanation: An attempt to execute the specified command failed, because the file *filename*. could not be found for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not fork packet display task: *message*

Module: NETWATCH

Explanation: An attempt to fork the packet display task failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not fork queue task: *message*

Module: PING

Explanation: An attempt to fork the queue task failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not fork send task: *message*

Module: PING

Explanation: An attempt to fork the send task failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

module: error: Could not open device PRN: *message*

Module: LPR

Explanation: An attempt to execute the specified command failed, because the print device PRN: could not be opened. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not open file *filename*: *message*

Module: LPR

Explanation: An attempt to execute the specified command failed, because *filename* could not be opened for reading. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Could not open requested file: *message*

Module: LPR

Explanation: An attempt to execute the specified command failed, because the requested file could not be opened. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Decrease size and/or range: *message*

Module: PING

Explanation: An attempt to build an ICMP echo request of the requested size and/or range failed for the reason specified in *message*. See the explanation for *message* contained in"Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: DOS error while setting date

Module: SETCLOCK

Explanation: An error occurred somewhere within DOS while attempting to set the system date.

User Response: Reissue the command, and if the condition persists, contact your network administrator.

---

module: error: DOS error while setting time

Module: SETCLOCK

Explanation: An error occurred somewhere within DOS while attempting to set the system time.

User Response: Reissue the command, and if the condition persists, contact your network administrator.

---

module: error: Draft folder must be specified after '-d' parameter

Module: COMPOSE

Explanation: The -d parameter must be followed by a valid folder name.

User Response: Reissue the COMPOSE command with a valid folder name following the -d parameter.

---

module: error: Duplicate or conflicting parameters!

Module: FOLDER

Explanation: Two or more parameters are in conflict or are duplicated.

User Response: Examine the FOLDER command for duplicate or conflicting parameters, correct the problem, and reissue the FOLDER command.

---

module: error: Either host or '-s' must be specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the name of a host was not specified and server mode was not indicated.

User Response: Reissue the command supplying either a host name or the -s option.

---

module: error: Empty packet received

Module: COOKIE

Explanation: The quote server returned a packet of zero length.

User Response: The quote server is not following the quote of the day protocol ad defined in RFC 865. Reissue the command, specifying an RFC compliant quote server.

module: error: Error reading requested file: *message*

Module: LPR

Explanation: An attempt to read the requested file in order to copy it to device PRN: failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Error writing to PRN

Module: LPR

Explanation: An attempt to write the requested file on device PRN: failed.

User Response: Make sure that device PRN: is attached and reissue the command. If the situation persists, contact the appropriate support personnel.

---

module: error: File must be specified after '-f' parameter

Module: COMPOSE

Explanation: The -f parameter requires a valid file name following it.

User Response: Reissue the COMPOSE command with a valid file name following the -f parameter.

---

module: error: Filename must be specified

Module: LPR

Explanation: An attempt to execute the specified command failed, because the name of the file to be printed was not specified.

User Response: Reissue the command and specify the name of the file that should be printed.

---

module: error: Folder must be specified after '+'

Module: COMPOSE FOLDER INC NEXT

Explanation: The + subcommand means change to another folder and must be followed by a folder name.

User Response: Reissue the command with a valid folder name following the '+' subcommand, or omit the + subcommand.

---

module: error: Invalid class value *'value'*

Module: HOST

Explanation: An attempt to execute the specified command failed, because class *value* contains an option delimiter (-) as the first character.

User Response: Reissue the command and include a valid class name after the option.

---

module: error: Invalid option placement

Module: TFTP

Explanation: An attempt to execute the specified command failed, because the neither -p nor -g option was not specified before any other parameters.

User Response: Reissue the command, specifying the -p or -g option prior to any parameters on the command line.

---

module: error: Invalid parameter for negation -- *parameter*

Module: COMPOSE FOLDER INC NEXT POST

Explanation: The parameter shown is either not valid for this command or cannot be negated for this command.

User Response: Reissue the command without the negation (!) of this parameter.

---

module: error: Invalid parameter: *parameter*

Module: FOLDER INC POST

Explanation: The parameter displayed is not valid with this command.

User Response: Reissue the command without the invalid parameter.

---

module: error: Invalid type value *'string'*

Module: HOST

Explanation: An attempt to execute the specified command failed, because the type value *string* was invalid.

User Response: Reissue the command with a valid type value.

---

module: error: Invalid version in template, copy aborted

Module: CUSTOM

Explanation: An attempt to execute the specified command failed, because the internal version of the template custom structure was invalid.

User Response: Reissue the command with a valid version of the template.

module: error: Local printer not ready

Module: LPR

Explanation: The local printer has timed out.

User Response: Determine why print device PRN: has timed out, correct the situation, and reissue the command.

---

module: error: Local printer out of paper

Module: LPR

Explanation: The local printer is either out of paper or not turned on, where the term local printer refers to print device PRN:.

User Response: Correct the situation and reissue the command.

---

module: error: Mailfile must be specified

Module: POST

Explanation: The POST command must have a mail file name specified.

User Response: Reissue the POST command with a valid mail file name.

---

module: error: Mailhome must be specified after '-H' parameter

Module: COMPOSE FOLDER INC NEXT

Explanation: The -H parameter causes the command to use a different Mailhome path, but it must be followed by a valid path.

User Response: Reissue the command with a valid path following the -H parameter.

---

module: error: Message must be specified after '-m' parameter

Module: COMPOSE

Explanation: The -m parameter was found, but no message followed it.

User Response: Reissue the command with a message following the -m parameter.

---

module: error: Missing parameters

Module: TFTP

Explanation: TFTP has not been supplied with enough parameters. The -g, get, -p, and put options require several other parameters.

User Response: Reissue the command using the proper number of parameters.

---

module: error: Multiple files not supported

Module: LPR

Explanation: An attempt to execute the specified command failed, because multiple files were specified. These commands send only a single file to be printed.

User Response: Reissue the command specifying a single file to be printed.

---

module: error: Multiple folders not supported

Module: FOLDER

Explanation: The FOLDER command specifies more than one folder.

User Response: Reissue the command by specifying only one folder.

---

module: error: Multiple hosts not supported

Module: COOKIE FTP PING SETCLOCK

Explanation: An attempt to execute the specified command failed, because more than one host name was specified on the command line.

User Response: Reissue the command supplying a single host name.

---

module: error: No alternate POP port specified after '-a' parameter

Module: INC

Explanation: A valid POP2 host name must follow the -a parameter.

User Response: Reissue the INC command with a valid POP2 host following the -a parameter.

---

module: error: No available (down) hardware tsrs to listen with

Module: NETWATCH

Explanation: An attempt to execute the specified command failed, because the is no downed hardware TSR to listen with. This command requires that the hardware TSR be down.

User Response: Reissue the command after downing the appropriate hardware TSR.

---

module: error: No from field in mail file.

Module: POST

Explanation: SMTP requires information in the FROM: field of the mail to be sent.

User Response: Complete the FROM: field and the reissue the POST command.

module: error: No host specified after '-s' parameter

Module: INC

Explanation: A valid POP2 host must follow the '-s' parameter.

User Response: Reissue the INC command with the name of your POP2 host following the '-s' parameter.

---

module: error: No hostname has been specified

Module: COOKIE REXEC RSH

Explanation: An attempt to execute the specified command failed, because no host name was specified.

User Response: Reissue the command and include the host name.

---

module: error: No parameter specified after !

Module: COMPOSE FOLDER INC NEXT

Explanation: The negation (!) parameter requires a valid parameter following.

User Response: Reissue the command with the parameter that you wish to negate following the !.

---

module: error: No password specified after '-p' parameter

Module: INC

Explanation: A password must follow the -p parameter.

User Response: Reissue the INC command with a password following the -p parameter.

---

module: error: No primary recipients found in address list.

Module: POST

Explanation: SMTP requires a valid recipient in the address list.

User Response: Correct the mail file and reissue the POST command.

---

module: error: No remote folder specified after '-f' parameter

Module: INC

Explanation: A folder name must follow the -f parameter.

User Response: Reissue the INC command with a valid folder name following the -f parameter.

---

module: error: No resolvable name servers specified

Module: HOST

Explanation: An attempt to execute the specified command failed, because a resolvable name server was not specified on the command line and none are specified in the custom structure.

User Response: Reissue the command and specify a name server or modify the custom structure to include at least one name server and then reissue the command.

---

module: error: No target specified

Module: FINGER

Explanation: An attempt to execute the specified command failed, because neither a host name or user@host combination was specified on the command line.

User Response: Reissue the command, specifying a host name or user@host combination.

---

module: error: No time server specified

Module: SETCLOCK

Explanation: An attempt to execute the command failed, because no time server was specified on the command line and there are none defined in the custom structure.

User Response: Reissue the command, and either specify a time server on the command line or include at least one in the custom structure.

---

module: error: No To: field found

Module: POST

Explanation: The SMTP requires information in the TO: field to establish what user is going to receive the mail.

User Response: Complete the To: field and reissue the POST command.

---

module: error: No user specified after '-u' parameter

Module: INC

Explanation: A user name must follow the -u parameter.

User Response: Reissue the INC command with a user name following the -u parameter.

module: error: No valid address found in from field.

Module: POST

Explanation: SMTP requires the mail file to contain a valid FROM: address.

User Response: Correct the mail file using a valid FROM: address and reissue the POST command.

---

module: error: Old or corrupt custom structure

Module: CUSTOM

Explanation: An attempt to customize the custom structure failed because of an incompatibility with CUSTOM. The internal version number for the custom structure differs from that of CUSTOM itself or the custom structure has become corrupted.

User Response: Reissue the command after replacing your version of the custom structure (NETDEV.SYS) with the one supplied with TCP/IP for DOS.

---

module: error: Only one class name may be specified

Module: HOST

Explanation: An attempt to execute the specified command failed, because the class name option was specified more than once on the command line.

User Response: Reissue the command, specifying the class name option only once.

---

module: error: Only one configuration file can be specified

Module: TELNET

Explanation: An attempt to execute the specified command failed, because multiple instances of the f option were specified.

User Response: Reissue the command specifying the f option once.

---

module: error: Only one emulator type can be specified

Module: TELNET

Explanation: An attempt to execute the specified command failed, because multiple instances of the e option were specified.

User Response: Reissue the command specifying the e option once.

---

module: error: Only one of host or '-s' must be specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the name of a host and the server mode were both indicated. These two tasks are mutually exclusive.

User Response: Reissue the command supplying either a host name or the -s option.

---

module: error: Only one type name may be specified

Module: HOST

Explanation: The type name option was specified on the command line more than once.

User Response: Reissue the command using the type name option only once.

---

module: error: Opening service *'service'*: *message*

Module: NETWATCH

Explanation: An attempt to open service *service* failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Option expected

Module: COOKIE CUSTOM FINGER FTP HOST LPR NETWATCH NICNAME PING SETCLOCK TELNET TFTP

Explanation: An attempt to execute the specified command failed, because the option delimiter (-) was encountered without a following option.

User Response: Reissue the command with an option following the delimiter.

---

module: error: Out of memory

Module: COMPOSE FOLDER NEXT POST

Explanation: There is not enough memory to complete the command.

User Response: If possible, unload some terminate and stay resident (TSR) programs and reissue the command.

module: error: Parameter expected

Module: COMPOSE FOLDER INC NEXT POST PREV REFILE REPL RMF RMM SCAN SEND SHOW

Explanation: The command found a - with no parameter after it.

User Response: Reissue the command without the -, or with a valid parameter following the -.

---

module: error: Requested name *'name'* is already in use

Module: NETWATCH

Explanation: Attempt to execute the specified command failed, because the requested name *name* is already in use.

User Response: In order to watch on *name*, use IFCONFIG to down *name*, then reissue the command.

---

module: error: Requested name *'name'* is not a hardware tsr

Module: NETWATCH

Explanation: An attempt to execute the specified command failed, because name *name* was not a hardware TSR. The hardware TSR only needs to be specified if you have more than one hardware installed in your system.

User Response: Use the IFCONFIG command to obtain the names of the resident TCP/IP for DOS TSRs, then reissue the command with the correct name.

---

module: error: Requested name *'name'* is not an installed tsr

Module: NETWATCH

Explanation: An attempt to execute the specified command failed, because the requested name *name* in not an installed TSR.

User Response: Reissue the command with the name of an installed, downed, TSR.

---

module: error: Resetting listen mode for *'service'* :*message*

Module: NETWATCH

Explanation: An attempt to reset listen mode for *service* failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Server closed connection prematurely

Module: LPR

Explanation: An attempt to execute the specified command failed, because the file did not transfer successfully. In all probability, the server closed the connection prematurely.

User Response: Reissue the command. If the situation persists, contact the appropriate server support personnel.

---

module: error: Setting listen mode for service *'service'*:*message*

Module: NETWATCH

Explanation: An attempt to set service *service* to listen mode failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Show program must be specified after '-p' parameter

Module: NEXT

Explanation: The -p parameter must be followed by the name of a valid executable program.

User Response: Reissue the NEXT command with the name of a valid executable SHOW program following the -p parameter.

---

module: error: Statefile must be specified after '-S' parameter

Module: COMPOSE FOLDER INC NEXT

Explanation: The -S parameter must be followed by a valid state file.

User Response: Reissue the command without the -S parameter, or follow it with a valid state file.

---

module: error: Subdirectories are not supported

Module: COMPOSE FOLDER REFILE REPL RMF RMM SCAN SHOW PREV NEXT INC

Explanation: The + parameter expects a folder name, not a DOS path.

User Response: Reissue the command with a folder name following the +.

module: error: The *'character'* option requires an Argument

Module: NETWATCH

Explanation: An attempt to execute the specified command failed, because the *'character'* option requires an argument.

User Response: Reissue the command and include the proper argument after the option.

---

module: error: The '-c' option requires a class name

Module: HOST

Explanation: An attempt to execute the specified command failed, because the -c option requires a class name.

User Response: Reissue the command and include the class name after the option.

---

module: error: The '-e' option requires an emulator type

Module: TELNET

Explanation: An attempt to execute the specified command failed, because the -e option requires the emulator type.

User Response: Reissue the command and include the emulator type after the option.

---

module: error: The '-f' option requires a file name

Module: TELNET

Explanation: An attempt to execute the specified command failed, because the -f option requires the name of the configuration file.

User Response: Reissue the command and include the name of the configuration file after the option.

---

module: error: The '-l' option requires a packet length

Module: PING

Explanation: An attempt to execute the specified command failed, because the -l option requires the length in bytes of the ICMP echo request.

User Response: Reissue the command and include the packet length after the option.

---

module: error: The '-l' option requires a remote login name

Module: REXEC RSH

Explanation: An attempt to execute the specified command failed, because the -l option requires a logon name. The logon name is your logon ID on the remote host.

User Response: Reissue the command and include the logon name after the option.

---

module: error: The '-m' option requires a packet rate

Module: PING

Explanation: An attempt to execute the specified command failed, because the -m option requires the rate, in pings per second, for the ICMP echo requests.

User Response: Reissue the command and include the packet rate after the option.

---

module: error: The '-n' option requires a packet count

Module: PING

Explanation: An attempt to execute the specified command failed, because the -n option requires the number of ICMP echo requests to send.

User Response: Reissue the command and include the packet count after the option.

---

module: error: The '-n' option requires a positive packet count

Module: PING

Explanation: An attempt to execute the specified command failed, because the -n option requires a positive number of ICMP echo requests to send.

User Response: Reissue the command and include a positive packet count after the option.

---

module: error: The '-n' option requires a query type

Module: HOST

Explanation: An attempt to execute the specified command failed, because the -n option requires a query type.

User Response: Reissue the command and include the query type after the option.

---

module: error: The '-p' option requires a password

Module: REXEC RSH

Explanation: An attempt to execute the specified command failed, because the -p option requires a password.

User Response: Reissue the command and include the password after the option.

module: error: The '-p' option requires a port number

Module: TELNET

Explanation: An attempt to execute the specified command failed, because the -p option requires the TCP port number.

User Response: Reissue the command and include the TCP port number after the option.

---

module: error: The '-p' option requires a printer name

Module: LPR

Explanation: An attempt to execute the specified command failed, because the -p option requires a printer name.

User Response: Reissue the command and include the printer name after the option.

---

module: error: The '-r' option requires a maximum packet range

Module: PING

Explanation: An attempt to execute the specified command failed, because the -r option requires the range (size in bytes) of the ICMP echo request.

User Response: Reissue the command and include the maximum packet range after the option.

---

module: error: The '-r' option requires a retry value

Module: host SETCLOCK

Explanation: An attempt to execute the specified command failed, because the -r option requires the number of retries to be attempted in the event of a timeout.

User Response: Reissue the command and include the retry value after the option.

---

module: error: The '-s' option requires a server name

Module: LPR

Explanation: An attempt to execute the specified command failed, because the -s option requires the name of an appropriate server.

User Response: Reissue the command and include the name of the appropriate server after the option.

---

module: error: The '-t' option requires a time-out value

Module: COOKIE FINGER HOST NICNAME PING SETCLOCK

Explanation: An attempt to execute the specified command failed, because the -t option requires the number of seconds which determine when a request or connection is timed out.

User Response: Reissue the command and include the time-out value after the option.

---

module: error: The '-u' option requires a local user name

Module: RSH

Explanation: An attempt to execute the specified command failed, because the -u option requires a local user name.

User Response: Reissue the command and include a local user name after the option.

---

module: error: The '-w' option requires a wait count

Module: PING

Explanation: An attempt to execute the specified command failed, because the -w option requires the number of seconds to wait before sending an ICMP echo request.

User Response: Reissue the command and include the wait count after the option.

---

module: error: The duplicate use of the '-p' option is not supported

Module: TELNET

Explanation: An attempt to execute the specified command failed, because multiple instances of the -p option were specified.

User Response: Reissue the command specifying the -p option once.

---

module: error: The invalid maximum packet range *'range'* was specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the packet range *range* was illegal. Legal packet ranges are positive integers.

User Response: Reissue the command, specifying a legal packet range.

module: error: The invalid packet length *'length'* was specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the packet *length* specified was illegal. Legal values for *length* are positive integers less than 32 767.

User Response: Reissue the command, specifying a legal packet length.

---

module: error: The invalid packet rate *'rate'* was specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the packet *rate* was illegal. Legal values for *rate* are positive integers less than 32 112.

User Response: Reissue the command, specifying a legal packet rate.

---

module: error: The invalid retry value *'value'* was specified

Module: HOST SETCLOCK

Explanation: An attempt to execute the specified command failed, because the retry value was illegal. Legal values for retry are positive integers.

User Response: Reissue the command, specifying a legal retry value.

---

module: error: The invalid time-out value *'value'* was specified

Module: COOKIE FINGER HOST NICNAME PING SETCLOCK

Explanation: The time-out value specified on the command line was out of range.

User Response: Reissue the command, specifying a legal time-out value.

---

module: error: The invalid wait count *'value'* was specified

Module: PING

Explanation: An attempt to execute the specified command failed, because the wait count specified was illegal. Legal wait counts are positive integers.

User Response: Reissue the command, specifying a legal wait count.

---

module: error: The port number must be a positive integer

Module: TELNET

Explanation: An attempt to execute the specified command failed, because the TCP port number was not a positive integer.

User Response: Reissue the command with a positive integer following the -p option.

---

module: error: Timed out connecting to host

Module: FINGER

Explanation: An attempt to execute the specified command failed, because the specified host did not respond during the connection period.

User Response: Reissue the command an include the -t option with a larger time-out value. If the situation persists, contact your network administrator.

---

module: error: Timed out terminating transmission

Module: FINGER

Explanation: The specified command timed out while attempting to terminate the transmission. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Too many devices/files specified

Module: CUSTOM

Explanation: An attempt to execute the specified command failed, because more than two parameters were indicated on the command line. The first parameter should be the device and file to be customized, and the second, a template to be overlayed onto the first.

User Response: Reissue the command specifying only two parameters on the command line.

---

module: error: Too many targets specified

Module: FINGER

Explanation: An attempt to execute the specified command failed, because more than 10 host or user@host combinations were specified.

User Response: Reissue the command reducing the number of host or user@host combinations.

module: error: Too many tsr names to watch on

Module: NETWATCH

Explanation: An attempt to execute the specified command failed, because too many TSR names were specified.

User Response: Reissue the command and reduce the number of TSR names to watch.

---

module: error: Transmitting to broadcast address not supported

Module: LPR PING TFTP

Explanation: An attempt to execute the specified command failed, because it attempted to transmit to a broadcast address.

User Response: No response is necessary.

---

module: error: Transmitting to local machine not supported

Module: : LPR PING TFTP

Explanation: An attempt to execute the specified command failed, because the local machine was specified as the object. These programs can only transmit to remote hosts.

User Response: Reissue the command, specifying a remote host.

---

module: error: Unable to allocate socket: *message*

Module: COOKIE FINGER HOST LPR NICNAME SETCLOCK

Explanation: An attempt to allocate a socket failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to allocate storage: *message*

Module: CUSTOM HOST LPR PING TFTP

Explanation: An attempt to allocate storage failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to bind socket: *message*

Module: FINGER HOST LPR SETCLOCK

Explanation: An attempt to bind a local address to a socket failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to close connection: *message*

Module: FINGER

Explanation: An attempt to close the connection failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator. error: Unable to compute file size:*messsage*

---

module: error: Unable to connect: *message*

Module: COOKIE FINGER LPR NICNAME

Explanation: An attempt to connect failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to determine local address: *message*

Module: LPR

Explanation: An attempt to determine the internet address of the local machine failed for the reason specified in *message*. See the explanation for message contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

module: error: Unable to determine printserver

Module: LPR

Explanation: An attempt to execute the specified command failed, because an LPD printer server was not specified on the command line and one was not specified in the custom structure.

User Response: Reissue the command and specify a printer server or modify the custom structure to include one. Note that LPR and QPR share the printer server specified in the custom structure.

---

module: error: Unable to initialize icmp protocol layer:*message*

Module: PING

Explanation: An attempt to initialize the ICMP protocol layer failed for the reason specified in *message*. See the explanation for message contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to load secondary command processor:*message*

Module: PING TFTP

Explanation: An attempt to load a secondary command processor failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to receive time response: *message*

Module: SETCLOCK

Explanation: An attempt to receive a time response failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to receive: *message*

Module: COOKIE FINGER NICNAME LPR

Explanation: An attempt to receive data failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to resolve address*address* : *nameserver*

Module: COOKIE FINGER HOST LPR NICNAME PING SETCLOCK TFTP

Explanation: An attempt to resolve the address *address* failed for the reason specified in *message*. See the explanation for message contained in "Name Server Messages" on page 302 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to set socket to non-blocking:*message*

Module: FINGER

Explanation: An attempt to set a socket to nonblocking mode failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to terminate transmission: *message*

Module: FINGER

Explanation: An attempt to terminate transmissions failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

module: error: Unable to transmit icmp echo request: *message*

Module: PING

Explanation: An attempt to transmit the ICMP echo request failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to transmit: *message*

Module: COOKIE FINGER LPR NICNAME

Explanation: An attempted transmission failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait for acknowledgment: *message*

Module: FINGER

Explanation: An attempt to execute the specified command failed, because it could not wait for an acknowledgment for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait for reply: *message*

Module: COOKIE FINGER NICNAME

Explanation: An attempt to execute the specified command failed, because the specified reason prevented it from waiting for a reply. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait for shutdown acknowledgment: *message*

Module: FINGER

Explanation: An attempt to wait for a shutdown acknowledgment failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait to close connection: *message*

Module: FINGER

Explanation: An attempt to wait to close the connection failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait to connect: *message*

Module: FINGER

Explanation: An attempt to wait for the connection failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: Unable to wait to receive: *message*

Module: FINGER

Explanation: An attempt to wait for a response failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

module: error: Unable to write to temporary mail file: *mailfile*

Module: SEND

Explanation: The temporary mail file has a read only attribute, or it is located on a write protected drive.

User Response: Specify a different mail file, or remove the file or drive protection.

---

module: error: Unbalanced [] in *'hostname'*

Module: FINGER

Explanation: An attempt to execute the specified command failed, because brackets ([ ]) were not balanced. When specifying a host as an internet literal, for example [hostname], you must provide a] for each [.

User Response: Reissue the command with the correct command line syntax.

---

module: error: Unknown emulator name: *'emulator name'*

Module: TELNET

Explanation: An attempt to execute the specified command failed, because *emulator name* is not a valid emulator type.

User Response: See the TELNET command in *IBM TCP/IP Version 2.0 for DOS: User's Guide* for the valid emulator types.

---

module: error: Unknown option : *character*

Module: CUSTOM COOKIE FINGER FTP HOST LPR NETWATCH NICNAME PING SETCLOCK TELNET TFTP

Explanation: An attempt to execute the specified command failed, because option *character* is not available with this command.

User Response: Reissue the command only using valid options. If necessary, use the help option (-?) to obtain a list of all the valid command options.

---

module: error: Unknown parameter: *parameter*

Module: COMPOSE FOLDER INC NEXT REFILE REPL RMM SCAN SEND SHOW

Explanation: The parameter is not a valid parameter with this command.

User Response: Reissue the command without *parameter*.

---

module: error: Unknown transfer mode specified

Module: TFTP

Explanation: An attempt to execute the specified command failed due to an unknown transfer mode being specified on the command line. The transfer mode must be specified as the 5th argument. Supported transfer modes are ASCII and BINARY.

User Response: Reissue the command specifying the 5th argument (transfer mode) as either ASCII or BINARY.

---

module: error: Upping service *'name'*: *message*

Module: NETWATCH

Explanation: An attempt to up the service *name* failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: error: You must specify a host and command

Module: REXEC RSH

Explanation: An attempt to execute the specified command failed, because neither a remote host and command were not specified.

User Response: Reissue the command, specifying a remote host and command.

---

module: error: You must specify a name to query

Module: HOST

Explanation: An attempt to execute the specified command failed, because neither a name to be resolved into an internet address nor an internet address which to be resolved into the corresponding host name were specified.

User Response: Reissue the command specifying either a host name or an internet address to be resolved.

---

module: error: You must specify at least one domain name server

Module: HOST

Explanation: An attempt to execute the specified command failed, because a domain name server was not specified on the command line and none are specified in the custom structure.

User Response: Reissue the command and specify a domain name server or modify the custom structure to include at least one domain name server and then reissue the command.

module: error: You must specify at least one IEN-116 nameserver

Module: HOST

Explanation: An attempt to execute the specified command failed, because a IEN-116 name server was not specified on the command line and none are specified in the custom structure.

User Response: Reissue the command and specify a IEN-116 name server or modify the custom structure to include at least one IEN-116 name server and then reissue the command.

---

module: error: You must specify the name you wish resolved

Module: HOST

Explanation: An attempt to execute the specified command failed, because a host name to be resolved into an internet address was not supplied on the command line.

User Response: Reissue the command, specifying the name you wish to have resolved using a IEN-116 name server.

---

module: error: You must specify the structure and template

Module: CUSTOM

Explanation: An attempt to execute the specified command failed, because the -c (copy) option requires the name of the custom structure and template.

User Response: Reissue the command and include the names of the custom structure and template.

General Module Internal Errors

The following internal error messages can appear in different modules.

---

module: INTERNAL ERROR: Bad return from getopt(): *value*

Module: COOKIE CUSTOM FTP FINGER HOST LPR NETWATCH NICNAME PING REXEC RSH TFTP SETCLOCK

Explanation: A normally unreachable section of code has been accessed.

User Response: Report this occurrence to your TCP/IP for DOS support personnel.

---

module: INTERNAL ERROR: main: Impossible condition: Report this occurrence to DOS/IP support

Module: TFTP

Explanation: A normally unreachable section of code has been accessed.

User Response: Please contact your TCP/IP for DOS support personnel and provide any information about the conditions under which this error occurred.

General Module Warnings

The following are warning messages that can appear in different modules.

---

module: warning: Address for host *hostname* not found.

Module: POST

Explanation: No name server was able to supply an address for *hostname*

User Response: Reissue the POST command with the correct IP address.

---

module: warning: Bad address *address* detected and ignored.

Module: POST

Explanation: Mail could not be delivered to *address*.

User Response: Reissue the POST command with a correct address for the *address* shown.

---

module: warning: Binary printing ignored on UNIX LPD print server

Module: LPR

Explanation: The -b option directing binary file printing was specified when printing was to be performed by a remote UNIX LPD print server. The -b option is valid only for local printing.

User Response: If binary file printing is required, reissue the command and specify a local printer. If binary file printing is not required, reissue the command removing the -b option.

---

module: warning: Can not get data from console, issuing quit command

Module: FTP

Explanation: A Ctrl-Z was entered while in the FTP shell.

User Response: No response is necessary.

---

module: warning: Connection timed out

Module: FINGER NICNAME

Explanation: An attempt to execute the specified command failed, because the time-out period expired before receiving any data.

User Response: Reissue the command, and if possible, increase the timeout period.

---

module: warning: Could not connect to host *hostname*

Module: POST

Explanation: The *hostname* was not running an SMTP server.

User Response: Reissue the POST command at a later time, or if a POP2 server is available, SEND the mail to the POP2 server and let it deliver the mail to *hostname*.

---

module: warning: Could not get address for *mail gateway* to deliver mail to *name*

Module: POST

Explanation: The *mail gateway* could not be located. No name server could provide an address for the *mail gateway*.

User Response: Correct the *mail gateway* name and reissue the POST command.

---

module: warning: Could not send mail to *address*

Module: POST

Explanation: POST attempted to deliver mail to *address* using SMTP, but the mail could not be delivered.

User Response: Examine the address carefully. If everything appears to be correct, reissue the POST command.

---

module: warning: Date field ignored in mail draft

Module: SEND

Explanation: The SEND command found that the DATE: field already contained a date. The SEND command ignored the DATE: field and stamped the field with the proper date.

User Response: No action is required.

---

module: warning: Error transmitting mailfile: *message*

Module: POST

Explanation: The contents of *message* should explain this error.

User Response: Follow the action recommended by the system.

---

module: warning: Host *address* reports an internal error

Module: HOST

Explanation: The domain name server *address* was unable to process the query because of an internal problem.

User Response: Reissue the command directing it to a different domain name server, or wait a period of time

and reissue the command to the same domain name server. If the problem persists contact your network administrator.

---

module: warning: Host *address* returns a format error

Module: HOST

Explanation: The domain name server *address* was unable to interpret the query.

User Response: Reissue the command directing it to a different domain name server, or wait a period of time and reissue the command to the same domain name server. If the problem persists contact your network administrator.

---

module: warning: *host* has ended this FTP session.

Module: FTP

Explanation: The remote *host* has ended the FTP session.

User Response: If more information is required, contact the responsible support personnel for the remote host's FTP service.

---

module: warning: Host *hostname* rejected DATA command.

Module: POST

Explanation: The *hostname* has rejected the data transmission.

User Response: Reissue the POST command. If the same error continues to occur, ensure that you can POST mail to another host, if possible, and that other stations can POST mail to *hostname*.

---

module: warning: Host *hostname* rejected delivery address *address*.

Module: POST

Explanation: The *hostname* responded to SMTP, but rejected mail delivery.

User Response: Ensure that the address in the mail file is valid for *hostname*.

---

module: warning: Host *hostname* rejected HELO command

Module: POST

Explanation: The remote host has refused to accept mail from this system at this time.

User Response: Contact the network administrator of the remote host to determine why it is rejecting mail from your user ID.

---

module: warning: Host *hostname* rejected mail file after it was transmitted

Module: POST

Explanation: The remote host has refused mail from your system after it was delivered.

User Response: Contact the network administrator of the remote host to determine why it is rejecting mail from your user ID.

---

module: warning: Host *hostname* rejected MAIL FROM command.

Module: POST

Explanation: The remote host has refused to accept mail from this system at this time.

User Response: Contact the network administrator of the remote host to determine why it is rejecting mail from your user ID.

---

module: warning: Maximum number of arguments exceeded, stopping at argument *number 'command'*

Module: FTP

Explanation: An attempt was made to execute more than 100 FTP commands in a command file. The last command executed was *command* which is number *value* in the file.

User Response: None necessary.

---

module: warning: Name server *address* does not support recursive queries

Module: HOST

Explanation: Recursive query support is optional on the part of domain name servers.

User Response: Reissue the command and direct the query to a domain server which provides recursive query support.

---

module: warning: Name server *address* refuses to answer the question

Module: HOST

Explanation: Name server *address* refused to respond to the query for policy reasons. For example, the domain name server might not want to provide the information to your particular machine, or it might not perform that particular operation such as a zone transfer.

User Response: Reissue the command and direct the query to a domain name server that provides the information you want. *'name'* does not exist

module: warning: No CMOS clock found. Ignoring '-c' option

Module: SETCLOCK

Explanation: The -c option was specified on an IBM PC or PC/XT. These machines do not come with a CMOS clock. This option is ignored.

User Response: No response is necessary.

---

module: warning: No response from *address*

Module: HOST

Explanation: The domain name server *address* did not respond. This situation can be due to the domain name server crashing or from exceeding the time-out period.

User Response: Reissue the command after determining that the domain name server at *address* is operational. If the domain name server is operational, increase the time-out period using the -t option.

---

module: warning: No valid addresses found in sender field of mail header

Module: POST

Explanation: The SMTP header is not correct.

User Response: Correct the mail header and then reissue the POST command.

---

module: warning: Only one tftp server may be active at a time.

Module: TFTP

Explanation: An attempt was made to execute the specified command while in a secondary command processor. Only one server can be active.

User Response: No response is necessary.

---

module: warning: Server*address*: *message*

Module: HOST

Explanation: An attempt to obtain name resolution failed, because name server *address* reported a problem. See the explanation for *message* contained in "Name Server Messages" on page 302 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: warning: Time service not responding, Clock not set.

Module: SETCLOCK

Explanation: An attempt to execute the specified command failed, because there was no response from any of the time servers. Time servers may not have responded, because they are down or are not currently providing time service.

User Response: Reissue the command, specifying a different time server.

---

module: warning: Timed out closing connection

Module: FINGER

Explanation: The specified command timed out while attempting to close the transmission. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: warning: Timed out receiving data:*message*

Module: FINGER

Explanation: The specified command timed out while attempting to receive data. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: warning: Timed out sending to host

Module: FINGER

Explanation: The specified command timed out while attempting to send data.

User Response: Reissue the command. If the situation persists, contact your local TCP/IP for DOS support personnel.

---

module: warning: Timed out waiting for reply

Module: POST

Explanation: You have timed out waiting for a reply from the remote host.

User Response: If this error continues to occur, then contact the network administrator of the remote host to determine why it is slow to respond.

module: warning: Unable to allocate local resources: *message*

Module: HOST

Explanation: An attempt to allocate local resources when attempting name resolution failed for the reason specified in *message*. See the explanation for *message* found in "Name Server Messages" on page 302 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: warning: Unable to transmit: *message*

Module: SETCLOCK

Explanation: An attempt to transmit failed for the reason specified in *message*. See the explanation for *message* found in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

---

module: warning: Unknown error message (*value*) from dm_resolve

Module: HOST

Explanation: The domain name server returned an unknown error code, *value*.

User Response: Contact your network administrator or your TCP/IP for DOS support personnel for more information.

---

module: warning: Unrecognized class type *'string'* will be ignored

Module: HOST

Explanation: The class type specified by *string* is not recognized as a legitimate class and is ignored. Only class mnemonics as specified in RFC 1035 are recognized.

User Response: Reissue the command after consulting *IBM TCP/IP Version 2.0 for DOS: User's Guide* to obtain the valid classes.

---

module: warning: Unrecognized query type *'string'* will be ignored

Module: HOST

Explanation: The type of query specified by *'string'* is not recognized as a legitimate query type and is ignored. All query type mnemonics defined in RFC 1035 with the exception of AXFR are legitimate.

User Response: Reissue the command after consulting *IBM TCP/IP Version 2.0 for DOS: User's Guide* to obtain the valid query type.

---

module: warning: Error waiting for tftp connection: *message*

Module: TFTP

Explanation: An attempt to initialize the command failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

Generic Text Messages

The following generic text messages can appear in different modules.

`Address already in use`

Explanation: A socket has already been bound to an address.

`Address family not supported by protocol family`

Explanation: Attempt was made to specify an address family other than AF_INET.

`Arg list too long`

Explanation: The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32K bytes.

`Argument too large`

Explanation: The argument to a math function is not in the domain of the function.

`Bad memory address`

Explanation: The system encountered a memory fault. Since TCP/IP does not have memory protection, some system calls check the validity of pointers passed to them. This error occurs when a pointer check reveals that the passed pointer is zero (0).

`Bad file number`

Explanation: The specified file handle is not a valid file handle value, does not refer to an open file, or an attempt was made to write to a file or device opened for read access (or the reverse).

`Block device required`

Explanation: A character device was specified where a block device is required.

`Broken pipe`

Explanation: Attempt was made to write on a socket or pipe whose foreign side is no longer accepting data.

`Can't assign requested address`

Explanation: Attempt was made to assign an address to an unsupported socket or an address could not be resolved.

`Can't send after socket shutdown`

Explanation: Attempt was made to read or write on a socket which has been shutdown.

`Connection refused`

Explanation: Attempt at connecting to a peer was refused. This can result from trying to connect to a service that is not supported or by exceeding the maximum number of connections.

`Connection reset by peer`

Explanation: A stream connection was forcibly closed (reset) by a peer host. This happens, for example, when the remote socket loses the connection due to a time-out or reboot.

`Connection timed out`

Explanation: A request to connect or send to a peer failed, because they did not correctly respond for a period of time. The time-out period is dependent on the specific protocol used, i.e. stream or datagram.

`Cross-device link`

Explanation: Attempt was made to move a file to a different device (using the rename() function).

`Destination address required`

Explanation: Attempt was made to perform an operation on a socket without specifying the destination address.

`Directory not empty`

Explanation: Attempt was made to remove a directory which is not empty.

`Disc quota exceeded`

Explanation: Attempt to write to a file or create a directory or link failed, because your quota of storage was exhausted.

`Error reading or writing the NETCUST device`

Explanation: Attempt to read from or write to the device NETCUST failed.

`Exec format error`

Explanation: Attempt was made to run a file that was not executable or that had a non-valid executable file format.

File exists

Explanation: The O_CREAT and O_ESCL flags were specified when opening a file, but the named file already existed.

File name too long

Explanation: The file portion of a path was longer than twelve (12) characters (eight for the file name itself, one for a period, and three for the extension) or the length of the path was greater than 64 characters.

File table overflow

Explanation: The system's table of open files is full; no more requests to open a file can be accepted until a previously open file is closed.

File too large

Explanation: The size of the specified file exceeds the maximum (approximately 2G bytes).

Host is down

Explanation: Attempt was made to perform a socket operation to a host which was down.

I/O error

Explanation: An I/O error occurred when attempting to write to a network device.

Illegal seek

Explanation: Attempt was made to perform a seek on a socket or pipe.

Interface is deaf now

Explanation: Attempt was made to disable packet reception on a network interface that was already disabled.

Interface is down now

Explanation: Attempt was made to bring an interface down which was already down.

Interface is hearing now

Explanation: Attempt was made to enable packet reception on an network interface that was already enabled.

Interface is installed now

Explanation: Attempt was made to install a network interface which was already installed.

Interface is up now

Explanation: Attempt was made to bring an interface up which was already up.

Interrupted system call

Explanation: An interrupt occurred during a system call.

Invalid argument

Explanation: An invalid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file.

IOCTL failed

Explanation: Attempt to perform an IOCTL on the NETCUST device failed.

Is a directory

Explanation: Attempt was made to write to a directory.

Message too long

Explanation: Attempt was made to send a packet which was larger than the internal message buffer or some other limit through a socket.

Mount device busy

Explanation: Attempt was made mount a device which is already in use or an attempt was made to unmount a drive which was the current drive, has been substituted or joined.

Network dropped connection on reset

Explanation: The foreign host has rebooted.

Network is down

Explanation: There is no access to the network.

Network is unreachable

Explanation: Attempt was made to perform a network operation to an unreachable network.

No buffer space available

Explanation: Attempt to perform an operation on a socket failed due to a lack of buffer space.

No children

Explanation: Attempt to initiate the tasking system or to create a child process failed.

No error condition exists

Explanation: Value indicating no error.

No INET TSR has been loaded

Explanation: Attempt to use the INET TSR failed as it has not been loaded into memory.

No more processes

Explanation: Attempt to add a child process failed as the maximum number of tasks allowed has been reached.

No room for additional entry

Explanation: Attempt to allocate resources for network access failed due to lack of resources, for example, the maximum number of connections has been reached.

No route to host

Explanation: Attempt was made to perform a socket operation to an unreachable host.

No space left on device

Explanation: No more space for writing is available on the device. (For example, the disk is full).

No such device

Explanation: An invalid or non-existant device was supplied as an argument on a system call.

No such device or address

Explanation: The specified device did not exist.

No such file or directory

Explanation: The specified file or directory does not exist or could not be found. This message can occur whenever a specified file did not exist or a component of a path name does not specify an existing directory. It can also occur if a file name exceeds 8 characters, or if the file name extension exceeds 3 characters.

No such process

Explanation: The specified process did not exist.

No Utility TSR has been loaded

Explanation: Attempt to use the Utility TSR failed as it had not been loaded into memory.

Not a directory

Explanation: A directory name was not specified where one is required, for example, in a path name.

Not a typewriter

Explanation: The specified device was not a typewriter or a device to which this call applies.

Not enough core

Explanation: Not enough storage was available. This message can occur when not enough storage was available to run a child process or when the allocation request in an sbrk() or getcwd() call cannot be satisfied.

Not owner

Explanation: Attempt was made to modify a file in such a way forbidden by anyone except its owner.

Operation already in progress

Explanation: Attempt was made to perform an operation on a nonblocking object which already has an operation in progress. For example, attempting to close a TCP connection which is in the process of being closed.

Operation not supported on socket

Explanation: Attempt was made to perform an operation on a socket which is not supported. For example, trying to accept a connection on a socket of type SOCK_DGRAM.

Operation now in progress

Explanation: Attempt was made to perform an operation which takes a long time to complete on a nonblocking object.

Operation would block

Explanation: Attempt was made to perform an operation on an object in nonblocking mode which would cause a process to block. For example, attempting to read from a socket created as nonblocking when no information is present.

Operation would cause deadlock

Explanation: Locking violation: the file cannot be locked after 10 attempts.

Permanent failure of the network driver

Explanation: The network driver was not functional. Examine the network hardware and confirm that it is functioning properly and that it is configured correctly.

Permission denied

Explanation: The permission setting of the file does not allow the specified access. This error can occur in a variety of circumstances. It signifies that an attempt was made to get access to a file in a way that is incompatible with the attributes of the file.

Protocol family not supported

Explanation: Attempt was made to use a protocol family which has been neither configured nor implemented into the system.

Protocol not available

Explanation: Attempt was made to specify a bad option or level in a getsockopt() or setsockopt() call.

Protocol not supported

Explanation: Attempt was made to use a protocol which has neither been configured nor been implemented into the system.

Protocol wrong type for socket

Explanation: A protocol was specified which is not supported by the requested socket type. For example, the UDP protocol was specified for socket type SOCK_STREAM.

Read-only file system

Explanation: Attempt was made to write to a read-only file.

Result too large

Explanation: An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the path name argument to the getcwd() function is longer than expected).

RVD related disk error

Explanation: An unknown error was generated by an RVD disk operation.

Service agent table is full

Explanation: Attempt to load a service into the service table failed as it would exceed the maximum number allowed.

Socket is already connected

Explanation: A request was made to connect to an already connected socket or a sendto() call was attempted with a socket of type TCP.

Socket is not connected

Explanation: A request was made to send or receive data on a socket which was not connected.

Socket operation on non-socket

Explanation: Attempt was made to perform a socket operation on a non-socket or the socket type is not valid.

Socket type not supported

Explanation: Attempt was made to use a socket type which has either not been configured or implemented into the system.

Software caused connection abort

Explanation: A reset was sent to the foreign host due to some internal condition.

Failure on the part of the network driver

Explanation: Failure on the part of the network driver to deliver the requested packet to the wire.

Text file busy

Explanation: Attempt has been made to modify a pure-procedure program that is currently being executed.

Too many levels of symbolic links

Explanation: Attempt was made to lookup a path name which contains greater than the maximum number of symbolic links.

Too many links

Explanation: Attempt was made to exceed the maximum number of links to a file.

Too many open files

Explanation: No more file handles are available, so no more files can be opened.

Too many processes

Explanation: Attempt was made to fork a task which would overflow the process table.

Too many references: can't splice

Explanation: The limit of files has been exhausted.

Too many users

Explanation: The system call could be completed, because too many users were logged into the system.

Unknown error condition

Explanation: An indeterminable error occurred.

Unclean situation (you clean it up)

Explanation: An unforeseen situation has occurred. Perform any necessary cleanup. This error is most notably caused by the death of a process.

UTIL TSR is not up

Explanation: Attempt to use UTIL failed, because it has not been loaded.

IFCONFIG Errors

The following error messages can appear when using the IFCONFIG command.

`ifconfig: error: downing hardware tsr: Interface is down now`

Module: IFCONFIG

Explanation: The IFCONFIG *hwr* DOWN command was issued when the hardware TSR has been terminated.

User Response: Entering the IFCONFIG *hwr* DOWN command when the hardware TSR has been terminated has no effect.

`ifconfig: error: downing utility tsr: Interface is down now`

Module: IFCONFIG

Explanation: The IFCONFIG UTIL DOWN command was issued when UTIL has been terminated.

User Response: Entering the IFCONFIG UTIL DOWN command when UTIL has been terminated has no effect.

`ifconfig: error: hwr: 'address' requires an internet address/mask`

Module: IFCONFIG

Explanation: The IFCONFIG INET IP CONFIG *hwr* ADDRESS command must be followed by a valid internet address.

User Response: Reissue the command with a valid internet address following the address parameter.

`ifconfig: error: hwr: base: requires a hexadecimal argument`

Module: IFCONFIG

Explanation: The base port address parameter was either missing or not valid.

User Response: Reissue the command using a valid base port address following the base parameter.

`ifconfig: error: hwr: baud: requires one of the following values: 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 4800, 9600, 19200, 38400`

Module: IFCONFIG

Explanation: The baud rate parameter was either missing or not valid.

User Response: Reissue the command using a valid baud rate following the baud parameter.

`ifconfig: error: hwr: 'broadcast' requires an internet address/mask`

Module: IFCONFIG

Explanation: The IFCONFIG INET IP CONFIG *hwr* BROADCAST command must be followed by a valid internet address.

User Response: Reissue the command with a valid internet address following the broadcast parameter.

`ifconfig: error: hwr: com: requires a value in the range (1 - 8)`

Module: IFCONFIG

Explanation: The COM port parameter was either missing or not valid.

User Response: Reissue the command using a valid COM port following the COM parameter.

`ifconfig: error: hwr: data: requires a value in the range (5 - 8)`

Module: IFCONFIG

Explanation: The number of data bits parameter was either missing or not valid.

User Response: Reissue the command using a valid data bits number following the data parameter.

`ifconfig: error: hwr: dialtmo: requires a value in the range (0 - 65535)`

Module: IFCONFIG

Explanation: The dial-up time-out parameter was either missing or not valid.

User Response: Reissue the command using a valid dial-up time-out following the dialtmo parameter.

`ifconfig: error: hwr: flags: 'route' requires you to specify ON or OFF`

Module: IFCONFIG

Explanation: The only valid route flags are on and off. One of these must be specified.

User Response: Reissue the command using either on or off following the FLAGS ROUTE parameters.

`ifconfig: error: hwr: int: requires a value in the range (0x60 - 0x80)`

Module: IFCONFIG

Explanation: The interrupt vector parameter was either missing or not valid.

User Response: Reissue the command using a valid interrupt vector following the int parameter.

---

ifconfig: error: *hwr*: mtu: requires a value in the range (nnn - nnn)

Module: IFCONFIG

Explanation: The parameter following the mtu parameter is missing or not valid.

User Response: Reissue the command using a valid mtu value following the mtu parameter.

---

ifconfig: error: *hwr*: 'netmask' requires an internet address/mask

Module: IFCONFIG

Explanation: The IFCONFIG INET IP CONFIG *hwr* NETMASK command must be followed by a valid internet address.

User Response: Reissue the command with a valid internet address following the netmask parameter.

---

ifconfig: error: *hwr*: '*parameter*' is an ill formed address

Module: IFCONFIG

Explanation: The *parameter* must be a valid internet address in dotted decimal format or hexadecimal format.

User Response: Reissue the command, replacing the invalid parameter with a valid internet address.

---

ifconfig: error: *hwr*: '*parameter*' is an unknown command for this service

Module: IFCONFIG

Explanation: The *parameter* is not valid with the *hwr* subcommands.

User Response: See *IBM TCP/IP Version 2.0 for DOS: User's Guide* for a list of valid parameters for the *hwr* subcommands.

---

ifconfig: error: *hwr*: '*parameter*' is an unknown flag

Module: IFCONFIG

Explanation: The parameter following the flags parameter is not valid. The only valid parameter that follows flags is route.

User Response: Reissue the command with the route parameter and a valid route flag following the flags parameter.

---

ifconfig: error: *hwr*: parity: type '*parameter*' invalid

Module: IFCONFIG

Explanation: The parity type parameter was not valid.

User Response: Reissue the command using a valid parity type following the parity parameter. The valid parity types are: none, odd, even, bit1, and bit0.

---

ifconfig: error: *hwr*: parity: requires a parity type

Module: IFCONFIG

Explanation: The parity type parameter was missing.

User Response: Reissue the command using a valid parity type following the parity parameter. The valid parity types are: none, odd, even, bit1, and bit0.

---

ifconfig: error: *hwr*: stop: requires a value in the range (1 - 2)

Module: IFCONFIG

Explanation: The number of stop bits parameter was either missing or not valid.

User Response: Reissue the command using a valid stop bits number following the stop parameter.

---

ifconfig: error: inet: 'ackdelay' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new ackdelay was not valid.

User Response: Reissue the command using an ackdelay value between 1 and 65 535 ticks.

---

ifconfig: error: inet: config does not find an interface named '*parameter*'

Module: IFCONFIG

Explanation: Before a network interface can be configured using the IFCONFIG INET IP CONFIG command, it must be created.

User Response: Create the network interface using the IFCONFIG INET IP CREATE command, then reissue the IFCONFIG INET IP CONFIG command.

---

ifconfig: error: inet: 'default_mss' value '*parameter*' out of range (1 - 65 535)

Module: IFCONFIG

Explanation: The *parameter* for the new default_mss was not valid.

User Response: Reissue the command using a default_mss value between 1 and 65 535 bytes.

ifconfig: error: inet: delete does not find an interface named 'pv.'parameter'

Module: IFCONFIG

Explanation: The interface *parameter* does not exist.

User Response: Use the IFCONFIG INET IP SHOW command to see a list of active network interfaces. Reissue the IFCONFIG INET IP DELETE command using a valid network interface name.

---

ifconfig: error: inet: 'far_rtt' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new far_rtt was not valid.

User Response: Reissue the command using a far_rtt value between 1 and 65 535 ticks.

---

ifconfig: error: inet: 'flushsqtmo' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new flushsqtmo was not valid.

User Response: Reissue the command, using a flushsqtmo value between 1 and 65 535 ticks.

---

ifconfig: error: inet: 'ip delete' requires an interface name

Module: IFCONFIG

Explanation: The name of the network interface to be deleted was not given.

User Response: Use the IFCONFIG INET IP SHOW command to see a list of active network interfaces. Reissue the IFCONFIG INET IP DELETE command using a valid network interface name.

---

ifconfig: error: inet: 'keep_tmo' value '*parameter*' out of range (1 - 65 535)

Module: IFCONFIG

Explanation: The *parameter* for the new keep_tmo was not valid.

User Response: Reissue the command using a keep_tmo value between 1 and 65 535 ticks.

---

ifconfig: error: inet: legal operations for route are 'add','delete', or 'flush'

Module: IFCONFIG

Explanation: The add, delete, or flush parameter must follow the IFCONFIG INET IP ROUTE command.

User Response: Reissue the command with a valid parameter following ROUTE.

---

ifconfig: error: inet: legal types for route add are 'net', 'default' or 'host'.

Module: IFCONFIG

Explanation: The net, default, or host parameter must follow the IFCONFIG INET IP ROUTE ADD command.

User Response: Reissue the command with a valid parameter following ADD.

---

ifconfig: error: inet: legal types for route delete are 'net', 'default' or 'host'.

Module: IFCONFIG

Explanation: The net, default, or host parameter must follow the IFCONFIG INET IP ROUTE command.

User Response: Reissue the command with a valid parameter following DELETE.

---

ifconfig: error: inet: 'max_keep' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new max_keep was not valid.

User Response: Reissue the command using a max_keep value between 1 and 65 535 transmissions.

---

ifconfig: error: inet: 'maxretries' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new maxretries was not valid.

User Response: Reissue the command using a maxretries value between 1 and 65 535 ticks.

---

ifconfig: error: inet: 'near_rtt' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new near_rtt was not valid.

User Response: Reissue the command using a near_rtt value between 1 and 65 535 ticks.

---

ifconfig: error: inet: '*parameter*' is an unknown command for this service

Module: IFCONFIG

Explanation: The *parameter* is not valid with the INET subcommand.

User Response: See *IBM TCP/IP Version 2.0 for DOS: User's Guide* for a list of valid parameters for the INET subcommand.

`ifconfig: error: inet: 'pktcount' requires an argument in the range (0 - 65535)`

Module: IFCONFIG

Explanation: The parameter for the new pktcount was not valid.

User Response: Reissue the command using a pktcount value between 0 and 65 535 ticks.

`ifconfig: error: inet: 'reasmtmo' requires an argument in the range (0 - 65535)`

Module: IFCONFIG

Explanation: The parameter for the new reasmtmo was not valid.

User Response: Reissue the command using a reasmtmo value between 0 and 65 535 ticks.

`ifconfig: error: inet: 'reflushtmo' requires an argument in the range (0 - 65535)`

Module: IFCONFIG

Explanation: The parameter for the new reflushtmo was not valid.

User Response: Reissue the command using a reflushtmo value between 0 and 65 535 ticks.

`ifconfig: error: inet: 'route add default' requires a router address`

Module: IFCONFIG

Explanation: The IFCONFIG INET IP ROUTE ADD DEFAULT command requires a valid router address.

User Response: Reissue the command using a valid router address.

`ifconfig: error: inet: 'route add host' requires` *`destination router`*

Module: IFCONFIG

Explanation: The IFCONFIG INET IP ROUTE ADD HOST command requires a valid destination address and a valid router address.

User Response: Reissue the command using valid destination and router addresses.

`ifconfig: error: inet: 'route add net' requires` *`destination router mask`*

Module: IFCONFIG

Explanation: The command, IFCONFIG INET IP ROUTE ADD NET, requires a valid destination address, a valid router address, and a valid network mask.

User Response: Reissue the command, using valid destination, router, and network mask addresses.

`ifconfig: error: inet: 'route delete default' requires a router address`

Module: IFCONFIG

Explanation: The IFCONFIG INET IP ROUTE DELETE DEFAULT command requires a valid router address.

User Response: Reissue the command using a valid router address.

`ifconfig: error: inet: 'route delete host' requires` *`destination router`*

Module: IFCONFIG

Explanation: The IFCONFIG INET IP ROUTE DELETE HOST command requires a valid destination address and a valid router address.

User Response: Reissue the command using valid destination and router addresses.

`ifconfig: error: inet: 'route delete net' requires` *`destination router mask`*

Module: IFCONFIG

Explanation: The IFCONFIG INET IP ROUTE DELETE NET command requires a valid destination address, a valid router address, and a valid network mask.

User Response: Reissue the command using valid destination, router, and network mask addresses.

`ifconfig: error: inet: route: '`*`parameter`*`' is an ill formed destination address`

Module: IFCONFIG

Explanation: The *parameter* for the router address is not a valid internet address.

User Response: Reissue the command using a valid internet address for the router parameter.

`ifconfig: error: inet: route: '`*`parameter`*`' is an ill formed network mask`

Module: IFCONFIG

Explanation: The *parameter* for the network mask is not a valid internet address.

User Response: Reissue the command using a valid internet address for the network mask parameter.

`ifconfig: error: inet: route: '`*`parameter`*`' is an ill formed router address`

Module: IFCONFIG

Explanation: The *parameter* for the router address is not a valid internet address.

User Response: Reissue the command using a valid internet address for the router parameter.

ifconfig: error: inet: 'routetmo' requires an argument in the range (0 - 65535)

Module: IFCONFIG

Explanation: The parameter for the new routetmo was not valid.

User Response: Reissue the command using a routetmo value between 0 and 65 535 ticks.

---

ifconfig: error: inet: 'tw_expire' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new tw_expire was not valid.

User Response: Reissue the command using a tw_expire value between 1 and 65 535 ticks.

---

ifconfig: error: inet: 'win_percent' value '*parameter*' out of range (1 - 100)

Module: IFCONFIG

Explanation: The *parameter* for the new win_percent was not valid.

User Response: Reissue the command using a win_percent value between 1 and 100 percent.

---

ifconfig: error: inet: 'zerowintmo' value '*parameter*' out of range (1 - 65535)

Module: IFCONFIG

Explanation: The *parameter* for the new zerowintmo was not valid.

User Response: Reissue the command using a zerowintmo value between 1 and 65 535 ticks.

---

ifconfig: error: Negative arguments are invalid

Module: IFCONFIG

Explanation: IFCONFIG does not accept a parameter that has a value less than zero.

User Response: Reissue the command with valid parameter values.

---

ifconfig: error: No Packet Driver found!

Module: IFCONFIG

Explanation: A packet driver for the network interface has not been loaded in memory.

User Response: Load the appropriate packet driver before running TCPSTART.

---

ifconfig: error: shutdown: '*hwr*' could not be removed, is currently up

Module: IFCONFIG

Explanation: The IFCONFIG *hwr* DOWN command must be executed before the SHUTDOWN command is issued.

User Response: Run the IFCONFIG *hwr* DOWN command, then reissue the SHUTDOWN command.

---

ifconfig: error: shutdown: '*hwr*' is not last in memory

Module: IFCONFIG

Explanation: The specified hardware TSR is not the last TSR loaded in memory. The TSRs must be removed in the reverse order that they were loaded.

User Response: Remove all TSRs loaded after the *hwr* TSR.

---

ifconfig: error: shutdown: 'inet' is not last in memory

Module: IFCONFIG

Explanation: The specified INET TSR is not the last TSR loaded in memory. The TSRs must be removed in the reverse order that they were loaded.

User Response: Remove all TSRs loaded after the specified INET TSR.

---

ifconfig: error: shutdown: 'util' could not be removed, is currently up

Module: IFCONFIG

Explanation: The IFCONFIG UTIL DOWN command must be executed before the SHUTDOWN command is issued.

User Response: Run the IFCONFIG UTIL DOWN command, then reissue the SHUTDOWN command.

---

ifconfig: error: shutdown: 'util' is not last in memory

Module: IFCONFIG

Explanation: The specified UTIL TSR is not the last TSR loaded in memory. The TSRs must be removed in the reverse order that they were loaded.

User Response: Remove all TSRs loaded after the specified UTIL TSR.

`ifconfig: error: 'tsr' is not an installed service`

Module: IFCONFIG

Explanation: Either UTIL, INET, or the appropriate hardware TSR is not installed.

User Response: Be sure the network interface has been configured using the CUSTOM program, and that the TCPSTART command has been issued.

`ifconfig: error: Unknown option -letter`

Module: IFCONFIG

Explanation: You have used an invalid option with the IFCONFIG command.

User Response: See *IBM TCP/IP Version 2.0 for DOS: User's Guide* for a list of valid options for the IFCONFIG command.

`ifconfig: error: upping hardware tsr: Interface is now up`

Module: IFCONFIG

Explanation: The IFCONFIG *hwr* UP command was issued when the hardware TSR was already loaded in memory.

User Response: Entering the IFCONFIG *hwr* UP command when the hardware TSR is already loaded in memory has no effect.

`ifconfig: error: upping hardware TSR: Unrecoverable hardware error`

Module: IFCONFIG

Explanation: The network interface has not been properly configured.

User Response: Use the CUSTOM program to double check the network interface configuration. Check with your network administrator to be sure that the interface configuration is correct.

`ifconfig: error: upping utility tsr: Interface is now up`

Module: IFCONFIG

Explanation: The IFCONFIG UTIL UP command was issued when UTIL was already loaded in memory.

User Response: Entering the IFCONFIG UTIL UP command when UTIL is already loaded in memory has no effect.

`ifconfig: error: util: 'flags chkindos' requires you to specify ON or OFF`

Module: IFCONFIG

Explanation: The only acceptable parameter for IFCONFIG UTIL FLAGS CHKINDOS is on or off.

User Response: Reissue the command with the on or off parameter following CHKINDOS.

`ifconfig: error: util: 'parameter' is an unknown command for this service`

Module: IFCONFIG

Explanation: The *parameter* is not valid with the UTIL subcommand.

User Response: See *IBM TCP/IP Version 2.0 for DOS: User's Guide* for a list of valid parameters for the UTIL subcommand.

`ifconfig: error: util: 'parameter' is an unknown flag`

Module: IFCONFIG

Explanation: The only valid parameter following IFCONFIG UTIL FLAGS is CHKINDOS.

User Response: Reissue the command with CHKINDOS following FLAGS.

`ifconfig: error: util: pool: value out of range (10240 - 61440)`

Module: IFCONFIG

Explanation: The value given for the new pool size was not between 10 240 and 65 535 bytes.

User Response: Reissue the command with a valid pool size.

`ifconfig: error: util: quantum: value out of range (0 - 18)`

Module: IFCONFIG

Explanation: The value given for the new quantum was not between 0 and 18.

User Response: Reissue the command with a valid quantum.

Name Server Messages

The following error messages can appear when using a name server.

`Format error in name resolution query`

Explanation: The name resolution request had an invalid format. If this error is displayed contact TCP/IP for DOS support.

`Name resolution query rejected`

Explanation: Due to administrative policy the query was rejected.

`Name resolution server did not reply`

Explanation: The name server did not respond within the time-out value.

`Name resolution server does not implement request`

Explanation: The name server is not supporting name service for internet addresses. Contact your network administrator.

`Name resolution server error`

Explanation: The name server detected an error in its processing. If this error is displayed contact your network administrator.

`Name resolver unable to allocate resource`

Explanation: System resources to attempt name resolution were not currently available. The error is usually due to an out of memory condition.

`No error condition exists`

Explanation: Value indicating no error. This message should normally never be displayed. If this error is displayed, contact TCP/IP for DOS support.

`Unknown host name specified`

Explanation: The server could not determine an address for the specified name.

NFS Errors

The following error messages can appear in the IBMNFS module.

IBMNFS: `a dosnfs tsr has not been activated`

Module: NFS

Explanation: An attempt to execute a command failed, because the DOSNFS TSR has not been activated.

User Response: This error message should never be displayed. Please contact TCP/IP for DOS support.

IBMNFS: `a dosnfs tsr has not been loaded`

Module: NFS

Explanation: An attempt to execute NFS has failed, because DOSNFS has not been previously loaded into memory.

User Response: Load DOSNFS and then reissue the command.

IBMNFS: `a utility tsr has not been loaded`

Module: NFS

Explanation: Network services are not available, because no utility TSR has been loaded.

User Response: Make sure that a utility, internet, hardware, and DOSNFS TSR are loaded into memory and reissue the command.

IBMNFS: `close failed on file #`*value*`:` *message*

Module: NFS

Explanation: An attempt to close the indicated file while removing the DOSNFS TSR failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: No response is necessary.

IBMNFS: `dosnfs:` *message*

Module: NFS

Explanation: An attempt by NFS to invoke the DOSNFS TSR failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: auth: missing hostname`

Module: NFS

Explanation: An attempt to execute the AUTH command failed, because no host name was specified.

User Response: Reissue the command and include the host name.

IBMNFS: `error: auth: unknown host:` *hostname*

Module: NFS

Explanation: An attempt to execute the AUTH command failed, because *hostname* cannot be resolved into an internet address.

User Response: Reissue the command and include a resolvable host name.

IBMNFS: `error: Authentication is required`

Module: NFS

Explanation: An attempt to mount a remote file system or print device failed, because authentication is required.

User Response: Use the `auth` command to obtain authentication and reissue the command.

IBMNFS: `error: bad or missing hostname`

Module: NFS

Explanation: An attempt to mount an NFS drive failed, because the host name did not follow the `path@` specification.

User Response: Reissue the command and include the host name after the `path@` specification.

IBMNFS: `error: bad or missing path`

Module: NFS

Explanation: An attempt to mount an NFS drive failed, because the path did not follow the `hostname:` specification.

User Response: Reissue the command and include the host name after the `hostname:` specification.

IBMNFS: `error: bad or missing server name`

Module: NFS

Explanation: An attempt to attach the specified print device failed, because the specified server name was either bad or missing.

User Response: Reissue the command specifying a valid server.

IBMNFS: `error: bad or missing service`

Module: NFS

Explanation: An attempt to attach the specified print device failed, because the specified service was either bad or missing.

User Response: Reissue the command specifying a valid service.

IBMNFS: `error: cannot open file:` *file*

Module: NFS

Explanation: An attempt to open *file* failed.

User Response: Reissue the command specifying the name of the file containing NFS commands.

IBMNFS: `error: disk` *drive* `detach failed:` *message*

Module: NFS

Explanation: An attempt to detach disk *drive* failed for the specified reason. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: dosnfsd ping failed:` *message*

Module: NFS

Explanation: An attempt to ping the DOSNFSD daemon failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: drop failed:` *message*

Module: NFS

Explanation: An attempt to unmount all the NFS drives from the specified host failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: error while attaching:` *message*

Module: NFS

Explanation: An attempt to attach a print device failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: error while detaching:` *message*

Module: NFS

Explanation: An attempt to detach a print device failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: error while flushing:` *message*

Module: NFS

Explanation: An attempt to close a print device failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: export list failed:` *message*

Module: NFS

Explanation: An attempt to list all the mountable file systems or all the currently mounted file systems for the specified host failed for the specified reason. See the explanation for

message contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: error: illegal server or path name: *server path name*

Module: NFS

Explanation: An attempt to mount an NFS drive failed, because the *server path name* was missing or was incorrect.

User Response: Reissue the command and include the NFS server and path in either the form mount attach *server :path* or in the form mount attach *path@server*.

IBMNFS: error: illegal server or service name: *server | service*

Module: NFS

Explanation: An attempt to attach the specified print device failed, because no server or service was specified.

User Response: Reissue the command specifying both a server and service.

IBMNFS: error: invalid error setting: *setting*

Module: NFS

Explanation: An attempt to assign the error resolution failed, because *setting* was invalid. Valid settings are hard and soft.

User Response: Reissue the command specifying a valid setting.

IBMNFS: error: invalid or unknown host: *hostname*

Module: NFS

Explanation: An attempt to mount or unmount an NFS drive, list the mountable file systems, or attach a print device failed, because *hostname* could not be resolved into an internet address.

User Response: Reissue the command and include a resolvable host name.

IBMNFS: error: invalid scroll setting: *setting*

Module: NFS

Explanation: An attempt to assign the scroll setting failed, because *setting* was invalid. Valid settings are on and off.

User Response: Reissue the command specifying a valid setting.

IBMNFS: error: maximum file nesting level exceeded

Module: NFS

Explanation: An attempt to execute a script file failed the maximum number of nested script files was exceeded. The maximum is 100.

User Response: Reissue the command and limit the number of nested script files to under 100.

IBMNFS: error: Maximum number of arguments exceeded

Module: NFS

Explanation: More than 100 execute files were nested.

User Response: Reissue the command nesting no more than a total of 100 executable files.

IBMNFS: error: messages: invalid message value: *value*

Module: NFS

Explanation: An attempt to set the message level to *value* failed, because *value* was invalid.

User Response: Consult the for the available values and reissue specifying a valid value.

IBMNFS: error: missing device name

Module: NFS

Explanation: An attempt to attach, detach, or close a print device failed, because no print device was specified.

User Response: Reissue the command specifying the print device you wish.

IBMNFS: error: missing filename

Module: NFS

Explanation: An attempt to use the execute command failed, because no command file was specified.

User Response: Reissue the command specifying a command file.

IBMNFS: error: missing hostname

Module: NFS

Explanation: An attempt to list mountable file systems, unmount NFS drives, or perform an NFS ping failed, because no host was specified.

User Response: Reissue the command specifying the host for which you wish to list mountable directories or to drop all associated drives.

IBMNFS: error: missing or invalid drive letter

Module: NFS

Explanation: An attempt to mount an NFS drive failed, because the specified drive letter was missing or invalid.

User Response: Reissue the command and include a valid drive letter followed immediately by a colon.

IBMNFS: `error: missing quote`

Module: NFS

Explanation: An attempt to execute the specified command failed, because the quotation marks (" ") were not balanced.

User Response: Reissue the command making sure that all quotes are balanced.

IBMNFS: `error: missing umask`

Module: NFS

Explanation: An attempt to specify the umask failed, because no value was specified.

User Response: Reissue the command specifying a umask.

IBMNFS: `error: mount failed:` *message*

Module: NFS

Explanation: An attempt to mount a NFS drive failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: mount ping failed:` *message*

Module: NFS

Explanation: An attempt to ping the mounted daemon failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: nfs ping failed:` *message*

Explanation: An attempt to ping the NFSD daemon failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error: rpc connection failed:` *message*

Module: NFS

Explanation: The rpc connection failed or was broken for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: `error:` *subcommand*`: illegal bufsiz:` *value*

Module: NFS

Explanation: An attempt to assign the bufsiz on the indicated *subcommand* failed, because the bufsiz was out of the allowable range (512 — 8192 bytes).

User Response: Reissue the command specifying a bufsize within the range.

IBMNFS: `error:` *subcommand*`: illegal cache mark:` *value*

Module: NFS

Explanation: An attempt to assign the cache size on the *subcommand* failed, because *value* was out of the allowable range (0 — 8192 bytes).

User Response: Reissue the command specifying a cache size within the range.

IBMNFS: `error:` *subcommand*`: illegal failure timeout:` *value*

Module: NFS

Explanation: An attempt to assign the failure time-out value in *subcommand* failed, because it was an illegal value.

User Response: Reissue the command specifying a legal failure time-out value.

IBMNFS: `error:` *subcommand*`: illegal retry timeout:` *value*

Module: NFS

Explanation: An attempt to assign the retry time-out value in *subcommand* failed, because it was an illegal value.

User Response: Reissue the command specifying a positive integer as a retry timeout.

IBMNFS: error: *subcommand*: missing bufsiz

Module: NFS

Explanation: An attempt to assign the bufsiz for *subcommand* failed, because none was specified.

User Response: Reissue the command specifying a bufsiz.

IBMNFS: error: *subcommand* : missing cache mark *text*

Module: NFS

Explanation: An attempt to assign the cache size on the indicated command failed, because none was specified.

User Response: Reissue the command specifying a cache size.

IBMNFS: error: *subcommand* : missing failure timeout

Module: NFS

Explanation: An attempt to assign the failure time-out on the indicated command failed, because none was specified.

User Response: Reissue the command specifying a failure timeout.

IBMNFS: error: *subcommand*: missing queuename

Module: NFS

Explanation: An attempt to assign a queue name on the indicated *subcommand* failed, because none was specified.

User Response: Reissue the command specifying a queue name.

IBMNFS: error: *subcommand*: missing retry timeout

Module: NFS

Explanation: An attempt to assign a retry time-out on the indicated *subcommand* failed, because none was specified.

User Response: Reissue the command specifying a retry timeout.

IBMNFS: error: *subcommand*: missing user name

Module: NFS

Explanation: An attempt to assign a user name on the indicated *subcommand* failed as none was specified.

User Response: Reissue the command specifying the user name.

IBMNFS: error: *subcommand*: missing volume id

Module: NFS

Explanation: An attempt to assign a volume id failed as none was specified.

User Response: Reissue the command specifying the volume ID.

IBMNFS: error: *subcommand*: option expected

Module: NFS

Explanation: An attempt to execute *subcommand* failed, because it requires an option. To find the available options, see *IBM TCP/IP Version 2.0 for DOS: User's Guide*.

User Response: Reissue the command supplying correct options.

IBMNFS: error: *subcommand*: unknown option - *option*

Module: NFS

Explanation: An attempt to execute *subcommand* failed, because *option* does not exist for this command. To find the available options, see *IBM TCP/IP Version 2.0 for DOS: User's Guide*.

User Response: Reissue the command supplying correct options.

IBMNFS: error: shutdown: *name*: could not be downed: *message*

Module: NFS

Explanation: An attempt to down the *name* TSR failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: error: shutdown: *name*: could not be removed: *message*

Module: NFS

Explanation: An attempt to remove the *name* TSR failed for the specified reason. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: error: TCP/IP networking has not been activated

Module: NFS

Explanation: An attempt to execute a command failed, because networking services have not been activated. This occurs when the UTIL TSR is down even when the up command has been given with kstart.

User Response: Activate network services and reissue the command.

IBMNFS: error: Unable to load secondary command processor: *message*

Module: NFS

Explanation: An attempt to load a secondary command processor failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: error: unknown command: *command*

Module: NFS

Explanation: An attempt to execute *command* failed, because it was not recognized.

User Response: See *IBM TCP/IP Version 2.0 for DOS: User's Guide* for the available commands and reissue specifying a valid command.

IBMNFS: error: unknown device: *device*

Module: NFS

Explanation: An attempt to attach, detach, or close the print device *device* failed, because it does not exist. Legal print devices are PRN, LPT1, LPT2, and LPT3.

User Response: Reissue the command specifying a legal print device.

IBMNFS: error: unknown host: *hostname*

Module: NFS

Explanation: An attempt to perform an NFS ping failed, because *hostname* could not be resolved into an internet address.

User Response: Reissue the command and include a resolvable host name.

IBMNFS: error: unmount failed: *message*

Module: NFS

Explanation: An attempt to unmount an NFS drive failed for the reason specified in *message*. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: error: verification failed: *message*

Module: NFS

Explanation: An attempt to execute the auth command failed for the specified reason. See the explanation for *message* contained in the appropriate IBM DOS technical reference book for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: initialization: *message*

Module: NFS

Explanation: An attempt to initiate network initialization failed for the reason specified in *message*. See the explanation for *message* contained in "Generic Text Messages" on page 291 for more information.

User Response: The action you take depends on the *message* displayed. If the explanation for *message* is insufficient to resolve the problem, contact your network administrator.

IBMNFS: kstart has not been run

Module: NFS

Explanation: An attempt to execute NFS has failed, because background network services were not operational.

User Response: Execute KSTART with the up option and reissue NFS

IBMNFS: shutdown: a DOSNFS tsr has not been loaded

Module: NFS

Explanation: An attempt to remove the DOSNFS TSR failed as none has been loaded.

User Response: No response is necessary.

IBMNFS: shutdown: DOSNFS tsr is not last in memory

Module: NFS

Explanation: An attempt to remove the DOSNFS TSR failed, because it is not last in memory. This message can also occur if KSTART had been executed after DOSNFS was loaded.

User Response: Remove any resident programs from memory that had been loaded after DOSNFS and reissue the command. If KSTART had been executed after DOSNFS was loaded, execute kstart down and then reissue the command.

IBMNFS: TSR could not find open file *#value*

Module: NFS

Explanation: While attempting to remove the DOSNFS TSR the open file *#value* could not be found to close.

User Response: No response is necessary.

TSR Errors

The following error messages can appear in the TSR modules.

---

module: Error encountered reading NETCUST

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: The *module* could not properly read the NETCUST structure.

User Response: Make sure that the NETDEV.SYS is installed in your CONFIG.SYS.

---

module: Install aborted, will leave too little memory (<64k) for DOS

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: The install aborted, because loading the TSR would cause less than the minimum memory requirement (64K) to be left for DOS.

User Response: Do not load this TSR or unload some RAM-resident software before loading this new TSR.

---

module: Invalid characters in specified name, only [A-Z, 0-9, _] allowed
Usage: *module* [-?] [name]

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: A name was specified for a *module* with invalid characters.

User Response: Only use the characters [A-Z, 0-9, underscore (_)] if a *module* name.

---

module: Invalid option specified
Usage: *module* [-?] [name]

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: The specified option is not valid with the *module* used.

User Response: Specify the proper options for the *module* you are using or use a module which understands the option you want to use.

---

module: More than one adapter found.

Module: 3C523 UB2

Explanation: More than one network interface adapter was found.

User Response: Specify which adapter to use, or remove one from your system. Use your reference diskette to view your configuration and determine which adapter to specify.

---

module: Multiple names not supported
Usage: *module* [-?] [name]

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: Two or more names were specified to install a *module*.

User Response: Use only one name when installing a *module*.

---

module: No adapter found in given channel.

Module: 3C523 UB2

Explanation: No *module* adapter was found in the specified channel.

User Response: Use your reference diskette to view your configuration and reissue the command specifying the correct channel.

---

module: No room: too many servants

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: An attempt was made to load too many services.

User Response: Do not load the new TSR or unload one of the old ones before loading the new one.

---

module: Specified name already in use, select a different name
Usage: *module* [-?] [name]

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: A *module* is already in use with the name you specified, or with the default name if you did not specify one.

User Response: Specify a different name for the *module*.

---

module: Specified name too long, eight character maximum
Usage: *module* [-?] [name]

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: A name was specified for the *module* longer than eight characters.

User Response: Reissue the command using a name of eight characters or less for the *module*.

module: This adapter is not supported on a micro channel machine
Usage: *module* [-?] [name]

Module: 3C523 UB2

Explanation: The *module* adapter specified is not supported on a Micro Channel* machine and your system is a Micro Channel machine.

User Response: Use an adapter that is supported on a Micro Channel machine.

---

module: This adapter is only supported on a micro channel machine
Usage: *module* [-?] [name]

Module: 3C523 UB2

Explanation: The *module* adapter specified is only supported on a Micro Channel machine and your system is not a Micro Channel machine.

User Response: Use an adapter that is supported on other than Micro Channel machines.

---

module: *type-of-service*: TSR service already exists

Module: ASI 3C501 3C503 3C523 INET DOSNFS UB UB2 UTIL

Explanation: Only one Utility TSR, one NFS TSR, and one Internet Protocol on DOS TSR service can be loaded at a time. You have tried to load two of the specified *type-of-service*.

User Response: Do not load the new TSR of that type or unload the old TSR of that type before loading the new one.

UB2: INIT - Time Out.

Explanation: The Ungermann-Bass** NICps/2** adapter could not be initialized within the prescribed amount of time. This usually means that there is a program conflict.

User Response: Reboot your system without loading any programs that conflict with the adapter's use.

UB2: No adapter found.

Explanation: No Ungermann-Bass NICps/2 adapter was found in your system.

User Response: Use your reference diskette to view your configuration and determine what type of adapter (if any) is in your system.

3C523: No enabled adapter found.

Explanation: There was no enabled 3C523 adapter found within your system.

User Response: Use your reference diskette to view your configuration and determine which adapter to specify.

3C523: specified board is disabled

Explanation: The 3C523 adapter specified was disabled.

User Response: Either enable the desired 3C523 adapter, or use a different 3C523 adapter.

3C523: warning: found disabled board

Explanation: A disabled 3C523 adapter was found within your system.

User Response: Either enable the 3C523 adapter, or specify another adapter that is enabled.

Appendix L. Related Protocol Specifications

IBM is committed to industry standards. The internet protocol suite is still evolving through Requests for Comments (RFC). New protocols are being designed and implemented by researchers, and are brought to the attention of the Internet community in the form of RFCs. Some of these are so useful that they become a recommended protocol. That is, all future implementations for TCP/IP are recommended to implement this particular function or protocol. These become the *de facto* standards, on which the TCP/IP protocol suite is built.

Many features of TCP/IP for DOS are based on the following RFCs:

- *User Datagram Protocol*, RFC 768, J.B. Postel
- *Trivial File Transfer Protocol*, (Revision 2) RFC 783, K.R. Sollins
- *Internet Protocol*, RFC 791, J.B. Postel
- *Internet Control Message Protocol*, RFC 792, J.B. Postel
- *Transmission Control Protocol*, RFC 793, J.B. Postel
- *Simple Mail Transfer Protocol*, RFC 821, J.B. Postel
- *Standard for the Format of ARPA Internet Text Messages*, RFC 822, D. Crocker
- *DARPA Internet Gateway*, RFC 823, R.M. Hinden, A. Sheltzer
- *Ethernet Address Resolution Protocol: or Converting Network Protocol Addresses to 48.Bit Ethernet Address for Transmission on Ethernet Hardware*, RFC 826, D.C. Plummer
- *Telnet Protocol Specification*, RFC 854, J.B. Postel, J.K. Reynolds
- *Telnet Binary Transmission*, RFC 856, J.B. Postel, J.K. Reynolds
- *Telnet Echo Option*, RFC 857, J.B. Postel, J.K. Reynolds
- *Quote of the Day Protocol*, RFC 865, J.B. Postel
- *Time Protocol*, RFC 868, J.B. Postel, K. Harrenstien
- *Standard for the Transmission of IP Datagrams over Public Data Networks*, RFC 877, J.T. Korb
- *Telnet End of Record Option*, RFC 885, J.B. Postel
- *Broadcasting Internet Datagrams*, RFC 919, J.C. Mogul
- *Broadcasting Internet Datagrams in the Presence of Subnets*, RFC 922, J.C. Mogul
- *Post Office Protocol—Version 2*, RFC 937, M. Butler, J.B. Postel, D. Chase, J. Goldberger, J.K. Reynolds
- *Internet Standard Subnetting Procedure*, RFC 950, J.C. Mogul, J.B. Postel
- *DoD Internet Host Table Specification*, RFC 952, K. Harrenstien, M.K. Stahl, E.J. Feinler
- *NICNAME/WHOIS*, RFC 954, K. Harrenstien, M.K. Stahl, E.J. Feinler
- *File Transfer Protocol*, RFC 959, J.B. Postel, J.K. Reynolds
- *Mail Routing And The Domain Name System*, RFC 974, C. Partridge
- *XDR: External Data Representation Standard*, RFC 1014, Sun Microsystems Incorporated

- *Domain Names—Concepts and Facilities*, RFC 1034, P.V. Mockapetris
- *Domain Names—Implementation and Specification*, RFC 1035, P.V. Mockapetris
- *RPC: Remote Procedure Call Protocol Version 2 Specification*, RFC 1057, Sun Microsystems Incorporated
- *Routing Information Protocol*, RFC 1058, C.L. Hedrick
- *Assigned Numbers*, RFC 1060, J.K. Reynolds, J.B. Postel
- *Telnet Terminal-Type Option*, RFC 1091, J. VanBokkelen
- *NFS: Network File System Protocol Specification*, RFC 1094, Sun Microsystems Incorporated
- *Hitchhikers Guide to the Internet*, RFC 1118, E. Krol
- *Requirements for Internet Hosts—Communication Layers*, RFC 1122, R.T. Braden, editor
- *Requirements for Internet Hosts—Application and Support*, RFC 1123, R.T. Braden, editor
- *Line Printer Daemon Protocol*, RFC 1179, The Wollongong Group, L. McLaughlin III, editor
- *TCP/IP Tutorial*, RFC 1180, T.J. Socolofsky, C.J. Kale
- *Finger User Information Protocol* RFC 1196, D.P. Zimmerman
- *IAB Official Protocol Standards*, RFC 1200, Defense Advance Research Projects Agency, Internet Activities Board
- *FYI on Questions and Answers: Answers to Commonly Asked "New Internet User" Questions*, RFC 1206, G.S. Malkin, A.N. Marine
- *FYI on Questions and Answers: Answers to Commonly Asked "Experienced Internet User" Questions*, RFC 1207, G.S. Malkin, A.N. Marine, J.K. Reynolds
- *Glossary of Networking Terms*, RFC 1208, O.J. Jacobsen, D.C. Lynch.

These documents can be obtained from:

SRI International
Network Information Systems Center
Room EJ291
333 Ravenswood Avenue
Menlo Park, CA. 94025

Many RFCs are available online. Hard copies of all RFCs are available from the NIC, either individually or on a subscription basis. Online copies are available using FTP from the NIC at `nic.ddn.mil`. Use FTP to download the files, using the following format:

```
RFC:RFC-INDEX.TXT
RFC:RFCnnnn.TXT
RFC:RFCnnnn.PS
```

Where:

nnnn	Is the RFC number.
TXT	Is the text format.
PS	Is the PostScript** format.

You can also request RFCs through electronic mail, from the automated NIC mail server, by sending a message to service@nic.ddn.mil with a subject line of RFC nnnn for text versions or a subject line of RFC nnnn.PS for PostScript versions. To request a copy of the RFC index, send a message with a subject line of RFC INDEX.

For more information, contact nic@nic.ddn.mil.

Glossary, Bibliography, and Index

Glossary

This glossary describes the most common terms associated with TCP/IP communication in an internet environment, as used in this book.

If you do not find the term you are looking for, see *IBM Dictionary of Computing*, SC20-1699.

This glossary includes some terms from *IBM Dictionary of Computing*.

For abbreviations, the definition usually consists only of the words represented by the letters; for complete definitions, see the entries for the words.

A

ABEND. The abnormal termination of a program or task.

accelerator key. A key or combination of keys that invokes an application-defined function. Also known as a function key.

action bar. The highlighted area at the top of a panel that contains the choices currently available in the application program that a user is running.

active open. The state of a connection that is actively seeking a service. Contrast with *passive open*.

adapter. (1) A piece of hardware that connects a computer and an external device. (2) An auxiliary device or unit used to extend the operation of another system.

address. The unique code assigned to each device or workstation connected to a network. A standard internet address is a 32-bit address field. This field can be broken into two parts. The first part contains the network address; the second part contains the host number.

Address Resolution Protocol (ARP). A protocol used to dynamically bind an internet address to a hardware address. ARP is implemented on a single physical network and is limited to networks that support broadcast addressing.

American National Standard Code for Information Interchange (ASCII). (1) The standard code, using a coded character set consisting of 7-bit coded characters (8 bits including parity check), used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters. (2) The default file transfer type for FTP, used to transfer files that contain ASCII text characters.

American National Standards Institute (ANSI). An organization consisting of producers, consumers, and general interest groups, that establishes the procedures by which accredited organizations create and maintain voluntary industry standards in the United States.

ANSI. American National Standards Institute.

application. The use to which an information processing system is put, for example, a payroll application, an airline reservation application, a network application.

argument. A parameter passed between a calling program and a called program.

API. application program interface.

application program interface. The formally-defined program language interface, which is between an IBM system control program or a licensed program and the user of the program.

ARP. Address Resolution Protocol.

ASCII. American National Standard Code for Information Interchange.

asynchronous. Without regular time relationship; unexpected or unpredictable with respect to the execution of program instruction. See *synchronous*.

attribute. A characteristic or property. For example, the color of a line, or the length of a data field.

authorization. The right granted to a user to communicate with, or to make use of, a computer system or service.

AUTOEXEC.BAT. A batch file that resides in the root directory of drive C. AUTOEXEC.BAT contains commands that DOS executes every time you boot a PC.

B

backbone. (1) In a local area network multiple-bridge ring configuration, a high-speed link to which rings are connected by means of bridges. A backbone can be configured as a bus or as a ring. (2) In a wide area network, a high-speed link to which nodes or data switching exchanges (DSES) are connected.

background task. A task with which the user is not currently interacting, but continues to run.

Basic Input/Output System (BIOS). A set of routines that permanently resides in read-only memory (ROM)

in a PC. The BIOS performs the most basic tasks, such as sending a character to the printer, booting the computer, and reading the keyboard.

batch. (1) An accumulation of data to be processed. (2) A group of records or data processing jobs brought together for processing or transmission. (3) Pertaining to activity involving little or no user action. See *interactive*.

BIOS. Basic Input/Output System.

block. A string of data elements recorded, processed, or transmitted as a unit. The elements can be characters, words, or physical records.

bridge. A router that connects two or more networks and forwards packets among them. The operations carried out by a bridge are done at the physical layer and are transparent to TCP/IP and TCP/IP routing.

broadcast. The simultaneous transmission of data packets to all nodes on a network or subnetwork.

broadcast address. An address that is common to all nodes on a network.

bus topology. A network configuration in which only one path is maintained between stations. Any data transmitted by a station is concurrently available to all other stations on the link.

button. (1) A mechanism on a pointing device, such as a mouse, used to request or initiate an action. (2) A rounded-corner rectangle with text inside, used in graphics applications for actions that occur when the pushbutton is selected.

C

case-sensitive. A condition in which entries for an entry field must conform to a specific lower -, upper -, or mixed-case format in order to be valid.

checksum. The sum of a group of data associated with the group and used for checking purposes.

Class A network. An internet network in which the high-order bit of the address is 0. The host number occupies the three low-order octets.

Class B network. An internet network in which the high-order bit of the address is 1 and the next high-order bit is 0. The host number occupies the two low-order octets.

Class C network. An internet network in which the two high-order bits of the address are 1 and the next high-order bit is 0. The host number occupies the low-order octet.

click. To press and release the select button on a mouse.

client. A function that requests services from a server, and makes them available to the user.

client-server relationship. A device that provides resources or services to other devices on a network is a *server*. A device that employs the resources provided by a server is a *client*.

clipboard. A temporary storage area used for copying and storing data.

CMS. Conversational Monitor System.

command. The name and any parameters associated with an action that can be performed by a program. The command is entered by the user; the computer performs the action requested by the command name.

command prompt. A displayed symbol, such as [C:\] that requests input from a user.

compile. (1) To translate a program written in a high-level language into a machine language program. (2) The computer actions required to transform a source file into an executable object file.

compiler. A program that translates a source program into an executable program (an object program).

CONFIG.SYS. A file that contains the configuration options for a DOS personal computer.

configuration file. For the base operating system, the CONFIG.SYS file that describes the devices, system parameters, and resource options of a personal computer.

connection. (1) An association established between functional units for conveying information. (2) The path between two protocol modules that provides reliable stream delivery service. In an internet, a connection extends from a TCP module on one machine to a TCP module on the other.

conversational monitor system (CMS). A virtual machine operating system that provides general interactive time sharing, problem solving, and program development capabilities, and operates only under control of the VM/370 VM control program.

D

daemon. A background process usually started at system initialization that runs continuously and performs a function required by other processes.

datagram. The basic unit of information that is passed across the internet, it consists of one or more data packets.

data set. The major unit of data storage and retrieval in MVS, consisting of a collection of data in one of several prescribed arrangements and described by control information to which the system has access. Synonymous with *file* in VM and DOS.

default. A value, attribute or option that is assumed when none is explicitly specified.

destination node. The node to which a request or data is sent.

dialog box. A movable window, fixed in size, which provides information that is required by an application to continue your request.

directory. A named grouping of files in a file system.

diskette. (1) A small magnetic disk enclosed in a jacket. (2) A thin, flexible magnetic disk and a semi-rigid or hard plastic protective jacket, in which the disk is permanently enclosed.

DNS. Domain Name System.

domain. In an internet, a part of the naming hierarchy. Syntactically, a domain name consists of a sequence of names (labels) separated by periods (dots).

Domain Name System. A system in which a *resolver* queries name servers for resource records about a host.

domain naming. A hierarchical system for naming network resources.

DOS. Disk Operating System.

dotted-decimal notation. The syntactic representation for a 32-bit integer that consists of four 8-bit numbers, written in base 10 and separated by periods (dots). Many internet application programs accept dotted decimal notations in place of destination machine names.

dragging. Moving an object on the display screen as if it were attached to the pointer, or mouse; performed by holding the select button and moving the pointer.

drive. The device used to read and write data on disks or diskettes.

E

EBCDIC. Extended binary-coded decimal interchange code.

encapsulation. A process used by layered protocols in which a lower level protocol accepts a message from a higher level protocol and places it in the data portion of the low level frame.

enhanced keyboard. A 101-key keyboard, such as the keyboards that come with PS/2 computers.

entry field. A panel element, usually highlighted in some manner and usually with its boundaries indicated, where users type in information.

Ethernet. The name given to a local area packet-switched network technology invented in the early 1970s by Xerox Incorporated. Ethernet uses a Carrier Sense Multiple Access/Collision Detection (CSMA/CD) mechanism to send packets.

extended binary-coded decimal interchange code (EBCDIC). A coded character set consisting of 8-bit coded characters.

eXternal Data Representation (XDR). A standard developed by SUN Microsystems Incorporated for representing data in machine-independent format.

F

file. In DOS, OS/2, and VM, a named set of records stored or processed as a unit. Synonymous with *data set* in MVS.

File Transfer Protocol (FTP). A TCP/IP protocol used for transferring files to and from foreign hosts. FTP also provides the capability to access directories. Password protection is provided as part of the protocol.

fixed disk. A rigid magnetic disk, such as the internal disks used in the system units of IBM personal computers and in external hard disk drives.

foreground task. The task with which the user is interacting.

foreign host. Any host on the network including the local host.

foreign network. In an internet, any other network interconnected to the local network by one or more intermediate gateways or routers.

foreign node. See *foreign host*.

FTP. File Transfer Protocol.

G

gateway. (1) A functional unit that interconnects a local data network with another network having different protocols. (2) A host that connects a TCP/IP network to a non-TCP/IP network at the application layer. See also *router*.

H

handle. A temporary data representation that identifies a file.

header file. A file that contains constant declarations, type declarations, and variable declarations and assignments. Header files are supplied with all programming interfaces.

hop count. The number of hosts through which a packet passes on its way to its destination.

host. A computer connected to a network, which provides an access method to that network. A host provides end-user services.

hot key. A combination of keys that causes a program to perform some action.

I

ICMP. Internet Control Message Protocol.

IEEE. Institute of Electrical and Electronic Engineers.

include file. A file that contains preprocessor text, which is called by a program, using a standard programming call. Synonymous with *header file*.

installation. The process of placing one or more DOS components on a personal computer's fixed disk.

Institute of Electrical and Electronic Engineers (IEEE). An electronics industry organization.

Integrated Services Digital Network (ISDN). A digital end-to-end telecommunication network that supports multiple services including, but not limited to, voice and data.

interactive. Pertaining to a program or a system that alternately accepts input and then responds. An interactive system is conversational, that is, a continuous dialog exists between user and system. See *batch*.

International Organization for Standardization (ISO). An organization of national standards bodies from various countries established to promote development of standards to facilitate international exchange of goods and services, and develop cooperation in intellectual, scientific, technological, and economic activity.

internet or internetwork. A collection of packet switching networks interconnected by gateways, routers, bridges, and hosts to function as a single, coordinated, virtual network.

internet address. The unique 32-bit address identifying each node in an internet. See also *address*.

Internet Control Message Protocol (ICMP). The part of the Internet Protocol layer that handles error messages and control messages.

Internet Protocol (IP). The TCP/IP layer between the higher level host-to-host protocol and the local network protocols. IP uses local area network protocols to carry packets, in the form of datagrams, to the next gateway, router, or destination host.

interoperability. The capability of different hardware and software by different vendors to effectively communicate together.

interrupt number. One of eight lines available on a PC that is used to send a signal from an installed hardware board to the CPU requesting attention. Different hardware boards should use different interrupt numbers.

IP. Internet Protocol.

ISDN. Integrated Services Digital Network.

ISO. International Organization for Standardization.

KB. Kilobyte; 1024 bytes.

L

LAN. Local area network.

Line printer daemon (LPD). The remote printer server that allows other hosts to print on a printer local to your host.

local area network (LAN). A data network located on the user's premises in which serial transmission is used for direct data communication among data stations.

local host. In an internet, the computer to which a user's terminal is directly connected without using the internet.

local network. The portion of a network that is physically connected to the host without intermediate gateways or routers.

Logical ANDing. When the Boolean operator AND is applied to two bits, the result is one when both bits are one; otherwise, the result is zero. When two bytes are ANDed, each pair of bits is handled separately; there is no connection from one bit position to another.

LPD. Line printer daemon.

LPR. A client command that allows the local host to submit a file to be printed on a remote print server.

M

MAC. Media access control.

mapping. The process of relating internet addresses to physical addresses in the network.

MARK. A Windows function that marks a section of text to be copied or cut.

mask. (1) A pattern of characters used to control retention or elimination of portions of another pattern of characters. (2) To use a pattern of characters to control retention or elimination of another pattern of characters. (3) A pattern of characters that controls the keeping, deleting, or testing of portions of another pattern of characters.

MB. Megabyte; 1 048 576 bytes or 1024 KB.

MBUF. Memory buffer.

media access control (MAC). The method used by network adapters to determine which adapter has access to the physical network at a given time.

memory buffer. A location in memory where incoming and outgoing packets are stored.

menu. A type of panel that consists of one or more selection fields.

menu item. A selection item on a pull-down menu.

mouse. A device that is used to move a pointer on the screen and select items.

modem (modulator/demodulator). A device that converts digital data from a computer to an analog signal that can be transmitted on a telecommunication line, and converts the analog signal received to data for the computer.

multitasking. A mode of operation that provides for the concurrent performance execution of two or more tasks.

N

name server. The server that stores resource records about hosts.

NDIS. Network Driver Interface Specification.

network. An arrangement of nodes and connecting branches. Connections are made between data stations.

network adapter. A physical device, and its associated software, that enables a processor or controller to be connected to a network.

network administrator. The person responsible for the installation, management, control, and configuration of a network.

Network Driver Interface Specification (NDIS). An industry-standard specification used by applications as an interface with network adapter device drivers.

Network File System (NFS). The NFS protocol, which was developed by Sun Microsystems Incorporated, allows computers in a network to access each other's file systems. Once accessed, the file system appears to reside on the local host.

NFS. Network File System.

node. (1) In a network, a point at which one or more functional units connect channels or data circuits. (2) In a network topology, the point at an end of a branch.

O

octet. A byte composed of eight binary elements.

open system. A system with specified standards and that therefore can be readily connected to other systems that comply with the same standards.

Open Systems Interconnection (OSI). (1) The interconnection of open systems in accordance with specific ISO standards. (2) The use of standardized procedures to enable the interconnection of data processing systems.

OSI. Open Systems Interconnection.

P

packet. A sequence of binary digits, including data and control signals, that is transmitted and switched as a composite whole.

parameter. A variable that is given a constant value for a specified application.

parse. To analyze the operands entered with a command.

passive open. The state of a connection that is prepared to provide a service on demand. Contrast with *active open*.

path. The course or route of drives and subdirectories leading from the root directory and drive of an operating system to where files or data information are stored.

PC. Personal computer.

PC Network. A low-cost broadband network that allows attached IBM personal computers, such as IBM

5150 Personal Computers, IBM Computer ATs, IBM PC/XTs, and IBM Portable Personal Computers to communicate and to share resources.

PDU. Protocol Data Units.

peer-to-peer. In network architecture, any functional unit that resides in the same layer as another entity.

PING. The command that sends an ICMP Echo Request packet to a gateway, router, or host with the expectation of receiving a reply.

poolsize. The amount of memory that TCP/IP for DOS sets aside for memory buffers.

port. (1) An endpoint for communication between devices, generally referring to a logical connection. (2) A 16-bit number identifying a particular Transmission Control Protocol or User Datagram Protocol resource within a given TCP/IP node.

PORTMAP. Synonymous with *Portmapper*.

Portmapper. A program that maps client programs to the port numbers of server programs. Portmapper is used with Remote Procedure Call (RPC) programs.

process. (1) A unique, finite course of events defined by its purpose or by its effect, achieved under defined conditions. (2) Any operation or combination of operations on data. (3) A function being performed or waiting to be performed. (4) A program in operation; for example, a daemon is a system process that is always running on the system.

protocol. A set of semantic and syntactic rules that determines the behavior of functional units in achieving communication. Protocols can determine low-level details of machine-to-machine interfaces, such as the order in which bits from a byte are sent; they can also determine high-level exchanges between application programs, such as file transfer.

protocol suite. A set of protocols that cooperate to handle the transmission tasks for a data communication system.

pull-down. An extension of the action bar that displays a list of choices that are available for a selected action bar choice.

R

radix. (1) The positive integer by which the weight of the digit place is multiplied to obtain the weight of the digit place with the next higher weight; for example, in the decimal numeration system the radix of each digit place is 10, in a biquinary code the radix of each fives place is 2. (2) Deprecated term for base.

RAM. Random Access Memory.

Random Access Memory (RAM). A memory device into which data is entered and from which data is retrieved in a nonsequential manner.

RARP. Reverse Address Resolution Protocol.

recursive. Pertaining to a process in which each step makes use of the results of earlier steps.

Remote Execution Protocol (REXEC). A protocol that allows the execution of a command or program on a foreign host. The local host receives the results of the command execution. This protocol uses the REXEC command.

remote host. Any *foreign host*, not including the local host.

remote logon. The process by which a terminal user establishes a terminal session with a remote host.

Remote Procedure Call (RPC). A facility that a client uses to request the execution of a procedure call from a server. This facility includes a library of procedures and an eXternal data representation.

Request For Comments (RFC). A series of documents that covers a broad range of topics affecting internetwork communication. Some RFCs are established as internet standards.

resolver. A program or subroutine that obtains information from a name server or local table for use by the calling program.

resource records. Individual records of data used by the Domain Name System. Examples of resource records include the following: a host's Internet Protocol addresses, preferred mail addresses, and aliases.

return code. (1) A code used to influence the execution of succeeding instructions. (2) A value returned to a program to indicate the results of an operation requested by that program.

Reverse Address Resolution Protocol (RARP). A protocol that maintains a database of mappings between physical hardware addresses and IP addresses.

REXEC. Remote Execution Protocol.

RFC. Request For Comments.

RIP. Routing Information Protocol.

route. A specific path used to send packets to another computer.

router. A device that connects networks at the ISO Network Layer. A router is protocol-dependent and connects only networks operating the same protocol. Routers do more than transmit data; they also select

the best transmission paths and optimum sizes for packets. In TCP/IP, routers operate at the Internetwork layer. See also *gateway*.

Routing Information Protocol (RIP). The protocol that maintains routing table entries for gateways, routers, and hosts.

routing table. A list of network numbers and the information needed to route packets to each.

RPC. Remote Procedure Call.

S

scan code. A code that the keyboard generates when you press a key. Every key on a keyboard has a unique scan code associated with it.

serial line. A network media that is a de facto standard, not an international standard, commonly used for point-to-point TCP/IP connections. Generally, a serial line consists of an RS-232 connection into a modem and over a telephone line.

server. A function that provides services for users. A machine can run client and server processes at the same time.

Simple Mail Transfer Protocol (SMTP). A TCP/IP application protocol used to transfer mail between users on different systems. SMTP specifies how mail systems interact and the format of control messages they use to transfer mail.

SMTP. Simple Mail Transfer Protocol.

socket. (1) An endpoint for communication between processes or applications. (2) A pair consisting of TCP port and IP address, or UDP port and IP address.

socket interface. An application interface that allows users to write their own applications to supplement those supplied by TCP/IP.

stream. A continuous sequence of data elements being transmitted, or intended for transmission, in character or binary-digit form, using a defined format.

subdirectory. A directory contained within another directory in a file system hierarchy.

subnet. A networking scheme that divides a single logical network into smaller physical networks to simplify routing.

subnet address. The portion of the host address that identifies a subnetwork.

subnet mask. A mask used in the IP protocol layer to separate the subnet address from the host portion of the address.

subnetwork. Synonymous with *subnet*.

synchronous. (1) Pertaining to two or more processes that depend on the occurrences of a specific event such as common timing signal. (2) Occurring with a regular or predictable time relationship. See *asynchronous*.

T

task. A part of a program that performs some operation. Every program has at least one task, and could have many.

task switch. A switch occurs when the tasker temporarily suspends one task in order to execute another task.

tasker. The part of TCP/IP for DOS that manages tasks. Periodically the tasker suspends the task currently executing and switches to another task.

tasking interval. The number of ticks per second that the tasker allocates to DOS and non-TCP/IP applications. Any remaining ticks for that second are allocated to TCP/IP background tasks.

TCP. Transmission Control Protocol.

TCP/IP. Transmission Control Protocol/Internet Protocol.

Telnet. The Terminal Emulation Protocol, a TCP/IP application protocol for remote connection service. Telnet allows a user at one site to gain access to a foreign host as if the user's terminal were connected directly to that foreign host.

terminal emulator. A program that imitates the function of a particular kind of terminal.

terminate and stay resident (TSR) program. A TSR is a program that installs part of itself as an extension of DOS when it is executed.

TFTP. Trivial File Transfer Protocol.

tick. A tick equals 1/18 second. Therefore, there are 18 ticks per second.

token. In a local network, the symbol of authority passed among data stations to indicate the station temporarily in control of the transmission medium.

token-ring network. A ring network that allows unidirectional data transmission between data stations by a token-passing procedure over one transmission medium, so that the transmitted data returns to the transmitting station.

Transmission Control Protocol (TCP). The TCP/IP layer that provides reliable process-to-process data stream delivery between nodes in interconnected com-

puter networks. TCP assumes that IP (Internet Protocol) is the underlying protocol.

Transmission Control Protocol/Internet Protocol (TCP/IP). A suite of protocols designed to allow communication between networks regardless of the technologies implemented in each network.

Trivial File Transfer Protocol (TFTP). A TCP/IP application primarily used to transfer files among personal computers. TFTP allows files to be sent and received, but does not provide any password protection or directory capability.

TSR. Terminate and stay resident. TSR usually refers to a terminate and stay resident program.

U

UDP. User Datagram Protocol.

user. A function that utilizes the services provided by a server. A host can be a user and a server at the same time. See *client*.

User Datagram Protocol (UDP). A packet-level protocol built directly on the IP layer. UDP is used for application to application programs between TCP/IP hosts.

virtual machine. (1) A virtual data processing system *that appears to be at the exclusive disposal of a particular user*, but whose functions are accomplished by sharing the resources of a real data processing system. (2) A functional simulation of a computer and its associated devices. Each virtual machine is controlled by a suitable operating system, such as conversational monitor system (CMS).

VM. Virtual Machine.

W

WAN. Wide area network.

well-known port. A port number that has been preassigned for specific use by a specific protocol or application. Clients and servers using the same protocol communicate over the same well-known port.

wide area network (WAN). A network that provides communication services to a geographic area larger than that served by a local area network.

window. An area of the screen with visible boundaries through which a panel or portion of a panel is displayed.

working directory. The directory in which an application program is found. The working directory becomes the current directory when the application is started.

X

XDR. eXternal Data Representation.

Numerics

3270. The designation for the standard terminal that is used to connect to IBM mainframes.

Bibliography

TCP/IP for DOS Publications

The following are books associated with TCP/IP Version 2.0 for DOS.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for DOS: Installation and Maintenance, SC31-6154.

This book provides system programmers, network administrators, and PC users responsible for installing TCP/IP for DOS with the information required to plan and implement the installation of TCP/IP for DOS. The topics include hardware and software requirements, pre-installation system performance considerations, instructions for installing TCP/IP for DOS, instructions for customizing the TCP/IP for DOS environment and installation examples.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for DOS: Programmer's Reference, SC31-6153.

This book is written for application and system programmers to aid them in writing application programs that use TCP/IP for DOS on a PC. Application programmers should know the DOS operating system, and have knowledge of multitasking operating system concepts. Application programmers should be knowledgeable in the C programming language.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for DOS: User's Guide, SC31-6152.

This book is written for people who use a PC with TCP/IP for DOS, such as end users and system programmers. The people who use this book should be familiar with DOS and the PC, and also understand DOS operating system concepts.

Other TCP/IP Publications

The following are other TCP/IP publications.

General Publications

The following list shows selected TCP/IP publications.

Introducing IBM's Transmission Control Protocol/Internet Protocol Products for OS/2, VM, and MVS, GC31-6080.

This book introduces managers, system designers, programmers, and other data processing personnel to the basic concepts of IBM's TCP/IP products for OS/2, VM, and MVS. This book also describes the relationship between IBM's TCP/IP implementations and other IBM products, including those based on SNA.

Internetworking With TCP/IP Volume I: Principles, Protocols, and Architecture, Douglas E. Comer, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

Internetworking With TCP/IP Volume II: Implementation and Internals, Douglas E. Comer, Prentice Hall, Englewood Cliffs, New Jersey, 1991.

MVS Publications

The following list shows the MVS publications contained in the TCP/IP for MVS library.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for MVS: Installation and Maintenance, SC31-6085.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for MVS: Programmer's Reference, SC31-6087.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for MVS: User's Guide, SC31-6088.

MVS/DFP Version 3 Release 3: Using the Network File System Server, SC26-4732.

OS/2 Publications

The following list shows the OS/2 publications contained in the TCP/IP for OS/2 library.

IBM Transmission Control Protocol/Internet Protocol Version 1.2 for OS/2: Installation and Maintenance, SC31-6075.

IBM Transmission Control Protocol/Internet Protocol Version 1.2 for OS/2: Programmer's Reference, SC31-6077.

IBM Transmission Control Protocol/Internet Protocol Version 1.2 for OS/2: Quick Reference Guide, SX75-0070.

IBM Transmission Control Protocol/Internet Protocol Version 1.2 for OS/2: User's Guide, SC31-6076.

VM Publications

The following list shows the VM publications contained in the TCP/IP for VM library.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for VM: Installation and Maintenance, SC31-6082.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for VM: Programmer's Reference, SC31-6084.

IBM Transmission Control Protocol/Internet Protocol Version 2.0 for VM: User's Guide, SC31-6081.

Other Related Publications

The following are other related publications.

BIOS Technical Publications

The following list shows selected BIOS technical publications.

IBM Personal System/2 and Personal Computer BIOS Interface Technical Reference, S68X-2341.

System BIOS for IBM PC/XT/AT Computers and Compatibles: The Complete Guide to ROM-Based System Software, Phoenix Technologies Ltd., Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1989, ISBN 0-201-51806-6.

DOS Publications

The following list shows selected DOS publications.

Disk Operating System Version 3.30 Reference, Z628-0224.

Disk Operating System Version 3.30 User's Guide, Z628-0224.

Using Disk Operating System Version 4.00, Z628-0258.

Getting Started with Disk Operating System Version 4.00, Z628-0258.

Using Disk Operating System Version 5.00, Z84F-9779.

Getting Started with Disk Operating System Version 5.00, Z84F-9779.

DOS Technical Publications

The following list shows selected DOS technical publications.

IBM Disk Operating System Version 3.30 Technical Reference, S628-0059.

IBM Disk Operating System Version 4.00 Technical Reference, S628-0254.

The MS-DOS Encyclopedia, Microsoft Press, Redmond, Washington, 1988. ISBN 1-55615-049-0.

MS-DOS Developer's Guide, J. Angermeyer, K. Jaeger, Howard W. Sams & Co., Indianapolis, Indiana, 1986, ISBN 0-672-22409-7.

COMPUTE!'s Mapping the IBM PC and PCjr, R. Davies, COMPUTE! Publications, Inc., Greensboro, North Carolina, 1985, ISBN 0-942386-92-2.

Advanced MS-DOS Programming, R. Duncan, Microsoft Press, Redmond, Washington, 1988, ISBN 1-55615-157-8.

The Peter Norton Programmer's Guide to the IBM PC, P. Norton, Microsoft Press, Redmond, Washington, 1985, ISBN 0-914845-46-2.

Programming Publications

The following list shows selected programming publications.

IBM AIX Operating System Version 1.2.1 Technical Reference Volume 1, SC23-2300.

IBM AIX Operating System Version 1.2.1 Technical Reference Volume 2, SC23-2301.

IBM AIX Version 3 for RISC/6000 Communication Concepts and Procedures Volume 1, GC23-2203.

IBM AIX Version 3 for RISC/6000 Communication Concepts and Procedures Volume 2, GC23-2203.

IBM AIX Version 3 for RISC/6000 Communications Programming Concepts, SC23-2198.

Networking on the Sun Workstation: Remote Procedure Call Programming Guide (800-1324-03), Sun Microsystems, Inc.

Network Programming, 800-1779-10, Sun Microsystems, Inc.

UNIX Network Programming, W. Richard Stevens, Prentice Hall, Englewood Cliffs, New Jersey, 1990, ISBN 0-13-949876-1.

UNIX Programmer's Reference Manual (4.3 Berkeley Software Distribution, Virtual VAX-11 Version). Department of Electrical Engineering and Computer Science. University of California, Berkeley, 1988.

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Readers' Comments

**IBM Transmission Control Protocol/
Internet Protocol Version 2.0 for DOS:
Programmer's Reference**

Publication No. SC31-6153-0

Use this form to tell us what you think about this manual. If you have found errors in it, or if you want to express your opinion about it (such as organization, subject matter, appearance) or make suggestions for improvement, this is the form to use.

To request additional publications, or to ask questions or make comments about the functions of IBM products or systems, you should talk to your IBM representative or to your IBM authorized remarketer. This form is provided for comments about the information in this manual and the way it is presented.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute your comments in any way it believes appropriate without incurring any obligation to you.

Be sure to print your name and address below if you would like a reply. If we have questions about your comment, may we call you? If so, please include your phone number.

Name _____ Address _____

Company or Organization _____ _____

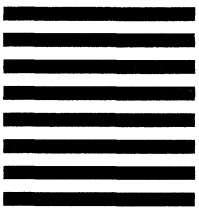
Phone No. _____ _____

Readers' Comments
SC31-6153-0

Cut or Fold
Along Line

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

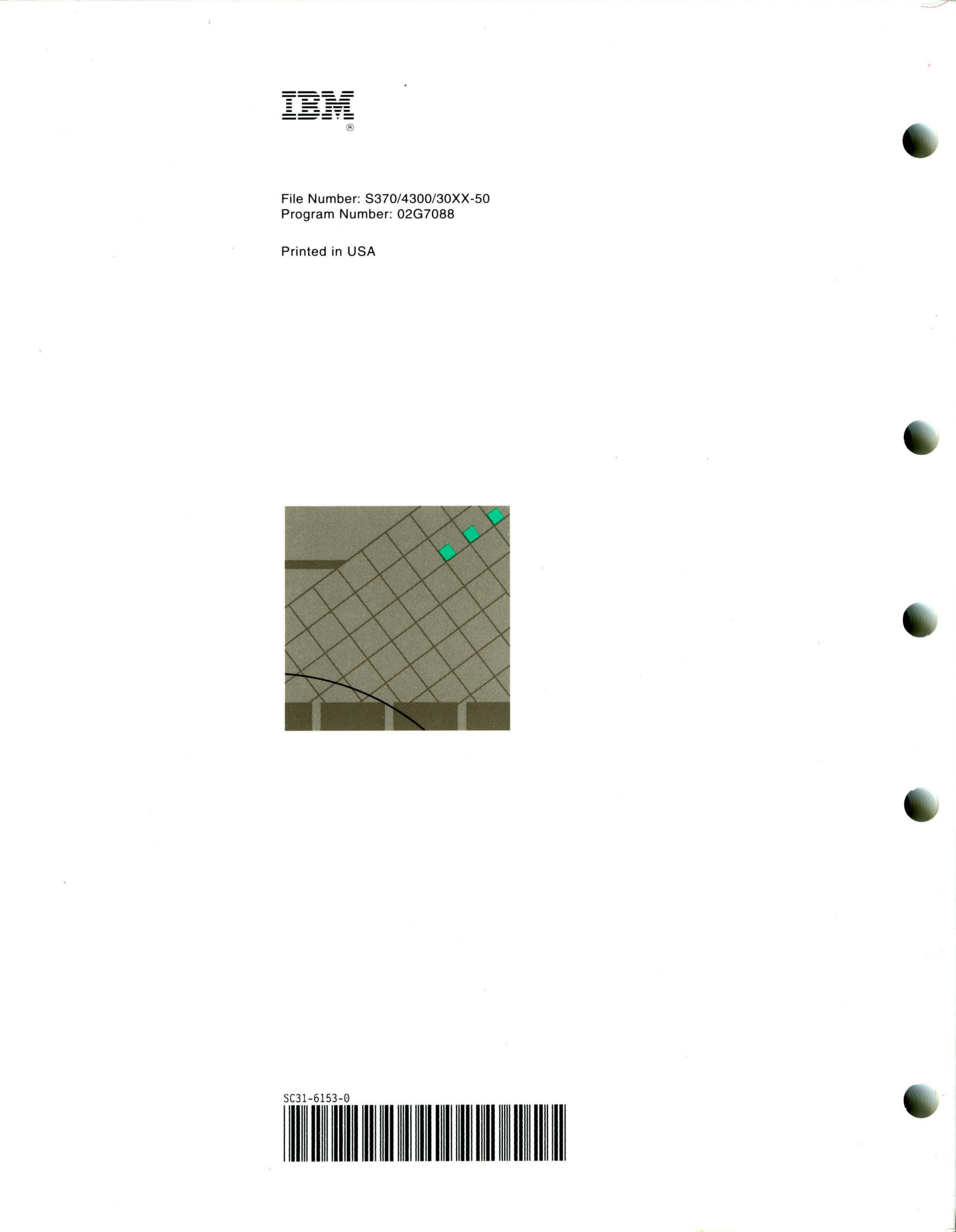
POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Information Development
Department E15
PO BOX 12195
RESEARCH TRIANGLE PARK, NORTH CAROLINA 27709-9990

Fold and Tape | **Please do not staple** | Fold and Tape

SC31-6153-0

Cut or Fold
Along Line



File Number: S370/4300/30XX-50
Program Number: 02G7088

Printed in USA

SC31-6153-0