Data General

Customer Documentation

Programming with TCP/IP on the DG/UX™ System

AViiON®
PRODUCT LINE

Programming with TCP/IP on the DG/UX™ System

093-701024-02

For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.

Ordering No. 093-701024

Printed in the United States of America
Revision 02, June 1991

Programming With TCP/IP on the DG/UX™ System
093-701024-02

Revision History:		*Effective with:*
Original Release	November 1988	DG TCP/IP (DG/UX™) Rel 4.0
First Revision	May 1990	TCP/IP for AViiON® Systems 4.30
Second Revision	June 1991	TCP/IP for AViiON® Systems 5.4

A vertical bar (|) in the margin of a page indicates substantive technical change from the previous revision. (The exception is Chapter 7, which contains entirely new material.)

Preface

This manual is intended to help you write networking applications that use components of the TCP/IP package. Specifically, this manual describes how to use system calls and library routines to access the Transmission Control Protocol (TCP), the User Datagram Protocol (UDP), and the Internet Protocol (IP). The set of calls that you use to access these protocols is commonly called the socket family of system calls. The library of routines that you use to access these protocols is called the Transport Layer Interface, or TLI for short.

Who Should Read This Manual?

This manual is for experienced applications programmers who want to develop programs that use TCP/IP on the DG/UX™ operating system. This manual assumes that you are thoroughly familiar with the C programming language, and that you understand the programming environment provided by the UNIX® operating system.

How This Manual Is Organized

This manual contains eight chapters, two appendixes, a glossary, and a list of related documents.

Chapter 1 Generally discusses how networking applications work. It introduces the terms "interface" and "protocol" and the notion of peer processes, and it also discusses connection-oriented versus connectionless communication and the client/server model of communication.

Chapter 2 Introduces the TCP/IP for AViiON™ Systems package and generally describes how networking applications use it. It also introduces sockets and the TLI.

Chapter 3 Tells how to create and name sockets, communicate through connection-oriented and connectionless sockets, set socket options at the socket level, perform operations on communication devices, multiplex input/output, and close sockets.

Chapter 4 Discusses how to write programs that use TCP. It tells how to set socket options at the transport level and discusses the notion of urgent data. It also includes two sample programs.

Chapter 5	Discusses how to write programs that use UDP. It also includes two sample programs.
Chapter 6	Discusses how to write programs that use IP. It tells how to set socket options at the IP level. It also includes a sample program.
Chapter 7	Describes how to use the TLI to access TCP/IP. It compares specific TLI routines to their system call counterparts. It also includes some sample programs.
Chapter 8	Provides manual pages of interest to those who program in the TCP/IP for AViiON Systems programming environment.
Appendix A	Lists the error messages that you could encounter when you use socket system calls.
Appendix B	Describes network library routines that aid in mapping hostnames to network addresses, network names to network numbers, protocol names to protocol numbers, and service names to port numbers.
Glossary	Provides a glossary of technical terms used in this manual.
Related Documents	Lists the documents that provide information beyond the scope of this manual.

How to Read this Manual

If you are an experienced programmer with no networking expertise, read the first three chapters and then read additional chapters as you need. If you know something about networking but are unfamiliar with TCP/IP, read Chapter 2. If you are generally familiar with TCP/IP but are unfamiliar with sockets, read Chapter 3. If you know how to use sockets, but are unfamiliar with the TLI, read Chapter 7. Otherwise, you can start reading at any point that is appropriate.

Readers, Please Note

Data General manuals use certain symbols and styles of type to indicate different meanings. The Data General symbol and typeface conventions used in this manual are defined in the following list. You should familiarize yourself with these conventions before reading the manual.

This manual also presumes the following meanings for the terms "command line," "format line," and "syntax line." A command line is an example of a command string that you should type verbatim; it is preceded by a system prompt and is followed by a delimiter such as the curved arrow symbol for the New Line key. A format line shows how to structure a command; it shows the variables that must be supplied and the available options. A syntax line is a fragment of program code that shows how to use a particular routine; some syntax lines contain variables.

Convention	Meaning
boldface	In command lines and format lines: Indicates text (including punctuation) that you type verbatim from your keyboard. All DG/UX commands, pathnames, and names of files, directories, and manual pages also use this typeface.
`constant width/ monospace`	Represents a system response on your screen. Syntax lines also use this font.
italic	In format lines: Represents variables for which you supply values; for example, the names of your directories and files, your username and password, and possible arguments to commands. In text: Indicates a term that is defined in the manual's glossary.
[*optional*]	In format lines: These brackets surround an optional argument. Don't type the brackets; they only set off what is optional. The brackets are in regular type and should not be confused with the boldface brackets shown below.
[ ]	In format lines: Indicates literal brackets that you should type. These brackets are in boldface type and should not be confused with the regular type brackets shown above.
...	In format lines and syntax lines: Means you can repeat the preceding argument as many times as desired.
`$` and `%`	In command lines and other examples: Represent the system command prompt symbols used for the Bourne and Korn shells and the C shell, respectively. Note that your system might use different symbols for the command prompts.
↵	In command lines and other examples: Represents the New Line key, which is the name of the key used to generate a new line. (Note that on some keyboards this key might be called Enter or Return instead of New Line.) Throughout this manual, a space precedes the New Line symbol; this space is used only to improve readability — you can ignore it.
< >	In command lines and other examples: Angle brackets distinguish a command sequence or a keystroke (such as <**Ctrl-D**>, <**Esc**>, and <**3dw**>) from surrounding text. Note that these angle brackets are in regular type and that you do not type them; there are, however, boldface versions of

these symbols (described below) that you do type.

<, >, >> In text, command lines, and other examples: These boldface symbols are redirection operators, used for redirecting input and output. When they appear in boldface type, they are literal characters that you should type.

Contacting Data General

Data General wants to assist you in any way it can to help you use its products. Please feel free to contact the company as outlined below.

Manuals

If you require additional manuals, please use the enclosed TIPS order form (United States only) or contact your local Data General sales representative. A list of related documents appears at the end of this manual with the TIPS order form.

For a complete list of AViiON® and DG/UX™ manuals, see the *Guide to AViiON® and DG/UX™ Documentation* (069-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Telephone Assistance

If you are unable to solve a problem using any manual you received with your system, free telephone assistance is available with your hardware warranty and with most Data General software service options. If you are within the United States or Canada, contact the Data General Service Center by calling 1-800-DG-HELPS. Lines are open from 8:00 a.m. to 5:00 p.m., your time, Monday through Friday. The center will put you in touch with a member of Data General's telephone assistance staff who can answer your questions.

For telephone assistance outside the United States or Canada, ask your Data General sales representative for the appropriate telephone number.

Joining Our Users Group

Please consider joining the largest independent organization of Data General users, the North American Data General Users Group (NADGUG). In addition to making valuable contacts, members receive FOCUS monthly magazine, a conference discount, access to the Software Library and Electronic Bulletin Board, an annual Member Directory, Regional and Special Interest Groups, and much more. For more information about membership in the North American Data General Users Group, call 1-800-877-4787 or 1-512-345-5316.

End of Preface

Contents

Chapter 1 — Introduction to Networking Applications

Chapter 2 — Introduction to TCP/IP, Sockets, and the TLI

Chapter 3 — Programming with Sockets

Chapter 4 — Programming With the Transmission Control Protocol

Chapter 5 — Programming with the User Datagram Protocol

Chapter 6 — Programming with the Internet Protocol and Internet Control Message Protocol

Chapter 7 — Using the Transport Layer Interface to Access TCP/IP

Tables

Figures

Chapter 1
Introduction to Networking Applications

This chapter generally describes how networking applications work. It introduces the terms *interface* and *protocol*, and it discusses the notion of *peer processes*. It then contrasts connection-oriented communication with connectionless communication. Finally, the chapter describes the client/server model of communication.

Successful programming with TCP/IP on the DG/UX system requires a firm grasp of the topics covered in this chapter. If you are already familiar with these topics, you can proceed to Chapter 2, which introduces the TCP/IP package and sockets.

The Structure of a Network

A network consists of a group of hosts using special hardware and software to communicate with one another. Networks can be complex. To help simplify them, designers organize networks into layers. Usually, layers are set up hierarchically.

The number of layers and each layer's function can vary from network to network. In all networks, though, each layer provides services to the higher layers, and the higher layers do not bother with the details of how the service is provided.

An interface consists of the types and forms of messages that each layer uses to communicate with the layers above or below it. It defines the services that a layer provides and the format of the data that a layer exchanges with its neighbors. A protocol specifies how programs on different computers but at the same layer communicate. As you will see as you progress through this chapter, the term protocol has slightly different meanings depending on the context in which it is used. The set of layers, interfaces, and protocols that govern communication is called a *network architecture*.

When a designer decides how many layers to put in a network and what each layer should do, she or he designs each layer so that it performs well defined and well understood functions. For example, one layer could be designed to regulate the flow of messages to the next higher layer. Another may be created to compress data for the next highest layer.

Usually the highest layer of an architecture contains user interface programs, which allow access to the lower layers of network software. This highest layer could consist of a simple set of command lines or a complex system of menus; the same set of services would be provided with either implementation.

The lowest layer is always the physical layer, where two systems actually connect. This layer could consist of wires or of microwaves; again, the same set of services would be provided with either implementation.

Standards organizations, in an effort to standardize communications protocols, have proposed a variety of specific network architectures. One of the most comprehensive proposals put forward is a seven-layer network architecture created by the International Standards Organization (ISO) called Open Systems Interconnection (OSI). Figure 1-1 shows two hosts communicating through a network that conforms to the OSI model.

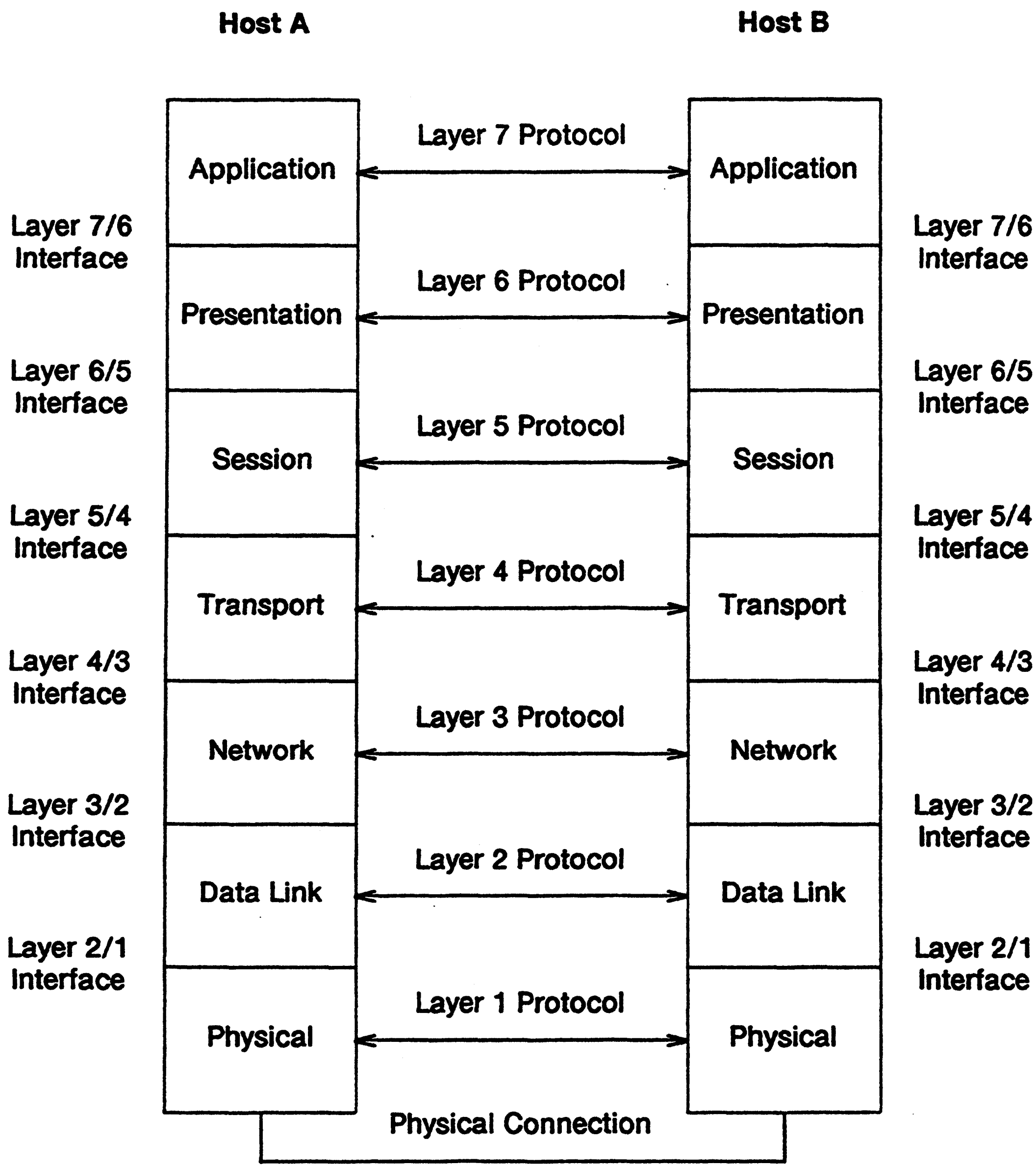


Figure 1-1 *OSI Seven-Layer Network Architecture*

When you write applications that use TCP/IP, you are most often concerned with what happens at the data link, network, and transport layers. The data link layer takes the wires or microwaves at the physical layer and transforms them into a channel that appears free of transmission errors. The network layer manages host-to-host communication. It is also concerned with the characteristics of the interface between the data link and the host and with how packets are routed through the network. The transport layer is responsible for data transmission between programs. Data exchange between programs is described in more detail in the next section.

Regardless of the layer at which it exists, a network application uses interface and protocol software to do work over a communication channel. To understand how this happens requires understanding how peer processes work.

What Are Peer Processes?

A *process* is a program in execution. Network software operates on a simple principle: a process at one layer on one host carries on a conversation with a process at the same layer on another host. In this context, a protocol specifies the set of rules that processes at comparable layers on local and remote systems use to communicate. Processes communicating at corresponding layers on different hosts are called *peer processes*.

The conversation that peer processes carry on is not a direct one. What really happens is that data and control information are sent from the layer at which one peer process exists down to the next layer, down to the lowest layer, across a physical medium, to the lowest layer on the other host, up to the next layer, and then up to the layer at which the other peer process exists. With cleanly defined interfaces and well-defined protocols, these details need never concern a user. With a well-defined programming interface, these details need never concern the network programmer either, but the programmer benefits from generally understanding the process.

Understanding Connection-oriented versus Connectionless Communication

At this point, you should understand that network applications carry on what appear to be direct conversations with peers but what are actually exchanges that involve lower layers of the network architecture. This virtual conversation between peers obeys the appropriate protocol for the layer. At the application interface, there are two general types of services offered: connection-oriented and connectionless.

Using *connection-oriented services* is similar to using the telephone. For example, if Greg wants to send Mike several messages, he could call Mike using the telephone service. Assuming that Greg dials the correct telephone number and the telephone lines are operable, a connection is established when Mike answers the phone. Greg's messages are delivered to Mike in the order that he sends them. This is considered a "stream-oriented" service (not to be confused with STREAMS, a facility invented at AT&T). Every now and then during the communication, Mike may indicate to Greg that he has received the messages sent (for example, say "uh-huh"). This is considered a "reliable" service.

Using *connectionless services*, on the other hand, is similar to using a postal service. In the previous example, if Greg sends the messages through a series of letters, Greg would first place Mike's complete address on envelopes. He would place the messages in the envelopes and release the letters to a postal service. This is considered a "datagram-oriented" service. The postal service chosen delivers the letters to Mike's address, but does not guarantee that the letters will arrive in order, nor that the letters will arrive at all. Though the chosen postal service is not as reliable as the phone system, it does provide reasonable service. Unlike when a connection is established, Greg has no way of knowing if and when the letters arrive. This is considered an "unreliable" service.

In summary, connection-oriented services are reliable and stream-oriented. Connectionless services are packet-oriented but "unreliable" because messages are neither guaranteed to arrive in order nor guaranteed to arrive at all.

Understanding the Client/Server Model

Communication cannot take place without some kind of underlying model to structure events. That is, there has to be a mutually agreed upon assignment of roles (who goes first?) and sequence of events (what do we say after hello?) for communication to occur between two parties. The client/server model provides a way for two communicating programs to relate to one another. The model describes how connections are initiated and how communicating parties interact.

An idea central to the client/server model is that services are desired and available on the network. *Server* programs provide these services and *client* programs use them.

A client program typically initiates the client/server relationship. A client has two interfaces: one to the end user and one to the server. You would start a client program when you want to use a service. Once started, the client program uses its protocol software to seek out the server program and request the service.

A server program offers services to the network community. There is typically a single server program on each host that provides a service to all clients that request it. Like any service provider in the real world, server programs make themselves easy for clients to find. Servers do the equivalent of listing their phone numbers in a public directory by registering their service "numbers" in a place known to clients.

Typically, a server program runs as a daemon process started at boot time that constantly listens for service requests. When such a daemon receives a service request, it "wakes up," quickly provides the client the requested service, and then goes back to listening for more requests. Most of the time (especially if the service provided is time consuming), a server daemon spawns a child to service the specific request, so that it may go back to listen for more requests. Thus the server daemon may service many requests at the same time.

The client process and the server process are peers. They must use the same communication conventions. In this context, a protocol specifies a formal and exhaustive definition of the conventions required by peer processes.

If a system runs several server programs at one time, each listening for service requests, the system could get clogged and performance could deteriorate. Rather than run this risk, a system can run a single server program that listens for various service requests and then passes the request to another server. For servers using TCP/IP for AViiON Systems, this server program is called **inetd**. The **inetd** daemon listens at a variety of ports specified in a configuration file. When a connection is requested to a port on which **inetd** is listening, **inetd** executes the appropriate server program to service the client. Clients are unaware that an intermediary such as **inetd** has played any part in the connection. For details about **inetd(1M)**, see the manual page.

With this background, you are equipped to learn about the protocols, interfaces, and client and server programs provided by TCP/IP. The next chapter generally describes the TCP/IP for AViiON Systems package and introduces its programming interface: sockets.

End of Chapter

Chapter 2
Introduction to TCP/IP, Sockets, and the TLI

The previous chapter generally described how networking applications work. This chapter introduces the TCP/IP for AViiON Systems package and describes how networking applications access it through sockets.

What is TCP/IP for AViiON Systems?

TCP/IP for AViiON Systems is a package of communications software that implements the TCP/IP family of networking protocols on the DG/UX operating system. The package consists of several kernel-level protocols, server programs, administrative utilities, user commands, and user-level protocols.

The *Defense Advanced Research Project Agency (DARPA)* developed the Internet protocols for the *ARPANET* network project. The University of California at Berkeley developed the 4.2 *Berkeley Software Distribution (BSD)* release of the UNIX® operating system based on the DARPA work. Data General developed the TCP/IP for AViiON Systems software package from the Berkeley release, substantially revising it to comply with the *Defense Data Network (DDN)* specifications. Many BSD 4.3 features subsequently have been added to the TCP/IP for AViiON Systems package.

Figure 2-1 shows a representation of the TCP/IP for AViiON Systems network architecture.

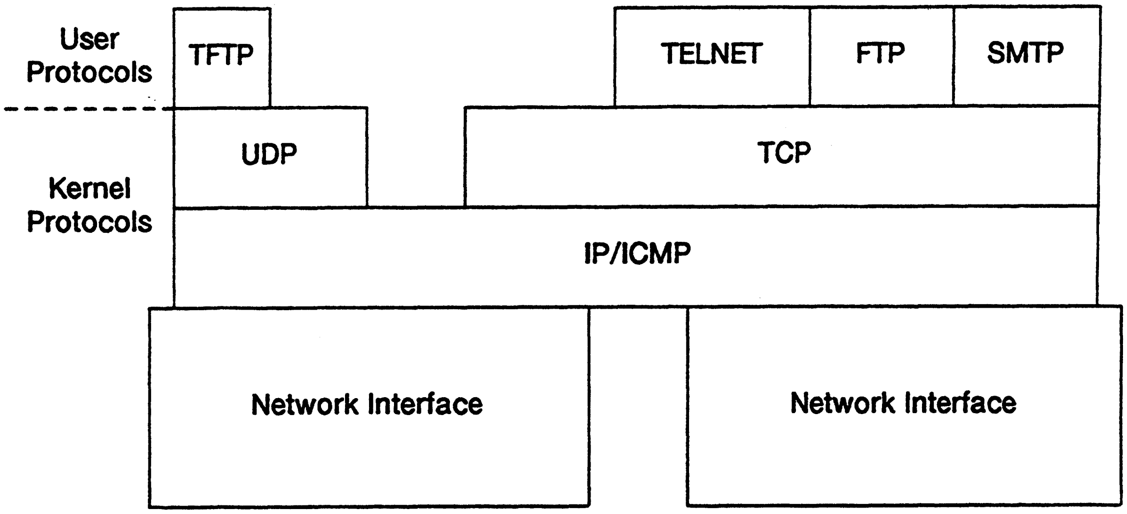


Figure 2-1 *TCP/IP for AViiON Systems Network Architecture*

The following sections discuss the software at each layer of this architecture.

Introduction to the Network Interface

At the lowest layer are network interfaces. They prepare IP traffic for transmission onto physical media and receive traffic from the media to deliver it to IP. Typical network interfaces include a device driver, which is kernel-level software that manages the communications hardware installed in the computer. TCP/IP currently uses network interfaces for use on IEEE 802.3/Ethernet media, IEEE 802.5/Token Ring media, and IXE (Internet to X.25 Encapsulation) interfaces for use on X.25 networks (synchronous lines). For more information about X.25, see *Setting Up and Managing X.25 on the DG/UX™ System.*

To prepare traffic for transmission, a network interface translates an IP address into an address that can be used on the underlying physical medium. Typically, the IP datagram is then encapsulated into a media-specific frame and sent to a physical communications device for delivery to the physical network.

When receiving traffic, a network interface accepts frames from a communications device and strips any network/media specific information from them. What remains is an IP packet, which is then delivered to IP.

Introduction to the Kernel-level Protocols

Kernel-level protocols are layered on top of the network interfaces. Kernel-level protocols include network protocols and transport protocols.

The network protocols include the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). IP is concerned only with host-to-host communication. Its job is to get a *datagram*, which is a self-contained packet of data carrying its source and destination address, to the next host on the route to the datagram's final destination. If an intermediate host is not available, IP examines routing information that it keeps to find a new path through the network. Since host availability changes, the packets that make up a complete message may have different routes and may end up at the destination out of their original order. Some packets may be lost, garbled, or duplicated in transmission.

ICMP handles error and control messages. Hosts use ICMP to send reports of problems about datagrams to the source of the datagram. It also provides an echo request/reply service to test whether a destination can be reached and is responding.

At the next layer up from the network protocols are two transport protocols, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). Transport protocols provide a mechanism that processes use to communicate.

TCP provides a reliable, full-duplex byte stream between communication endpoints. Applications use this byte stream to move data, such as files and messages. More specifically, TCP breaks a user's data stream into packets and passes these packets to IP. When applications use TCP, record boundaries may not be preserved.

When packets arrive at their destination, TCP reconstructs the data stream, checking to ensure that the data is complete and correct before sending it to an application program. If there is a problem, TCP causes the appropriate retransmissions.

Like TCP, UDP fits into the layered network architecture just above IP. UDP is a simple protocol that provides a way to deliver datagrams between process endpoints. It does not check that any datagrams were delivered or check for duplicate datagrams. When applications use UDP, record boundaries are preserved.

Introduction to the User-level Protocols

After the transport protocols come four user-level protocols: TFTP, TELNET, FTP, and SMTP. They are user-level because programs that use them execute code in user space. These protocols provide virtual terminal service, file transfer service, and electronic mail service between systems.

User programs implement the user-level protocols. For example, the **ftp** program implements the client side of the FTP protocol (the server side runs as **ftpd**). The **sendmail** program implements the client side of the *Simple Mail Transfer Protocol (SMTP)*, which allows the transmission of mail messages (the server side runs as **smtp**). For information about **sendmail**, see *Managing TCP/IP on the DG/UX™ System*.

Introduction to the DG/UX System Socket Interface

The basic building block for network communication through IP, TCP, and UDP is the *socket*. Socket is a term that can be used three ways. The first use of the term is conceptual: a socket is simply a communication endpoint that can be given a name. The second use of the term refers to the set of system calls that a programmer can use to access protocol software. When you refer to the socket interface, this is the sense of the term implied. The socket family of system calls implement the interfaces that open, name, transmit and receive data, and close communication endpoints. The calls provide support for both connection-oriented (TCP) and connectionless (UDP, IP) communication. The third use of the term refers to the **socket** system call itself. You use this system call to open a communication endpoint.

This manual most often uses the term socket in the second way. The primary purpose of this book is to describe how to use the socket family of system calls in network programs.

For now, though, it is useful to explore the first use of the term socket. Sockets (as communication endpoints) exist within communications domains. Domains are abstractions that imply both a specific addressing structure and an associated set of protocols. Sockets normally exchange data only with sockets in the same domain (it may be possible to cross between communications domains, but only if some translation process is performed).

Sockets in the DG/UX system support two communications domains: the Internet domain for process-to-process communication between hosts that communicate with one another using the DARPA standard communication protocols, such as IP, TCP, and UDP, and the UNIX domain for process-to-process communication on the same host. Naming sockets in the Internet domain is covered in Chapter 3 of this manual. In the UNIX domain, socket names are UNIX pathnames; for example, a socket may be named **/tmp/foo**. For more information about sockets in the UNIX domain, see *UNIX System V Release 4 Programmer's Guide: Networking Interfaces*.

To open a socket, you use the **socket**(2) system call. This call returns a file descriptor from which you can read and write data. This call is described in detail in Chapter 3.

As you have read, the DG/UX system implements TCP/IP functionality in the kernel. Network applications access this functionality through sockets in the Internet domain. Sockets help to establish a path from an application program on a local system through TCP/IP to an application program on a remote system. As Figure 2-2 shows, programs that implement user-level protocols such as TFTP and user-written applications access TCP/IP functionality in the kernel through the socket interface.

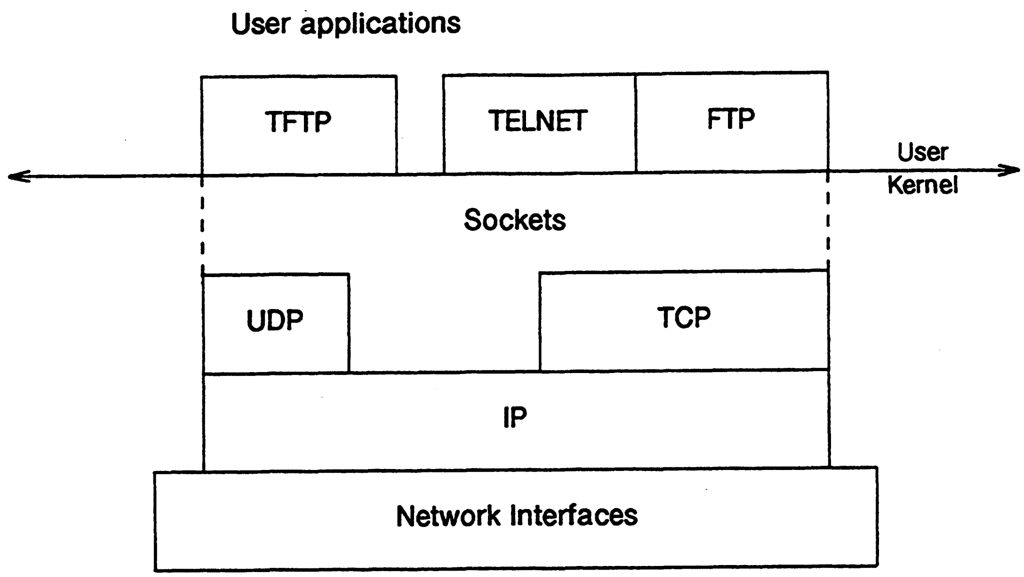


Figure 2-2 *TCP/IP Socket Interface*

Introduction to Socket Types

Within each domain, sockets are grouped into types according to the communication properties they provide. In general, processes communicate between sockets of the same type only.

Three types of sockets are available: stream, datagram, and raw. In the Internet domain, these types have specific uses:

- Programs use stream sockets to access TCP protocol software. Stream sockets send and receive data in continuous streams of bytes without logical breaks or duplication. Data can pass through the socket in both directions simultaneously, guaranteeing delivery in the original order in which the data is sent. Except for two-way data flow, stream sockets provide an interface similar to that of pipes.

- Programs use datagram sockets to access UDP protocol software. Datagram sockets send data in and receive data from both directions simultaneously, preserving logical breaks in the data, that is, data is delivered in complete packets rather than streams of bytes. The packets may arrive out of order or may fail to be delivered. Packets may also be duplicated (delivered more than once). Datagram sockets closely model the facilities found in many contemporary packet-switched networks.

- Raw sockets allow access to IP or ICMP. Raw sockets send information in datagrams. Raw sockets are provided mainly for those interested in developing new communication protocols, or for gaining access to some of the more esoteric facilities of an existing protocol. Only superusers can use raw sockets.

How Sockets Provide Peer-to-peer Communication

As you have read, IP is concerned only with getting a datagram from one host to the next. All hosts running TCP/IP understand how to handle IP datagrams. If there is not a direct path between hosts, *routers* accept packets from one physical network and forward them to hosts or routers on another.

Getting datagrams from one host to another requires that every network interface has a unique Internet address. An *Internet address* is a 32-bit number that represents a network and a host. For a thorough discussion of Internet addressing, address classes, and subnetting, see *Managing TCP/IP on the DG/UX™ System*.

Typically, applications require more than the host-to-host services provided by IP. They must be able to associate incoming data with the appropriate process. TCP and UDP provide such services by offering a set of *ports* within each host. A port is a number associated with a communications endpoint. A single process can communicate with several remote processes simultaneously. Each process can have several ports, using each to communicate with the port of a different remote process. Because each local TCP or UDP assigns port numbers independently, a particular port number is not guaranteed to be unique across an entire network. For unique identification, the port number is concatenated with the host's Internet address.

Figure 2-3 broadly illustrates the path that data follow through an implementation of TCP/IP.

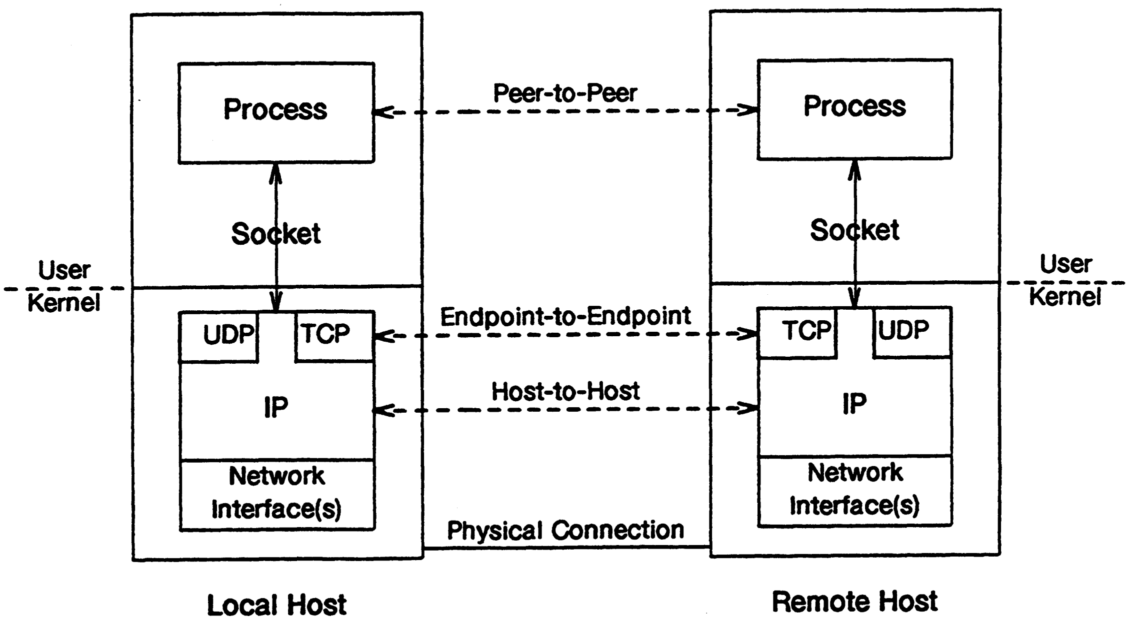


Figure 2-3 *TCP/IP Process Diagram*

The Internet address identifies a network interface on which IP is running. Each interface's Internet address is unique across the Internet; each port number is unique within the host. Thus, the combination of Internet address and port number is unique for every host on the Internet. Because of its uniqueness, the concatenation of Internet address and port number can effectively specify communication endpoints through which one peer process can use a socket to communicate with another. That is, the concatenation of address and port number can serve as the name of a communication endpoint, or the *socket name*. One process can use several different socket names at the same time.

To open, use, and discard sockets (communication endpoints), you use the socket family of system calls. The next chapter describes these system calls in some detail.

Introduction to the Transport Layer Interface

The Transport Layer Interface (TLI) is a user library developed by AT&T that uses STREAMS mechanisms to access transport-level services in the kernel. The interface to transport services is designed so that higher-layer applications can use these services without having to deal with all of the details of the underlying transport protocol. Thus, a programmer can write applications to access transport services provided by TCP/IP, an ISO (International Standards Organization) protocol, or a Netware protocol using a single access method, TLI.

Sockets and the TLI both provide programming interfaces to the transport layer of the TCP/IP network architecture. Sockets provide a specific interface to TCP, UDP, and IP. The TLI provides a generic interface to transport services through library routines. TLI routines manipulate kernel-resident information that conforms to the Transport Provider Interface (TPI), which is a message-based STREAMS protocol. The TPI is designed to provide a general interface between any given *transport provider* and any given *transport user*.

A transport provider is any set of routines that provide communications support at the transport layer for a transport user. A transport user, which could be any application program or session-layer software, obtains this support by issuing service requests, such as one to transfer data over a connection. The transport provider notifies the transport user of events such as the arrival of data on a connection.

To make use of the programming interface provided by the TLI, a transport user links in the TLI library. When you execute a transport user, TLI routines build a stream to access a TPI-compliant transport provider, which in turn accesses the protocol- and device-specific STREAMS modules and drivers needed to provide the transport service. Figure 2-4 shows this arrangement.

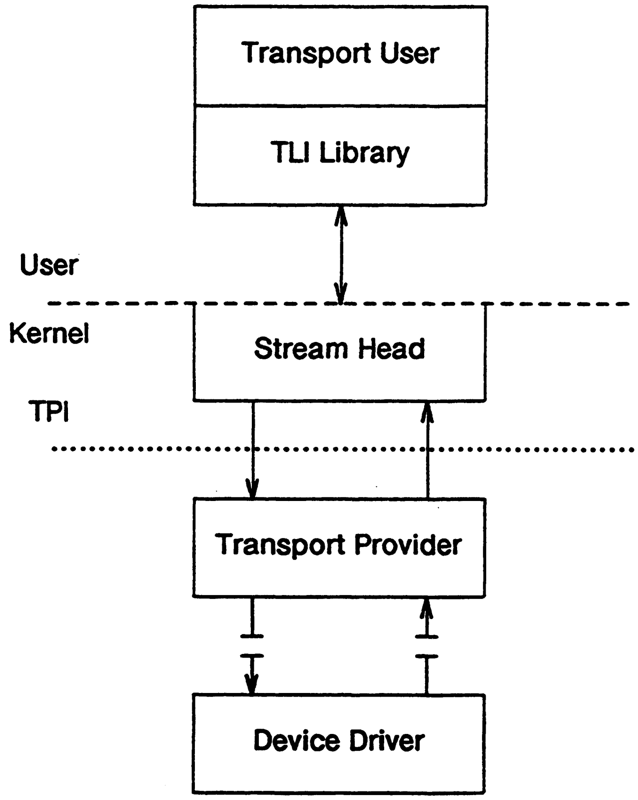


Figure 2-4 *Implementation of the TLI*

For more information about STREAMS, see the *UNIX® System V Release 4 Programmer's Guide: STREAMS*.

Chapter 1 of this manual discussed the client/server model of communication. This same model applies to networking applications that use the TLI to access TCP/IP. When a client application initiates communication with a server, the messages that pass between a transport user and a transport provider on either side conform to the Transport Provider Interface (TPI). The communication that takes place between the transport provider on a client system and the transport provider on a server conforms to TCP or UDP. Figure 2-5 illustrates this point.

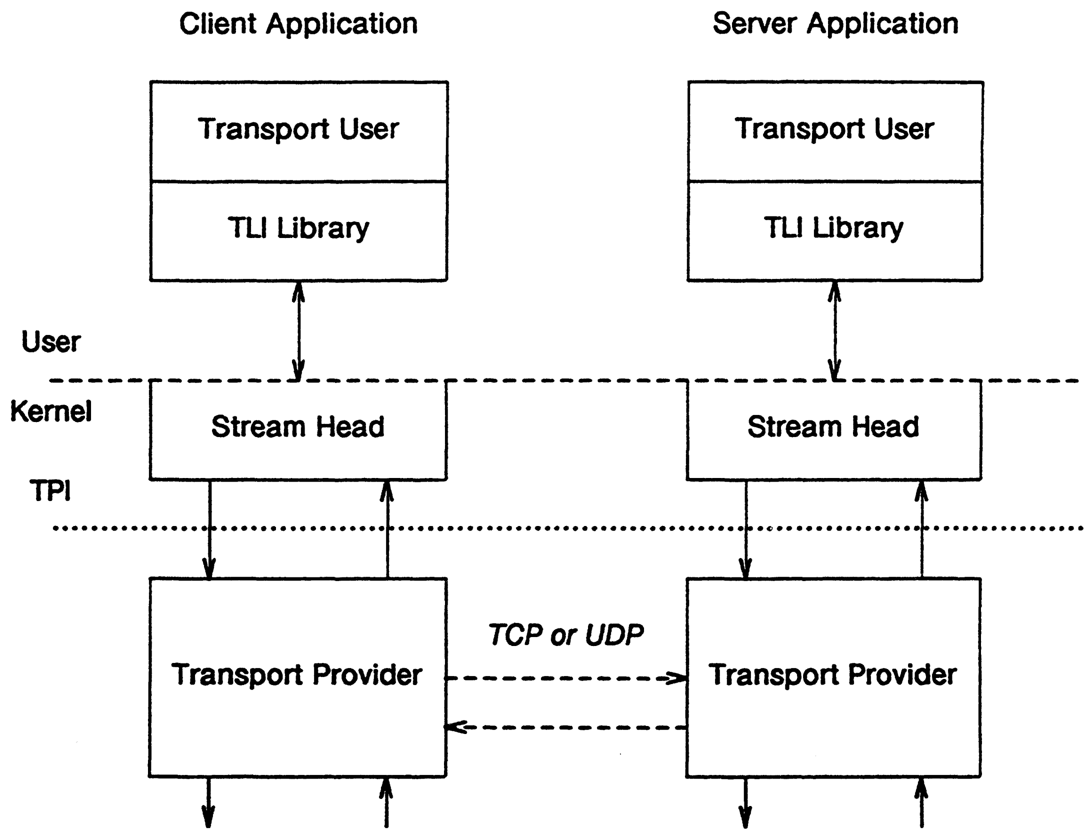


Figure 2-5 *Communication Through the Transport Provider Interface*

The TLI provides connection-oriented and connectionless services. These roughly correspond to the kinds of service provided by stream-type sockets and datagram-type sockets, respectively. As you learned in Chapter 1, connection-oriented services allow you to transmit data over an established connection in a reliable way. Connectionless services allows you to transmit data in self-contained units.

The sequence of events when a TLI-based client and a TLI-based server communicate resemble the sequence of events when a socket-based client and a socket-based server communicate. First, a local transport user (usually the client) must establish a channel of communication with its local transport provider; this channel is called the transport endpoint. Then, an address must be associated with the local endpoint. With connection-oriented service, a connection now may be established between the client and a server. Then, the client and server can transfer data to one another. When data transfer is complete, the connection is closed. For connectionless service, the client and server can transfer data immediately after the local transport endpoint is established.

The transport connection established is described in terms of the state of the transport endpoints. A transport endpoint has a current state. The TLI specifications tell how events cause a transport endpoint to change states. They also describe the events that can occur when a transport endpoint is in a particular state. For more information, see the *UNIX® System V Release 4 Programmer's Guide: Networking Interfaces*.

Chapter 7 describes in detail how to use TLI routines to access TCP/IP.

End of Chapter

Chapter 3
Programming with Sockets

The previous chapter introduced the different types of sockets and told how processes use sockets to exchange information. This chapter provides an overview to programming with sockets. It introduces the system calls that you use to open and name sockets, set socket options, communicate through sockets, perform a variety of operations on communications devices, multiplex input/output, and close sockets.

For more information about the socket system calls described in this and in later chapters, refer to the manual pages that appear online and in the *Programmer's Reference for the DG/UX™ System*.

Opening Sockets

You open a socket by issuing the **socket**(2) system call. Figure 3-1 shows the syntax of the call:

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

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

Figure 3-1 *Syntax of the socket System Call*

The *domain* is the communications domain to use (UNIX or Internet); *type* is the type of socket (stream, datagram, or raw); and *protocol* is the specific protocol in the domain specified. The **socket** call returns a descriptor (a small integer) that you may use in later system calls that operate on sockets.

Specifying a Socket's Domain, Type, and Protocol

For the UNIX domain, specify the constant **AF_UNIX** as the first argument to the **socket** call. For the Internet domain, specify the constant **AF_INET**. Domain numbers begin with the prefix **AF_**, which stands for address family.

Using sockets in the UNIX domain is beyond the scope of this manual. For more information about sockets in the UNIX domain, see *UNIX System V Release 4 Programmer's Guide: Networking Interfaces*.

Specify a socket type as the second argument to the **socket** call with one of the following constants: **SOCK_STREAM**, **SOCK_DGRAM**, or **SOCK_RAW**. In the Internet domain, these constants are associated with specific protocols, as Table 3-1 shows.

Table 3-1 Constants for the Socket Type in the socket System Call

Constant	Protocol
SOCK_STREAM	TCP
SOCK_DGRAM	UCP
SOCK_RAW	IP

The file **/usr/include/sys/socket.h** contains the definitions for all the constants for a socket's domain and type.

Optionally, you can request a particular protocol as the third argument to the **socket** call. If you specify a value of **0**, the system selects a default protocol for the socket type (for example, **SOCK_STREAM** sockets would use a protocol value of **IPPROTO_TCP**, or 6). Here is a partial listing of accepted constants for the protocol type. For a complete list, see **/usr/include/netinet/in.h**.

Table 3-2 Constants for the Protocol Type in the socket System Call

Constant	Value	Protocol
IPPROTO_TCP	6	TCP
IPPROTO_UDP	17	UDP
IPPROTO_RAW	255	IP
IPPROTO_ICMP	1	ICMP

For example, to open a stream socket in the Internet domain, you could use the following call:

```
int socket_des;
socket_des = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
   if (socket_des == -1)
       {
     .
     .
     .
       }
```

This call opens a stream socket with TCP providing the underlying communication support. Because **IPPROTO_TCP** is the default protocol for stream sockets, you alternatively could specify a 0 as the third argument. Also, as the example shows, you should set up a conditional statement to handle error conditions.

To opens a datagram socket, you could use the following call:

```
int socket_des;
socket_des = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);
   if (socket_des == -1)
       {
     .
     .
     .
       }
```

Here, we are requesting UDP to supply the protocol. Again, you should set up a conditional statement to handle error conditions.

For a table of the error conditions that could occur when you use the **socket** call, see Appendix A.

Binding Sockets

The socket opened by a **socket** call is simply a bookkeeping entry in a kernel table. In this form, it cannot be used for Internet communication; it must first be assigned to a specific Internet address and port number. As you read in the previous chapter, the concatenation of Internet address and port number is often called the socket name.

For a given protocol, any given socket name is unique throughout the Internet, and only one socket can be assigned a given name. In other words, the combination of protocol, Internet address, and port number for a given socket uniquely identifies the socket throughout the Internet.

The **bind** system call assigns a name to a socket. Figure 3-2 shows its syntax in the Internet domain:

```
#include <netinet/in.h>
#include <sys/socket.h>

int socket_des;
struct sockaddr_in name;
socket_des = socket(AF_INET,SOCK_STREAM,0);
.
.
.
bind(socket_des, &name, sizeof(name));
```

Figure 3-2 *Syntax of the bind System Call in the Internet Domain*

The **socket_des** is the file descriptor of the socket to which you are binding a name. The interpretation of the **name** varies from one communication domain to another (here, for a socket in the Internet domain, it is declared to be a structure of type **sockaddr_in**). The next section discusses this point at length. The last argument specifies the length of the name in bytes.

Naming Sockets in the Internet Domain

In the Internet domain, socket names are contained in a structure of type **sockaddr_in**, which is declared in the include file **/usr/include/netinet/in.h**. The **sockaddr_in** structure contains the following fields:

```
struct sockaddr_in    {

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

The fields that make up **sockaddr_in** are as follows:

`sin_family` Specifies the communications domain that the socket will use. To indicate the Internet domain, the entry must be **AF_INET**.

`sin_port` This field specifies the port number. A value of **0** in this field means that the system will choose an unused port number. Port numbers used for specific Internet functions are defined in **/usr/include/netinet/in.h** and in services databases such as the Network Information Service (NIS) and **/etc/services**.

`sin_addr` Specifies the Internet address portion of the socket name in a structure called **in_addr**. This structure, which is defined in **/usr/include/netinet/in.h**, lets you refer to Internet addresses either as a 32-bit integer or as 4 different bytes of information. The **in_addr** structure defines host Internet addresses as follows:

```
struct  in_addr  {
        union  {
               struct  { u_char s_b1,s_b2,s_b3,s_b4; } S_un_b;
               struct  { u_short s_w1,s_w2; } S_un_w;
               u_long  S_addr;
        } S_un;
#define   s_addr  S_un.S_addr
};
```

Here is an example of how you could fill the above structure:

```
address.sin_addr.s_addr= a<< 24 + b << 16 + c << 8 + d;
```

`sin_zero [8]` This field is used as padding to fill out the structure to match the size of the general **sockaddr** structure. This field must have the value 0.

For more information about Internet addressing, see *Managing TCP/IP on the DG/UX*™ System.

Using Wildcards in Socket Names: Implicit Binding

Binding names to sockets in the Internet domain can be complex. Servers should be able to accept connections from any network interface; they should not have to explicitly specify the name of the interface each time a connection is accepted. To allow servers to accept connections from anywhere, a wildcard address, **INADDR_ANY**, is declared in the include file **/usr/include/netinet/in.h**.

When you specify an address as **INADDR_ANY**, the system interprets the address as "any valid address." For example, to bind a specific port number to a socket, but leave the local address unspecified, the code in the following example could be used:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#define MYPORT 5001
...
int socket_des;
struct sockaddr_in address;
...
address.sin_family = AF_INET;
address.sin_addr.s_addr = INADDR_ANY;
address.sin_port = MYPORT;
bind(socket_des, &address, sizeof (address));
```

Sockets that have wildcard local addresses can receive messages directed to the specified port number and messages addressed to any of the possible addresses that have been assigned a host. For example, if a host is on networks 128.223 and 128.226, and you bind a socket as above, and then issue an **accept** call, the process will be able to accept connection requests that arrive either from network 128.223 or network 128.226. For the **connect** call, the address of the local interface chosen to send data to a remote host will be used.

Notice that in the above code fragment, the constant **MYPORT** is defined to have a value of 5001. If you define a specific port number in the code for a network application, it clearly limits the usefulness of the code. Alternatively, we could have used the **getservbyname** routine (described in the next section) to obtain a port number.

We have seen that a server process usually wishes to specify only the port number part of its socket's Internet address. Typically, client processes are unconcerned about the specifics of the local address to which they are bound other than that the address is unique. In such cases, they want to use wildcard specifications not only for the Internet address, but also for the port number. If the **sin_port** field of a **sockaddr_in** field is set to zero, the **bind** call interprets the specification to mean: "bind to any available port."

Client processes can often avoid the **bind** operation entirely. If a **connect** or **send** operation (both are described in subsequent sections) is attempted on an unbound socket, the system will bind the socket to an available address and port before proceeding with the requested operation. This is called implicit binding.

Using Network Library Routines

If you specify a single Internet address or port number in the code for a network application, it clearly limits the usefulness of the code. A number of routines are provided in the DG/UX system run-time libraries to aid in locating and constructing names or addresses from tables or databases that contain name/address pairs. These routines are helpful when you program with sockets. The include files for these routines are located in **/usr/include**.

All of the network library routines are described at length in Appendix B. Three of these routines are sufficiently useful to mention here: **gethostbyname**(3N), **gethostbyaddr**(3N), and **getservbyname**(3N).

The **gethostbyname**(3N) routine takes a hostname and returns a pointer to a **hostent** structure (see below). Since a host can have many addresses that have the same name, **gethostbyname**(3N) returns the first matching entry in the hosts database. The hosts database could be provided by the domain name system (DNS), the Network Information Service (NIS), or by **/etc/hosts**. The **gethostbyaddr** (3N) routine maps host addresses into this same **hostent** structure.

The hostname to network address mapping is represented by the **hostent** structure, which contains the following fields:

```
struct hostent {
        char        *h_name;        /* official name of host */
        char        **h_aliases;    /* alias list */
        int         h_addrtype;     /* host address type */
        int         h_length;       /* length of address */
        char        **h_addr_list;  /* list of address from name server */
#define h_addr      h_addr_list[0]  /* address, for backward compatibility */
};
```

The members of this structure are as follows:

h_name A pointer to the official name of the host.

h_aliases A pointer to a null-terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always AF_INET.

h_length The length, in bytes, of the address.

h_addr_list A pointer to the list of network address from the name server. Host addresses are returned in network byte order.

The **h_addr_list** is a new member of the **hostent** structure. It was needed because of widespread use of the domain name system (DNS), where one host may have a number of addresses. Applications coded before widespread use of the DNS used **h_addr** as a member of **hostent**. That member no longer exists, but is present as a macro for backward source level compatibility.

When you invoke the **getservbyname**(3N) routine, you pass it a service name and a protocol name, although you can specify NULL as the protocol name. If you specify NULL as the protocol, the routine searches from the beginning of **/etc/services** until it finds a matching service name or port number, or until it encounters the end of the file. The routine maps the service name it finds to a **servent** structure, which is defined as follows:

```
struct servent {
        char      *s_name;      /* official protocol name */
        char      **s_aliases;  /* alias list */
        long      s_port;       /* port service resides at */
        char      *s_proto;     /* protocol to use */
};
```

The members of this structure are as follows:

s_name A pointer to the official name of the service.

s_aliases A pointer to a null-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order (see Appendix B for a description of byte order).

s_proto A pointer to the name of the protocol to use when contacting the service.

Communicating Through Sockets

Once a socket is opened and bound, it can communicate with another socket. How this happens depends on whether a process uses stream sockets or datagram sockets. Only sockets of the same type and protocol may be connected, and then only if the protocol is one that allows connections. A socket may be connected to at most one other socket.

The following sections generally describe the sequence of events when a client program and a server program communicate through a socket. For the details about how a connection is established through stream sockets, see Chapter 4. For the details about how a connection is established through datagram sockets, see Chapter 5.

How Clients and Servers Communicate Through Stream Sockets

Table 3-3 shows a typical sequence of events when a client and server communicate using stream sockets. Note that the procedures in the first column are invoked by the client (a user starts a client program) and the procedures in the second column are invoked by the server (the server starts at boot time).

Table 3-3 Client/Server Communication Through Stream Sockets

Client	Server
1.	**s2=socket(AF_INET,SOCK_STREAM,0)**
2.	**getservbyname(...)**
3.	**bind(s2,...)**
4.	**listen(s2,...)**
5. **s1=socket(AF_INET,SOCK_STREAM,0)**	
6. **gethostbyname(...)**	
7. **getservbyname(...)** 8. **connect(s1,...)**	
9.	**s3 = accept(s2,...)**
10.	**fork()**
11. **write(s1,...)**	
12.	**read(s3,...)**

Table 3-3 summarizes the following sequence of events.

1. The server begins its initialization process by opening a socket, which is known to the process by its descriptor, s2. Arguments to the **socket** call specify the socket domain (Internet), the type of socket (stream), and the protocol to use (TCP is the default protocol for stream-type sockets).

2. In the Internet domain, a number of well-defined services are associated with reserved port numbers. (For example, the FTP file transfer service reserves TCP port number 21.) Mappings between services and port numbers are specified in services database such as the Network Information Service (NIS) or **/etc/services**. The server uses **getservbyname** to read the services database; **getservbyname** takes the name of a service and a protocol name as input, and returns a **servent** structure that contains the reserved port number assigned to the service.

3. The server binds an Internet address (INADDR_ANY) and port number (the one returned by **getservbyname**) to the socket. The socket is identified to **bind** by its descriptor, s2.

4. The server issues a **listen** call on the socket that is bound to the reserved port. This call does two things: it tells the system that the socket will be listening for incoming requests for service through the reserved port, and it sets a limit on the number of such requests that can be enqueued at the socket at any time. (Requests that arrive when the queue is full are ignored.) The **listen** call is described in detail in Chapter 4. The server has now finished its initialization process; it is ready to accept requests for service from clients.

5. The client process begins its initialization process by opening a TCP stream socket, which is known to the process by its descriptor, **s1**.

6. The client uses **gethostbyname** to map the server's symbolic name to an Internet address, which **gethostbyname** returns in a **hostent** structure. For details about **gethostbyname**(3N), see "Using Network Library Routines" in this chapter and the manual page.

7. The client process uses **getservbyname** to map the service name to its reserved port number, just as the server did.

8. The client asks that the socket it opened (**s1**) be connected to *name*, which is the Internet address and port to which the server's socket, **s2**, is bound. The **connect** call implicitly binds the client socket (**s1**) to any locally available Internet address and port. The **connect** call is described in detail in Chapter 4.

9. The server process calls the **accept** routine, which will accept the connection request at the head of the listen queue. (If the listen queue is empty, **accept** will not return to the caller until a connection request arrives in the listen queue.) The **accept** routine returns the accepted request in the form of a socket descriptor, **s3**. This socket descriptor refers to a newly opened socket that has been connected to the client socket, **s1**, from which the connection was requested. The **accept** call can be issued at any time after the **listen** call has been issued; it does not have to be synchronized with an incoming connection request. The **accept** call is described in detail in the Chapter 4 of this manual.

10. Most servers are designed to execute a **fork** operation at this point, creating a child process. The server's child process inherits from its parent the socket that accepted the connection from the client, **s3**. The child services the request from the client at the other end of the socket. When service is complete, the child process exits and ceases to exist. Meanwhile, the parent is free to service other clients by executing more **accept** operations on socket **s2** and spawning other children to service the requests thus accepted. In short, the parent process runs in a tight loop that accepts client requests and spawns a child for each one; each child services the request for which it was opened and exits when it completes service.

11. The client writes data into the socket to the server's child. For more information about the **write** system call, see "Transferring Data" later in this chapter.

12. The server's child reads data out of the socket from the client. For more information about the **read** system call, see "Transferring Data" later in this chapter.

The relationship between stream sockets is like the one that occurs between the two parties in a telephone call: once the connection has been set up, both parties can send and receive data at will without thinking about how information actually reaches the other party.

How Clients and Servers Communicate Through Datagram Sockets

Table 3-4 shows how a typical client/server might begin communication with datagram sockets. that the procedures in the first column are invoked by the client (a user starts a client program) and the procedures in the second column are invoked by the server (the server starts at boot time).

Table 3-4 Client/Server Communication Through Datagram Sockets

Client	Server
1.	**getservbyname(...)**
2.	**s2= socket(AF_INET,SOCK_DGRAM,0)**
3.	**bind(s2,...)**
4. **gethostbyname(...)**	
5. **getservbyname(...)**	
6. **s1= socket(AF_INET,SOCK_DGRAM,0)**	
7. **bind(s1,...)**	
8. **sendto(s1,...)** or **sendmsg(s1,...)**	
9.	**recvfrom(s2,...)** or **recvmsg(s2,...)**

Table 3-4 summarizes the following sequence of events:

1. In the Internet domain, a number of well-defined services are associated with reserved port numbers. (For example, TFTP reserves UDP port number 69.) Mappings between services and port numbers are specified in the file **/etc/services**. A server begins its initialization process by using **getservbyname** to read **/etc/services**; **getservbyname** takes the name of a service and a protocol name as input and returns a **servent** structure that contains the reserved port number assigned to the service.

2. The server opens a socket that is known to the process by its descriptor, **s2**. Arguments to the **socket** call specify the socket domain (Internet), the type of socket (datagram), and the protocol to use (UDP, which is the default protocol for datagram-type sockets).

3. The server binds an Internet address (INADDR_ANY) and port number (the one returned by **getservbyname**) to the socket. The socket is identified to **bind** by its descriptor, **s2**. The server has now completed its initialization process; it is ready to process requests for service from clients.

4. The client process begins its initialization process by using **gethostbyname** to map the server's symbolic name to an Internet address, which **gethostbyname** returns in a *hostent* structure. For details about **gethostbyname**(3N), see the manual page and Appendix B of this manual.

5. The client process uses **getservbyname** to map the service name to its reserved port number, just as the server did.

6. The client opens a UDP datagram socket, which is known to the process by its descriptor, **s1**.

7. The client performs a **bind** operation to assign the socket descriptor, **s1**, to any available local Internet address and port number.

8. The client uses the **sendto** or **sendmsg** call to send a datagram from its socket, **s1**, to the server's socket, **s2**. For more information about the **sendto** and **sendmsg** system calls, see "Transferring Data" later in this chapter.

9. The server process uses the **recvfrom** or **recvmsg** call to receive a datagram from its socket, **s2**. Each datagram contains not only data, but also the address of the socket from which it was sent. For more information about the **recvfrom** and **recvmsg** system calls, see "Transferring Data" later in this chapter.

Client/server communication through datagram sockets differs from its stream socket analog in some obvious ways: there are no **listen**, **accept**, or **connect** operations. The relationship between datagram sockets is like an exchange of letters through the mail: each datagram contains the address to which it is being sent, the address of its sender (like the return address on a letter), and data. When a server process receives a datagram, it acts on the data in the datagram and prepares a response datagram that contains not only data but also the address of the client as it appeared in the original datagram.

Transferring Data

Five pairs of system calls let you send and receive data: **write**(2) and **read**(2); **writev**(2) and **readv**(2); **send**(2) and **recv**(2); **sendto**(2) and **recvfrom**(2); and **sendmsg**(2) and **recvmsg**(2). Typically, you use **read** and **write**; **readv** and **writev**; and **send** and **recv** after you use a **connect** call. Conversely, you use **sendto** and **recvfrom**; and **sendmsg** and **recvmsg** with a connectionless protocol such as UDP.

The following sections describe how to use these pairs of system calls. They tell when it would be useful to use one pair instead of another.

Using the write and read System Calls

Whenever it is appropriate, sockets behave like UNIX files or devices, so you can use traditional operations like **write** and **read** with them. Figure 3-3 shows the syntax of the **write** and **read** system calls.

```
int socket_des;
char buf[128];
...
n = write(socket_des, buf, sizeof (buf));
n = read(socket_des, buf, sizeof (buf));
```

Figure 3-3 *Syntax of the write and read System Calls*

Using the writev and readv System Calls

Use the **writev** and **readv** system calls when you want a process to write or read a message without copying it into contiguous bytes. These two system calls use the **iovec** structure, which contains a sequence of pointers to blocks of memory from which the data should be read or into which the data should be stored.

The **iovec** structure looks like this:

```
struct iovec    {
      caddr_t          iov_base;
      int              iov_len;
};
```

The members of this structure are as follows:

iov_base Pointer to the base address of an area in memory where data should be placed.

iov_len The length of an area in memory where data should be placed.

Figure 3-4 shows a program fragment that illustrates the syntax of the the **writev** call.

```
#include<sys/types.h>
#include<sys/uio.h>

int sock_desc;  /* Socket descriptor */
int retval;     /* Return value from system call */
int iovcnt;     /* Number of elements in the scatter/gather array */

/* Declare the data */
char out_str1[] = "now is the time";
char out_str2[] = "for all good persons";
char out_str3[] = "to come to the aid";
char out_str4[] = "of their planet";

/* Declare the scatter/gather array */
struct iovec out_vector[4] =
    {
    {out_str1, sizeof(out_str1)},
    {out_str2, sizeof(out_str2)},
    {out_str3, sizeof(out_str3)},
    {out_str4, sizeof(out_str4)}
    };

/* Assume socket is open and connected to peer */

/* Compute the number of elements in the scatter/gather array */
iovcnt = sizeof(out_vector)/sizeof(struct iovec);

/* Write contents of scatter/gather array to previously opened socket */
retval = writev(sock_desc, out_vector, iovcnt);
```

Figure 3-4 *Syntax of the writev System Call*

Figure 3-5 shows a program fragment that illustrates the syntax of the the **readv** call.

```
#include<sys/types.h>
#include<sys/uio.h>

#define RECORDSIZE 255

int fildes;                              /* Socket descriptor */
int retval;                              /* Return value from system call */

/* Declare the buffers for the incoming data */
char in_str1[RECORDSIZE];
char in_str2[RECORDSIZE];
char in_str3[RECORDSIZE];
char in_str4[RECORDSIZE];

/* Build the scatter/gather array that references the buffers */
struct iovec in_vector[4] =
    {
    {in_str1, sizeof(in_str1)},
    {in_str2, sizeof(in_str2)},
    {in_str3, sizeof(in_str3)},
    {in_str4, sizeof(in_str4)}
    };

/* Compute the number of elements in the scatter/gather array */
iovcnt = sizeof(in_vector)/sizeof(struct iovec);

/* Read contents of scatter/gather array from previously opened socket */
retval = readv(fildes, in_vector, iovcnt);
```

Figure 3-5 *Syntax of the readv System Call*

Using the send and recv System Calls

The **send** and **recv** system calls are similar to **read** and **write**, except that they offer an extra *flag* argument, through which you can pass special options to manipulate data.

Figure 3-6 shows a program fragment that illustrates the syntax of the **send** call.

```
int i;                    /* Loop counter */
int sock_desc;            /* Socket descriptor */
int retcode;              /* Return code for system calls */
char sendbuf[ BUFSIZE ];  /* Data buffer */
int bufsize;              /* Bytes of data in buffer */
.
.
.
/* Assume that socket has already been opened and connected to peer */

/* Fill send buffer with binary data */
bufsize = 0;
for ( i = 0; i < BUFSIZE; i++ )
      {
      sendbuf[i] = (unsigned char) i;
      bufsize++;
      }
retcode = send(sock_desc, sendbuf, bufsize, 0);
```

Figure 3-6 *Syntax of the send System Call*

The first argument, **sock_desc**, is the descriptor for the opened and connected socket. The second argument, **sendbuf**, specifies a data buffer through which information is sent. The third argument, **bufsize**, specifies the size of the send buffer. The *flag* argument passed is 0; no flags were used.

Figure 3-7 shows a program fragment that illustrates the syntax of the **recv** call.

```
#define BUFSIZ    1024

int sock_desc;                  /* Socket descriptor */
int bytes_recv;                 /* Value returned from recv system call */
char recvbuffer[ BUFSIZ ];      /* Array to hold receive buffer */
.
.
.
/* Assume that socket is already opened and connected to peer */

/* Read the server socket for the data */
bytes_recv = recv(sock_desc, recvbuffer, sizeof(recvbuffer), 0);
```

Figure 3-7 *Syntax of the recv System Call*

In this fragment, the size of the buffer is calculated in the call (**sizeof(recvbuffer)**).

This fragment, like the fragment before it, passed **0** as the *flag* argument; no special options were desired. Here is a list of the flags that you can use with **send** or **recv**:

MSG_OOB Send and receive urgent (out-of-band) data. Urgent data is a feature specific to stream sockets. For more information on urgent data, see the section "Introduction to Urgent Data" in Chapter 4.

MSG_DONTROUTE Send data without routing packets. This option is currently used only by the routing table management process. The casual user is not usually interested in this option.

MSG_PEEK Look at data without reading. Users may want to preview data. When **MSG_PEEK** is specified with a **recv** call, any data present is returned to the user. The data, however, is treated as unread. Subsequent **recv** calls using this flag yield the same data. The next **read** or **recv** call applied to the socket will return the data previously previewed.

These options may be combined with the OR function. Here is an example:

```
n = recv(sock_desc, recvbuf, sizeof(recvbuf), (MSG_OOB | MSG_PEEK));
```

Using the sendto and recvfrom System Calls

Use the **sendto** and **recvfrom** system calls to send a message through unconnected sockets. Figure 3-8 shows the syntax for the **sendto** system call.

```
#include <sys/socket.h>
int retcode, socket_des;
char buf[128];
struct sockaddr_in to;
int flags;
...
retcode = sendto(socket_des, buf, sizeof(buf), flags, &to, sizeof(to));
```

Figure 3-8 *Syntax of the sendto System Call*

Use the **socket_des**, **buf**, and *flags* parameters the same way as for the **send** and **recv** system calls. The **sizeof(buf)** value provides the size of the buffer. The **to** value provides the address of the intended recipient, while the **sizeof(to)** value provides the size of the address.

When using an unreliable datagram interface, it is unlikely any errors will be reported to the sender. However, if the local system recognizes undelivered messages, **sendto** returns -1 and the global variable **errno** will contain an error number.

To receive messages on an unconnected datagram socket, use the **recvfrom** system call. Figure 3-9 shows its syntax.

```
#include <sys/socket.h>
int socket_des, retcode;
char buf[128]
struct sockaddr_in from;
int fromlen, flags;
...
fromlen = sizeof (from);
retcode = recvfrom(socket_des, buf, sizeof(buf), flags, &from, &fromlen);
```

Figure 3-9 *Syntax of the recvfrom System Call*

The **fromlen** parameter is handled in a value-result fashion, initially containing the size of the **from** buffer and changed upon return to reflect the size of the **sockaddr_in** structure returned from the source of the datagram.

For more information about how to use the **sendto** and **recvfrom** system calls and for sample programs that employ them, see Chapter 5.

Using the sendmsg and recvmsg System Calls

Use the **sendmsg** and **recvmsg**(2) calls when a long list of arguments required for **sendto** or **recvfrom** makes the program inefficient or hard to read. Instead of a long list of arguments, these system calls use the **msghdr** structure, which allows access to non-contiguous buffers. The **msghdr** structure looks like this:

```
struct msghdr   {
      struct sockaddr *         msg_name;
      int                       msg_namelen;
      struct iovec *            msg_iov;
      int                       msg_iovlen;
      caddr_t                   msg_accrights;
      int                       msg_accrightslen;
};
```

The members of this structure are as follows:

msg_name Pointer to the address associated with the message. If you use **sendmsg**, this is the address of the origin of the message. If you use **recvmsg**, this is the address of the destination of the message.

msg_namelen The size of the address.

msg_iov A pointer to a structure of type **iovec**. For details about **iovec** structures, see "Using the writev and readv System Calls."

msg_iovlen The number of elements in **msg_iov**.

msg_accrights The access rights sent and received. (This field is ignored for Internet domain sockets.)

msg_accrightslen The length of **msg_accrights**. (This field is ignored for Internet domain sockets.)

The program fragment in Figure 3-10 uses the **sendmsg** system call.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<errno.h>

int socket_des;                  /* Socket descriptor */
int retval;                      /* Return value */
struct sockaddr_in data_addr;

char out_str1[] = "now is the time";
char out_str2[] = "for all good persons";
char out_str3[] = "to come to the aid";
char out_str4[] = "of their planet";

struct iovec out_vector[4] =
    {
    {out_str1, sizeof(out_str1)},
    {out_str2, sizeof(out_str2)},
    {out_str3, sizeof(out_str3)},
    {out_str4, sizeof(out_str4)}
    };

struct msghdr out_header =
    {
    (struct sockaddr *) &data_addr,
    sizeof (struct sockaddr_in),
    out_vector,
    sizeof(out_vector) / sizeof (struct iovec),
    (caddr_t) NULL,
    0
    };

/* Assume that socket has already been opened and connected to peer */

retval = sendmsg (socket_des, &out_header, 0);
```

Figure 3-10 *Syntax of the sendmsg System Call*

The last argument of the **sendmsg** call is *flags*. In the previous example, no flags were passed.

Figure 3-11 shows a program fragment that uses the **recvmsg** system call.

```
#include<stdio.h>
#include<sys/types.h>
#include<sys/socket.h>
#include<sys/time.h>
#include<netinet/in.h>
#include<arpa/inet.h>
#include<errno.h>

#define BUFSIZ 512
int socket_des;                 /* Socket descriptor */
int retval;                     /* Return value */
struct sockaddr_in data_addr;

char in_str1[BUFSIZ];
char in_str2[BUFSIZ];
char in_str3[BUFSIZ];
char in_str4[BUFSIZ];

struct iovec in_vector[4] =
    {
    {in_str1, sizeof(in_str1)},
    {in_str2, sizeof(in_str2)},
    {in_str3, sizeof(in_str3)},
    {in_str4, sizeof(in_str4)}
    };

struct sockaddr_in from_addr;
struct msghdr in_header =
    {
    (struct sockaddr *) &from_addr,
    sizeof (struct sockaddr_in),
    in_vector,
    sizeof(in_vector) / sizeof (struct iovec),
    (caddr_t) NULL,
    0
    };

/* Assume that socket has already been opened and connected to peer */

retval = recvmsg (sock_desc, &in_header, 0);
printf ("received (length = %d): \n", retval);
printf ("    %s\n    %s\n    %s\n    %s\n",
        in_str1, in_str2, in_str3, in_str4);
```

Figure 3-11 *Syntax of the recvmsg System Call*

As with **sendmsg**, the last argument of the **recvmsg** call is *flags*. In the previous example, no flags were passed.

Setting and Reading Socket Options

Sockets have options that can be adjusted after the socket has been opened. These options can be set with the **setsockopt**(2) system call and read with the **getsockopt**(2) call. Socket options are interpreted at different levels of the protocol stack (for example the socket level or the transport level). The level at which the option should be interpreted is set through an argument in the **setsockopt** call.

Figure 3-12 shows the syntax of the **setsockopt** and **getsockopt** system calls. For more information about these system calls, see the appropriate manual page.

```
#include <sys/socket.h>
int socket_des setval getval;
int level;
int optname;
char *optval;
int optlen;
int *optleng;

setval = setsockopt (socket_des, level, optname, optval, optlen)

getval = getsockopt (socket_des, level, optname, optval, optleng)
```

Figure 3-12 *Syntax of the setsockopt and getsockopt System Calls*

The header file **/usr/include/sys/socket.h** contains definitions for socket level options. The argument **socket_des** is a socket that has been opened with the **socket** call. To manipulate options at the socket level, specify the *level* as **SOL_SOCKET**. Table 3-5 describes the options that you can specify for *optname*. To manipulate options at other levels, you must specify other values for level; these alternatives are described in later chapters.

For **setsockopt**, use *optval* and *optlen* to access option values. For **getsockopt**, use *optval* and *optlen* to identify a buffer in which the value for the requested options are to be returned. Specifically, *optlen* is a value-result parameter that initially contains the size of the buffer pointed to by *optval* and that is changed on return to indicate the actual size of the value returned. The *optname* parameter and any specified options are passed to the appropriate protocol module, where it gets interpreted.

The following socket options are recognized at the socket level. Except as noted, you can set and examine each with the **setsockopt** and **getsockopt** system calls.

NOTE: Default values are supplied, but a well-written program always sets the value of all the socket options it uses.

SO_LINGER *optval* is a pointer to **struct linger**.

optlen is **sizeof (struct linger)**.

Default *optval* is **l_onoff** = 0.

Controls the action taken when unsent data is queued on a TCP socket and a **close** call is performed. The **linger** structure consists of two fields: **int l_onoff** and **int l_linger**. When linger is set by making **l_onoff** in the **linger** structure true (nonzero), the **close** call will wait until all data has been delivered, or until the number of seconds specified by **l_linger** in the **linger** structure has been exceeded.

If **l_linger** is zero when **l_onoff** is true (nonzero), unsent data is discarded and the socket is immediately closed. If **l_onoff** is false (0), the socket will linger until all data is delivered, or until the connection with the peer is lost. If the size of the item pointed to by *optval* is not **sizeof(struct linger)** but like Berkeley 4.2 implementations, the BSD 4.2 semantics are used. These semantics make the *optval* a boolean switch. If the integer value pointed to by *optval* is false (0), the socket is closed immediately. If the value is true (nonzero), the socket lingers until all data is delivered, or until the connection with the peer is lost. The default action for **SO_LINGER** is to linger until all data is delivered, or the connection with the peer is lost.

SO_DEBUG

optval is a pointer to Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set (nonzero), causes the underlying protocol modules to collect debugging information. Currently, using this option has little effect on the underlying protocol implementation.

SO_KEEPALIVE

optval is a pointer to a Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set (nonzero), causes the periodic transmission of messages on a connected socket to ensure that the connection remains alive. Should the peer process fail to respond to the message within a reasonable time, the connection is considered broken. The user processes are notified with a SIGPIPE signal on the next socket write or by end of file on the next socket read.

SO_DONTROUTE

optval is a pointer to a Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set (nonzero), outgoing messages bypass the standard routing mechanisms and are delivered to the network specified indicated by the network portion of the destination address.

SO_BROADCAST *optval* is a pointer to a Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set (**optval* is nonzero) the associated socket is permitted to send broadcast datagrams on the socket. When reset/false (0), any attempt to send broadcast datagrams results in an **EPERM** error being returned. This option has meaning for datagram sockets only.

SO_REUSEADDR *optval* is a pointer to a Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set (*optval is nonzero), the rules for binding addresses are relaxed to allow local addresses to be reused. This puts the burden of ensuring socket uniqueness on the process issuing this socket option.

SO_OOBINLINE *optval* is a pointer to a Boolean flag of type **int**; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* = 0.

When true (*optval is nonzero), any out-of-band (urgent) data will be delivered to the process in sequence in the incoming data stream. Setting this option makes the out-of-band data accessible with **read** or **recv** calls without setting the **MSG_OOB** flag.

SO_SNDBUF *optval* is a pointer to an **int** variable that contains the size of the send buffer.

optlen is 4.

There is no default *optval*; the value depends on the protocol.

The integer value pointed to by *optval* is used as the maximum buffer capacity for outgoing traffic. The system may impose an arbitrary upper bound; it returns **ENOBUFS** when passed an out-of-range value.

SO_RCVBUF *optval* is a pointer to an **int** variable that contains the size of the receive buffer.

optlen is 4.

There is no default *optval*; the value depends on the protocol.

The integer value pointed to by optval is used as the maximum buffer capacity for incoming traffic. The system may impose an arbitrary upper bound; it returns **ENOBUFS** when passed an out-of-range value.

SO_TYPE *optval* is a pointer to an **int** variable that contains the socket type.

optlen is 4.

There is no default *optval*; the value depends on socket type.

Used only with **getsockopt**(2) to return the type of socket, such as **SOCK_STREAM**.

SO_ERROR *optval* is a pointer to an **int** variable that contains the error number.

optlen is 4.

There is no default *optval*; the value is set to the current socket error.

Used only with **getsockopt**(2) to return any pending error on the socket and to clear the error status.

The following example shows how to use the **getsockopt** system call to obtain the size of the send buffer.

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>

int sock_desc;              /* Socket descriptor */
int retval;                 /* Return value from getsockopt system call */
char buf[ BUFFERSIZE ];     /* Array to contain send buffer */
int bufsize;                /* Number of elements in send buffer */

/* Assume that socket is already open */

bufsize = sizeof(buf);

/* Get size of send buffer at the socket level with the SO_SNDBUF option */
retval = getsockopt(sock_desc, SOL_SOCKET, SO_SNDBUF, buf, &bufsize);
if( -1 == retval )
    {
    fprintf("Cannot get socket option SO_SNDBUF errno %d",errno);
    exit(1);
    }
```

Using the ioctl System Call

The **ioctl**(2) system call performs a variety of information requests and control operations on communications interfaces, protocol drivers, and other system-level entities. Figure 3-13 shows the syntax of the **ioctl**(2) call.

```
#include <sys/ioctl.h>

int des;
int command;
char *arg;
int ioctl(des, command, arg)
```

Figure 3-13 *Syntax of the ioctl System Call*

The **des** is a valid, active descriptor. Specify a valid control value for for the *command* argument. How you declare the *arg* depends on the *command* that you pass; the specifics are covered later in the section.

When you use the **ioctl** call on a socket, there are three kinds of device control commands that you may specify for the *command* argument: those that you can use with terminal devices, sockets, and regular files; those that you can use with Internet sockets; and those that you can use with sockets. Table 3-5 lists the device control commands that you can use with terminal devices, sockets, and regular files.

Table 3-5 ioctl Commands Used with Terminals, Sockets, and Files

Command	*arg* Type	Description
FIOASYNC	int *	Change the I/O mode of the socket. The default I/O mode for a socket is synchronous I/O mode. This means that no special notifications are made to the socket's process or process group when data is available to be read. Use asynchronous I/O mode when you want signals to indicate that data is available to be read from a socket. The process or process group to which the **SIGIO** signal is sent may be set by a previous invocation of the **ioctl** system call using the **SIOCSPGRP** command.
FIONBIO	int *	Set or clear nonblocking I/O mode on the socket. By default, nonblocking I/O mode is turned off. This means that all socket calls for the socket do not return to the user program until the actions that they perform are completed. When you choose nonblocking I/O mode, the socket system calls return as quickly as possible, and do not wait for the desired actions to complete. You can then use the **select** system call to determine status. The behavior of each socket system call when you use nonblocking I/O mode is described in its manual page.
FIONREAD	int *	Retrieve number of bytes of data available to be read on socket or other device associated with descriptor.

Table 3-6 lists the device control commands that apply only to Internet sockets.

Table 3-6 ioctl Commands that Apply Only to Internet Sockets

Command	*arg* Type	Description
SIOCATMARK	**int ***	Determine if TCP Urgent Data buffered in socket has been read by process.
SIOCGIFADDR	**struct ifreq ***	Get the Internet address associated with the network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**. It is described later in this section.
SIOCSIFADDR	**struct ifreq ***	Set the Internet address associated with the network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**. It is described later in this section.
SIOCGIFBRDADDR	**struct ifreq ***	Get the Internet broadcast address associated with the network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**. It is described later in this section.
SIOCSIFBRDADDR	**struct ifreq ***	Set the Internet broadcast address associated with the network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**. It is described later in this section.

(continued)

Table 3-6 ioctl Commands that Apply Only to Internet Sockets

Command	*arg* Type	Description
SIOCGIFCONF	**struct ifconf ***	Used to obtain information about all network interfaces in a system. It takes an **ifconf** structure (found in **<net/if.h>**) as its argument. The **ifconf** is described in detail later in this section. The structure contains a length field and a pointer to an array of **ifreq** structures. You should allocate an array of **ifreq** structures and set the **ifconf ifcu_req** field to point to the array you allocate. Set the **ifconf ifc_len** field to the byte length of the **ifreq** array. This command returns a name (**ifr_name** in the **ifreq** structure) and address (**ifr_ifru.ifru_addr** in the **ifreq** structure) for each network interface configured in the system specified in the **ifreq** structure. If an insufficient number of **ifreq** elements are supplied, all supplied elements are filled with interface information, and the **ifc_len** field is set to indicate that all available space was used. If there are enough elements to hold all of the network interface information, the information is returned and the **ifc_len** field is set to the amount of space actually used.
SIOCGIFDSTADDR	**struct ifreq ***	Get the Internet address associated with the remote host on the point-to-point network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**. Find out about a point-to-point interface by using **SIOCGIFFLAGS** (see below) and checking for **IFF_POINTOPOINT**. **SIOCGIFDSTADDR** on a non-point-to-point interface returns **EINVAL**.

(continued)

Table 3-6 ioctl Commands that Apply Only to Internet Sockets

Command	*arg* Type	Description
SIOCSIFDSTADDR	**struct ifreq ***	Set the Internet address associated with the remote host on the point-to-point network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.
SIOCGIFFLAGS	**struct ifreq ***	Get interface flags associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.
SIOCSIFFLAGS	**struct ifreq ***	Set interface flags associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.
SIOCGIFMETRIC	**struct ifreq ***	Get metric associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.
SIOCSIFMETRIC	**struct ifreq ***	Set metric associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.

(continued)

Table 3-6 ioctl Commands that Apply Only to Internet Sockets

Command	*arg* Type	Description
SIOCGIFNETMASK	**struct ifreq ***	Obtain Internet network mask associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.
SIOCSIFNETMASK	**struct ifreq ***	Set Internet network mask associated with specific network interface whose name is specified in the **ifr_name** field of the **ifreq** structure supplied to the call. The **ifreq** structure is defined in **<net/if.h>**.

(concluded)

Finally, Table 3-7 lists the device control commands that apply only to sockets.

Table 3-7 ioctl Commands that Apply Only to Sockets

Command	*arg* Type	Description
SIOCGPGRP	**int ***	Retrieve ID of process/process group that owns socket.
SIOCSPGRP	**int ***	Associate ownership of socket with specific process/process group. By default, the socket is not associated with the process ID or process group ID of the process that opened the socket.

As Table 3-6 indicates, any **ioctl** call that uses **SIOCGIFADDR, SIOCGIFDSTADDR, SIOCGIFBRDADDR, SIOCGIFNETMASK, SIOCGIFFLAGS,** or **SIOCGIFMETRIC** as the *command* argument requires *arg* to be declared as an **ifreq** structure, as follows:

```
struct ifreq *arg
```

Here is the definition of the **ifreq** structure:

```
typedef struct ifreq
        {
        char ifr_name [IFNAMSIZ];
        union
             {
             struct sockaddr ifru_addr;
             struct sockaddr ifru_dstaddr;
             struct sockaddr ifru_broadcast;
             struct sockaddr ifru_mask;
             unsigned short  ifru_flags;
             unsigned long   ifru_metric;
             unsigned long   ifru_data;
             } ifr_ifru;
         } inet_ifreq_type;
```

Table 3-8 describes the fields that make up the **ifreq** structure:

Table 3-8 Fields of the ifreq Structure

Name	Size	Offset	Description
ifr_name	16	0	interface name
ifr_ifru.ifru_addr.sa_family	2	16	address
ifr_ifru.ifru_addr.sa_data	14	18	data
ifr_ifru.ifru_dstaddr.sa_family	2	16	other end of p-to-p link
ifr_ifru.ifru_dstaddr.sa_data	14	18	data
ifr_ifru.ifru_broadaddr.sa_family	2	16	broadcast address
ifr_ifru.ifru_broadaddr.sa_data	14	18	data
ifr_ifru.ifru_flags	2	16	flags
ifr_ifru.ifru_metric	4	16	metric
ifr_ifru.ifru_data	4	16	for use by interface

The **ifr_name** field contains an interface name that identifies the network interface with which the **ifreq** structure is associated. The size of this field is the maximum size of an interface name. Interface names are null-terminated ASCII strings; the null byte is counted as part of the size of the name.

You can use several of the fields to provide information about a network interface that characterizes it on a network supporting IP. The **ifr_ifru.ifru_addr** structure specifies the Internet address. Use the **ifr_ifru.ifru_dstaddr** structure to provide information about the destination address associated with a network interface. This field has significance on those interfaces that provide a point-to-point interconnection. Use the **ifru_flags** field to provide information on the state of a network interface by means of interface flags.

Any **ioctl** call that uses **SIOCGIFCONF** as the **command** argument requires *arg* to be declared as follows:

```
struct ifconf *arg
```

Here is a definition of the **ifconf** structure:

```
typedef struct ifconf
      {
      unsigned long ifc_len;
      union
          {
          caddr_t         ifcu_buf;
          struct ifreq * ifcu_req;
          }
          ifc_ifcu;
      } inet_ifconf_type;
```

Table 3-9 describes the fields that make up the **ifconf** structure.

Table 3-9 Fields of the ifconf Structure

Name	Size	Offset	Description
ifc_len	4	0	Amount of space, in bytes, left unmodified in the array referenced by the **ifc_ifcu.ifcu_req** field.
ifc_icu.ifcu_buf	4	4	
ifc_ifcu.ifcu_req	4	4	References an array of **ifreq** structures. The **ioctl** call fills the array with information about each interface configured in the system.

The following example shows a program that uses **ioctl** on an **inen0** network interface to get the network mask. You can use a similar program to obtain information about any local network interface.

```
extern int errno;
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <net/if.h>
#include <netdb.h>
#include <stdio.h>

main( argc, argv )
                int argc;
                char *argv[];
{
                int s, ns, addrlen, i;
                char c, *cp;
                struct sockaddr_in addr_base;
                struct sockaddr_in *addr = &addr_base;
                struct ifreq ifr;
                struct sockaddr_in netmask;

        if( argc != 1 ) {
                fprintf( stderr, "Usage: serv \n" );
                exit (1);
        }

                addr_base.sin_port = 0;
                addr_base.sin_family = AF_INET;
                addr_base.sin_addr.s_addr = INADDR_ANY;

                s = socket( AF_INET, SOCK_STREAM, 0 );
                if( s == -1 ) {
                        fprintf( stderr, "open failed with errno %d\n", errno );
                        exit(1);
                }

                strcpy(ifr.ifr_name, "inen0");

                if (ioctl(s, SIOCGIFNETMASK, &ifr) == -1) {
                    perror ("ioctl failed");
                    exit (1);
                    }

                netmask = *((struct sockaddr_in *) &ifr.ifr_addr);
                fprintf( stderr, "netmask %x \n", netmask.sin_addr.s_addr );
                close( s );
}
```

For more information, see the **ioctl**(2) manual page.

Input/Output Multiplexing with the select System Call

The DG/UX system provides the ability to multiplex input/output requests among multiple sockets and files. Use the **select**(2) call to check for activity on several file descriptors at once. Figure 3-14 shows the syntax of the **select** system call.

```
int nfound, nfds, readfds, writefds, exceptfds;
struct timeval timeout;
...
nfound = select(nfds, &readfds, &writefds, &exceptfds, &timeout);
```

Figure 3-14 *Syntax of the select System Call*

The **select** call takes three bit masks as arguments. These bit masks are as follows:

readfds File descriptors for reading data.

writefds File descriptors to which data is written.

exceptfds File descriptors that have an exceptional condition pending. For sockets, this indicates the presence of urgent data in the stream.

Bit masks are used to store file descriptors. The **select** call watches these masks to determine whether or not there is data to be read, whether there is a file descriptor ready to accept data, or whether there are exceptions. Bit masks are created by or-ing bits of the form "1 << fildes". The form "1 << fildes" allows you to shift the bits in the mask to the left. That is, a descriptor **fildes** is selected if a 1 is present in the file descriptor bit of the mask. The parameter **nfds** specifies the range of file descriptors (that is, 1 plus the value of the largest descriptor) specified in a mask. The parameter **nfound** contains the total number of descriptors **select** returns.

The following example shows how the **select** call works.

```
#include <stdio.h>
#include <time.h>

int fdesc_1, fdesc_2; /* File descriptors from socket call */
int ibits;            /* I/O masks for the select call */
int nfds;             /* Number of file descriptors to check in select */

/* We have 2 file descriptors for which we are monitoring incoming traffic */
...

/* Get the maximum number for the descriptors */

nfds = max(fdesc_1, fdesc_2) +1;

/* Create the mask for all the descriptors */
/* to check for data on input */

/* RESTRICTION: the value of fdesc cannot be bigger than 31 */
ibits = 0;
ibits |= (1 << fdesc_1);
ibits |= (1 << fdesc_2);

if (select (nfds, &ibits, NULL, NULL, NULL) == -1) {
     perror("Error in select");
     exit(-1);
} else {
 if (ibits & (1 << fdesc_1)) {
     printf ("Data on the first socket \n");
     ...

   }
 if (ibits & (1 << fdesc_2)) {
     printf ("Data on the second socket \n");
     ...
   }
 }
...
```

You can specify a timeout value if you want to limit the amount of time to wait for activity. If **timeout** is set to 0, the selection takes the form of a poll, returning immediately. If the last parameter is a null pointer, the selection will block indefinitely. In this case, a return takes place only when a descriptor is selectable, or when the caller receives a signal that interrupts the system call. **select** normally returns the number of file descriptors selected. When the **select** call exceeds the timeout limit, a value of 0 is returned.

The **select** call provides a synchronous multiplexing scheme. You can have asynchronous notification of output completion, input availability, and exceptional conditions by using the SIGIO signal.

Closing Sockets

When a socket is no longer needed, you can discard it by applying a **close**(2) system call to the descriptor. Figure 3-15 shows the syntax of the **close** call.

```
int socket_des;
...
close(socket_des);
```

Figure 3-15 *Syntax of the close System Call*

If you use **close** on a socket that promises reliable delivery (a stream socket), the system continues to attempt to transfer undelivered data. The action of the **close** call depends on the state of the **SO_LINGER** option for that socket. If the option has not been previously set, the **close** call suspends until all data currently written to a socket is delivered or until the TCP connection is aborted. If the option has been set with a linger timeout of zero, then the TCP connection is aborted and the **close** call returns immediately. If the option has been set with a nonzero linger timeout, then all data previously written to the socket are delivered to the TCP peer subject to the timeout restriction. In this instance, **close** returns when all such data have been delivered or when the linger timeout period expires. If the linger timeout period expires, **close** returns a value of -1, and the errno value is **ETIMEDOUT**.

You can discard pending data prior to closing a socket with the **shutdown**(2) call, whose syntax is shown in Figure 3-16.

```
int socket_des, how;
...
shutdown(socket_des, how);
```

Figure 3-16 *Syntax of the shutdown System Call*

The value of *how* specifies the action to take: **0** terminates data reception at the socket, **1** terminates data transmission from the socket, and **2** terminates both transmission and reception.

If you use **shutdown** to terminate reception, all data waiting to be read from the socket is discarded. Any data that arrive later will also be discarded. If you issue a **read** or **recv** call after terminating reception, the call will return immediately with a transfer count of zero.

If you use **shutdown** to terminate transmission on a socket and subsequently try to **write** or **send** to that socket, the request will return the **EPIPE** error. Furthermore, if the socket has a connected peer, that peer will be told to expect no more incoming data, and all **read** or **recv** calls at the peer's end of the connection will return immediately with a transfer count of zero.

End of Chapter

Chapter 4
Programming With the Transmission Control Protocol

The previous chapter gave an overview to programming with sockets. This chapter covers topics specific to programming with stream type sockets that communicate through the Transmission Control Protocol (TCP). It describes the system calls that you use to establish a connection between stream sockets. It tells how to set socket options at the transport level. It discusses the notion of urgent data, and describes how you transmit and receive urgent data. It also provides an example of client and server programs using TCP.

For a thorough discussion of TCP, see *Internet Request for Comments (RFC) 793 (Transmission Control Protocol)*. Also, see *RFC 1122 (Requirements for Internet Hosts -- Communication Layers)* for requirements for host system implementations of TCP.

Establishing a Connection Through Stream Sockets

You know from reading earlier chapters or from experience that TCP provides reliable, stream-oriented, process-to-process service. You also know that the DG/UX system implements TCP as functions in the kernel. You use stream sockets to connect to and communicate with remote programs using TCP.

To initiate communication, your program must create a communication endpoint with the **socket**(2) call (see "Creating Sockets" in Chapter 3). If your program initiates a server process, it uses the **bind**(2) call to bind a name to the socket (see "Binding Sockets" in Chapter 3).

Establishing a connection between stream sockets usually involves one process acting as a client and the other as a server. The server, when willing to offer its advertised services, passively listens on its socket. It must be listening before the client tries to connect. The client requests services from the server by initiating a connection to the server's socket. The following sections focus on how this happens through stream sockets.

What Server Processes Do

Server processes are started at boot time and run in the background waiting for incoming connections. Each server has a well known port number. Clients use this port to access the server.

To establish a connection and begin accepting data, a server program does the following:

1. Uses **getservbyname**(3N) to look up its service definition. The service definition is a data structure containing the name of the service, an alias list, a port number, and the protocol the service uses (see Appendix A).

2. Uses **socket**(2) to create an interface for communicating across the network.

3. Uses **bind**(2) to associate a wildcard address to a socket.

4. Uses **listen**(2) to begin listening for incoming connections on the bound socket.

5. Uses **accept**(2) to accept an incoming connection.

6. Uses **fork**(2) to create a child process that uses the new socket returned by the **accept**(2) call.

7. As the child process begins processing data, the parent process calls **accept**(2) to service the next connection.

8. The server repeats steps 5 through 7 as needed.

The server's data structure in the service definition is used in later portions of the code to specify the port number on which the server listens for service requests. An example of the code follows:

```
struct servent *server_pointer;
char name[128];
char protocol[16];
...
server_pointer = getservbyname(name, protocol);
if (server_pointer == NULL) {
   fprintf(stderr,"service %s not supported \n", name);
   exit(1);
   }
```

The **getservbyname** call will fail if the service is not defined in **/etc/services** or in the Network Information Service (NIS).

Using the listen and accept System Calls

As pointed out earlier, after binding to its socket, a server must perform two steps to receive a client's connection: 1) listen for incoming requests and 2) accept a connection if it is requested. The server performs these steps with the system calls **listen**(2) and **accept**(2). In the Internet domain, you issue the **listen** call with the syntax shown in Figure 4-1.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket_des, backlog;
listen(socket_des, backlog);
```

Figure 4-1 *Syntax of the listen System Call*

The *backlog* specifies the maximum number of outstanding connections that can wait for acceptance from the server process. When the queue is full, messages requesting additional connections are ignored. As a result, a busy server has time to make room in the queue while the client retries the connection. To prevent processes from monopolizing system resources, the DG/UX system limits the **backlog** figure to no more than SOMAXCONN connections on any one queue. SOMAXCONN is defined in the **<sys/socket.h>** header file. The **listen** call does not block while waiting on a connection.

Once a server is listening, it can accept a connection. In the Internet domain, you issue the **accept** system call with the syntax shown in Figure 4-2.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket_des, snew_des, fromlen;
struct sockaddr_in from;
...
fromlen = sizeof(from);
snew_des = accept(socket_des, &from, &fromlen);
```

Figure 4-2 *Syntax of the accept System Call*

When the server accepts a connection, the **accept** call returns a new descriptor and a new socket. If you want to have the server identify its client, supply a buffer for the client socket's name through the **from** parameter. The value-result argument **fromlen** is initialized by the server to indicate how much space is associated with the **from** argument (the name of the client's socket). The **fromlen** argument is modified on return to reflect the true size of the name. If the client's name is not important, you can give the **from** and **fromlen** arguments a value of zero.

The **accept** call normally blocks. This means that the **accept** call does not return until a connection is available or until the system call is interrupted by a signal to the process. Furthermore, a process can indicate that it will accept connections from only a specific peer. For details about how it can do this, see the description of the socket option **TCP_PEER_ADDRESS** later in this chapter. The server process can accept successive connections from more than one client.

The following example uses the **listen** and **accept** system calls:

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

int sock_desc, new_sock;          /* Two socket descriptors */
int retcode;                      /* Return code from system calls */
struct servent *service;          /* Port number for server  */
struct sockaddr_in sname, cname;  /* Server and client socket names */
int cname_len = sizeof(cname);    /* Size of client socket name */

sock_desc = socket(AF_INET,SOCK_STREAM,0);  /* Open a socket */
if( -1 == sock_desc ) {
    fprintf(stderr,"Cannot open socket, errno %d\n",errno);
    exit(1);
    }

/* Get port number for the service we want from destination */
service = getservbyname("myservice", "tcp");
if ( NULL == service ) {
     fprintf(stderr, "Cannot get port number from getservbyname\n");
     exit(1);
     }

sname.sin_family = AF_INET;           /* Set up socket name in the Internet domain */
sname.sin_port = service->s_port;     /* Put port number for service in socket name */
sname.sin_addr.s_addr = INADDR_ANY;   /* Use wildcard address for socket name */

/* Bind the socket for the connection */
retcode = bind(sock_desc, &sname, sizeof(sname));
if( -1 == retcode ) {
    fprintf(stderr,"Cannot bind, errno %d\n", errno);
    exit(1);
    }

/* Get into the listen state */
retcode = listen(sock_desc, 1);
if( 0 != retcode ) {
    fprintf(stderr,"Cannot set socket to listen state, errno %d\n", errno);
    exit(1);
    }

/* Wait for a connection and return the first connection on the queue */
new_sock = accept(sock_desc, &cname, &cname_len);
if( -1 != retcode ) {
    fprintf(stderr," Error in accept, errno %d\n",errno);
    exit(1);
    }
```

What Client Processes Do

When you invoke a client program, the client process usually initiates a connection by doing the following:

1. Locating the service definition (port number of service) with the **getservbyname**(3N) call (for details, see "Using the Network Library Routines" in Chapter 3).

2. Looking up the destination host with the **gethostbyname**(3N) call (again, see "Using the Network Library Routines" in Chapter 3).

3. Create a socket for communicating across the network with the **socket** system call.

4. Optionally, associate a name to the socket with the **bind** system call. If a name is not associated with **bind**, TCP will choose and attach one when the **connect** call is invoked.

5. Attempt to connect to the server with the **connect** system call.

Binding a Stream Socket to an Unspecified Port

When a client program uses the **bind** call to associate a name to a socket, it can leave the local port unspecified (specified as zero). The DG/UX system will select an appropriate port number for it. Here is an example.

```
address.sin_addr.s_addr = MYADDRESS;
address.sin_port = 0;
bind(socket_des, &address, sizeof (address));
```

If you specify the local port, the value must meet two criteria: (1) it cannot be a port numbered 1 through 1024 (these are reserved for the superuser), and (2) it cannot be already bound to a socket, unless you apply the **SO_REUSEADDR** socket option to the socket being bound. If you apply this option to a newly created socket, then you can bind the new socket to a port number already bound to another socket. You cannot, however, use the new socket in any way that could create ambiguity. The new socket cannot be connected to the same remote address as the first socket, nor can two sockets with the same port number both listen for incoming connections.

The following example shows how you can set the **SO_REUSEADDR** option with the **setsockopt**(2) call.

```
#include <sys/types.h>
#include <sys/socket.h>
int socket_des;
struct sockaddr_in address;
...
setsockopt(socket_des, SOL_SOCKET, SO_REUSEADDR, (char*)0, 0);
bind(socket_des, &address, sizeof (address));
```

Using the connect System Call

In the Internet domain, you issue the **connect** call with the syntax shown in Figure 4-3.

```
#include <sys/socket.h>
int socket_des namelen;
struct sockaddr_in *name;

retcode = connect (socket_des, &name, namelen);
```

Figure 4-3 *Syntax of the connect System Call*

where **socket_des** is the descriptor of a socket that you create (on the client side of the connection) where you want datagrams sent, *name* is the name of the peer socket (on the server side of the connection) from where datagrams will arrive, and *namelen* is the length of the *name* in bytes.

You may use the **connect** call as illustrated in the following example.

```
#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

int sock_desc;                  /* Socket descriptor */
int retcode;                    /* Return code from connect system call */
char *dest_name;                /* Pointer to remote hostname */
struct sockaddr_in name;        /* Socket name for peer */
struct hostent *dest_addr;      /* Address for destination host*/
struct servent *service;        /* Port number for server  */
struct sockaddr_in to_addr;     /* Socket name for destination peer */

/* Open a socket */
sock_desc = socket(AF_INET,SOCK_STREAM,0);
if( -1 == sock_desc )
   {
    fprintf(stderr,"Cannot open socket, errno %d\n", errno);
    exit(1);
   }

/* Get port number for the service we want from destination */
service = getservbyname("myservice", "tcp");
if ( NULL == service )
    {
    fprintf(stderr, "Cannot get port number from getservbyname\n");
    exit(1);
    }

/* Get the internet address for the destination host */
dest_addr = gethostbyname(dest_name);
if ( NULL == dest_addr )
    {
    fprintf(stderr, "Cannot get destination address %s \n", dest_name);
    }

/* Set up the sockaddr structure with info about peer */
to_addr.sin_family = dest_addr->h_addrtype;
to_addr.sin_addr.s_addr = *( (int *) dest_addr->h_addr ) ;
to_addr.sin_port = service->s_port;

/* Try to connect to server */
retcode = connect(sock_desc, &to_addr, sizeof(to_addr));
```

The Internet address and port number of the server to which the client process wishes to speak gets assigned to the second argument of the **connect** call. If the client process's socket is unbound at the time of the **connect** call, the system automatically selects and binds a name to the socket. As pointed out earlier, this is how local addresses are usually bound to a client socket.

If the **connect** call fails for any reason, it returns an error code. If the **connect** fails, you must close the socket on which the error occurred and create a new socket with the **socket** call before you reattempt the connection. If there is no error, the socket is connected to the server, and data transfer may begin.

Setting and Reading Socket Options at the Transport Level

Recall from the previous chapter that sockets have options that can be adjusted after the socket has been created, and that these options can be set and read with the **setsockopt**(2) and **getsockopt**(2) system calls. Here again is the synopsis of these two system calls.

```
#include <sys/socket.h>
int socket_des setval getval;
int level;
int optname;
char *optval;
int optlen;
int *optleng;
setval = setsockopt (socket_des, level, optname, optval, optlen)

getval = getsockopt (socket_des, level, optname, optval, optleng)
```

To manipulate options at the transport level for TCP, specify the *level* as **IPPROTO_TCP**. Here are the valid options for TCP at the transport level.

TCP_NODELAY

optval is an int variable; nonzero = set/true, 0 = reset/false.

optlen is 4.

Default *optval* is 0.

When set, the system does not delay sending data to coalesce small packets. When the option is reset, the system may defer sending data to coalesce small packets to conserve network bandwidth.

TCP_MAXSEG

optval is an int variable.

optlen is 4.

There is no default *optval*; value is negotiated by TCP. This is a read-only option.

When set, all TCP SYN segments from the TCP endpoint include a TCP Maximum Segment Size (MSS) option. Values for the TCP Maximum Segment Size are between 0 and 65,535.

TCP_URGENT_INLINE

optval is an int variable; nonzero = set/true, 0 = reset/false.

optlen is 4

This has no effect in the DG/UX system. Use **SO_OOBINLINE** at the socket level.

TCP_PEER_ADDRESS

optval is a pointer to **struct sockaddr_in**.

optlen is size of **struct sockaddr_in**.

Default *optval* is **INADDR_ANY**.

When set, restricts the **listen** system call to allowing only those connections initiated by the supplied address. Used when a process wants to accept a connection from a single, specific remote host. Only one remote address may be specified; subsequent invocations of the option will override previous address settings.

Introduction to Urgent Data

You can think of a connection between two stream sockets as a pair of queues or data streams, one for data transmission in each direction. The TCP protocol provides a way to mark a sequence of bytes in such a data stream as urgent data. (Often, the terms "urgent data" and "out-of-band data" are used interchangeably.) Only one urgent message at a time can exist in a data stream.

The protocol driver that handles transmissions from a given socket can quickly notify the protocol driver that controls the connected receiving socket that urgent data is on the way before such data arrives at the receiving socket. The receiving socket's protocol driver, in turn, sends a **SIGURG** signal to the process or process group that owns the receiving socket. The signal notifies that process or process group that urgent data is being sent.

If there is urgent data in a stream, a special pointer called the "urgent mark" points to the last byte of urgent data in the stream. The TCP protocol specification stipulates that the first byte in an urgent-data sequence must be self-identifying; that is, given the end of an urgent message, the program that receives the message must be able to discern where it begins.

Transmitting and Receiving Urgent Data

A program can send urgent data with a **send** call or a **sendto** call with the *flags* field set to MSG_OOB. The urgent mark will point to the last byte in the send buffer.

To receive urgent data, use a **recv** call or a **recvfrom** call with the *flags* field set to MSG_OOB. If such a call is made when urgent data is neither available at the socket nor on the way to the socket, the error value EINVAL is returned.

It is possible to peek at received urgent data using a **recv** call or a **recvfrom** call with the *flags* field set to (MSG_OOB | MSG_PEEK). For more information about these flags, see "Using the send and recv System Calls" in Chapter 3.

If a process group owns the socket, a **SIGURG** signal is generated when the receiver's protocol driver is notified by its peer that urgent data is being sent. A process can use the **fcntl**(2) call with the **F_SETOWN** command argument to set the process group ID or process ID that will receive **SIGURG** signals from a socket.

If multiple sockets have urgent data awaiting delivery, a programmer may use a **select** call for exceptional conditions to determine which sockets have such data pending. Neither the signal nor the select indicate the actual arrival of the urgent data. Instead, the signal or select indicates that the urgent data has been sent by the remote peer. In other words, the **SIGURG** may be dispatched before the urgent data arrives at the socket. For more information about the **select** system call, see "Input/Output Multiplexing" in Chapter 3.

In addition to the urgent message itself, the logical mark called the "urgent mark" is placed in the data stream to indicate the end of the most recent urgent message. The remote login and remote shell applications use this facility to propagate signals between client and server processes. When a signal flushes any pending output from the remote process or processes, all data in the stream up to the urgent mark are discarded.

To find out if the read pointer is currently pointing at the urgent mark in the data stream, the **SIOCATMARK ioctl** is provided. Here is an example of how the **SIOCATMARK ioctl** may be used.

```
ioctl(s, SIOCATMARK, &yes);
```

Here, if **yes** returns a nonzero value, the next read returns data after the urgent mark. Otherwise (assuming urgent data has arrived), the next read provides data sent by the client prior to transmission of the urgent signal.

Receiving Out-of-line and In-line Data

A program receives urgent data one of two ways: either out-of-line (independently of normal data) or in-line (inserted in the normal data stream). To choose between out-of-line and in-line reception, set the socket-level option **SO_OOBINLINE** through the **setsockopt** system call; see **setsockopt**(2) for usage. If not explicitly set, **SO_OOBINLINE** defaults to the reset state, making out-of-line urgent reception the default.

When a program uses out-of-line urgent reception, the TCP protocol driver takes special steps to separate the urgent byte (the last byte in the most recently arrived urgent-data message) from the normal data stream. First, the urgent pointer, which always points to a byte in the normal data stream, is set to point to the byte that follows the urgent byte in the data stream. Then, the urgent byte is removed from the normal data stream and is stored in a special one-byte system buffer that is reserved for urgent data. (The most recently arrived urgent byte overwrites any datum that may already be in the urgent buffer.) Once the urgent byte is stored, a **recv** or **recvmsg** call with the **MSG_OOB** flag set will return the single byte of urgent data to the caller. Normal read-type operations (**read** calls and **recv** and **recvmsg** calls with the **MSG_OOB** flag reset) will return data from the normal data stream.

When a program uses in-line urgent reception, the TCP protocol driver leaves the urgent byte in the normal data stream and sets the urgent pointer to point to the urgent byte. If the urgent byte is not at the head of the stream, normal read-type operations (**read** calls and **recv** and **recvmsg** calls with the **MSG_OOB** flag reset) will return only data that precedes the urgent byte in the stream. If the urgent byte is at the head of the stream, normal read operations operate exactly as they would if there were no urgent data in the stream. For in-line urgent reception, urgent read operations (**recv** and **recvmsg** calls with the **MSG_OOB** flag set) can be thought of as variations of the normal read operations. If the urgent byte is not at the head of the stream, urgent read-type operations will return data up to and including the urgent byte. If the urgent byte is at the head of the stream, urgent read operations will return just the urgent byte. If there is no urgent byte in the stream, urgent read operations will return **EINVAL**.

Figure 4-4 shows the routine used in the remote login process to flush output on receipt of an interrupt or quit signal. This code reads the normal data up to the urgent mark (to discard it), and then reads the urgent byte. The code fragment, taken from the remote login program, provides an example of in-line urgent data reception.

```
#include <sys/ioctl.h>
#include <sys/file.h>
...
oob()
{
        int out = FWRITE;
        char waste[BUFSIZ];
        int mark;

        /* flush local terminal output */
        ioctl(1, TIOCFLUSH, (char *)&out);
        for (;;) {
                if (ioctl(rem, SIOCATMARK, &mark) < 0) {
                        perror("ioctl");
                        break;
                }
                if (mark)
                        break;
                (void) read(rem, waste, sizeof waste);
        }
        if (recv(rem, &mark, 1, MSG_OOB) < 0) {
                perror("recv");
                ...
        }
        ...
}
```

Figure 4-4 *Receiving In-Line Urgent Data*

A process may also read or peek at the urgent data without first reading up to the urgent mark. This is more difficult when the underlying protocol delivers the urgent data in line with the normal data but sends a **SIGURG** before the urgent data actually arrives at the socket. If the urgent byte has not yet arrived when a **recv** is done with the **MSG_OOB** flag set, the call returns an error of **EAGAIN**. Worse, there may be enough non-urgent data at the head of the input buffer that normal flow control prevents the peer from sending the urgent data until the buffer is cleared. In these circumstances, the process must then read enough of the enqueued data that the urgent data may be delivered.

Certain programs that use multiple bytes of urgent data and that must handle multiple urgent signals (for example **telnet**) expect each urgent message to retain its position within the data stream. Such programs set the **SO_OOBINLINE** option. Having set the option, a program can read all data up to the urgent mark using a **recv** call with the *flags* field set to zero, then read the data at the urgent mark with another **recv** with the *flags* field set to **MSG_OOB**. Reception of multiple urgent messages causes the mark to move, but no urgent data are lost.

A program would use the **SIOCATMARK** argument to the **ioctl** call in conjunction with the **recv** call to check the position of the read pointer. When a user process receives a **SIGURG** signal indicating that urgent data has arrived at the socket, the process's signal handler positions the socket's read pointer at the urgent mark. The signal handler repeatedly reads in a buffer of data from the socket and calls the **ioctl** call with **SIOCATMARK** to see if the read pointer is at the urgent mark. When the **ioctl** call returns a value of true in its return parameter, then the byte that the urgent mark points to can be read using the **recv** call. The following example shows how to use the **ioctl** call with **SIOCATMARK**.

```
#include <sys/ioctl.h>
int socket_des, is_at_mark;
...
ioctl(socket_des, SIOCATMARK, &is_at_mark);
```

Understanding the Subtleties of Urgent-Data Reception

With in-line urgent-data reception, if urgent data is in the socket data queue, a read operation without the **MSG_OOB** flag set reads at most the sequence of bytes that precedes the urgent byte in the queue. A read operation with the **MSG_OOB** flag set reads at most the sequence of bytes at the head of the queue that terminates with the urgent byte.

Also with in-line reception, the parameter returned by the **SIOCATMARK** ioctl is nonzero if and only if the next byte to be read from the socket is the urgent byte. With out-of-line reception, the parameter returned by the **SIOCATMARK** ioctl is nonzero if and only if the next byte to be read from the socket is the one that immediately followed the urgent byte before the urgent byte was pulled out of line.

With in-line reception, if a read operation with the **MSG_OOB** flag set returns **EAGAIN**, the urgent byte may simply not have arrived at the socket yet. On the other hand, the **EAGAIN** error may be a sign that there is so much data in the stream, that the stream is fully congested, and the urgent byte is trapped somewhere between the remote peer and the receiving socket. In this case, the receiving program must continue to read data from the socket until the urgent byte arrives.

With out-of-line reception, if an urgent byte arrives at the socket before a previous urgent byte has been read, the first urgent byte is lost. In contrast, with in-line reception, if an urgent byte arrives before a previous urgent byte has been read, the urgent pointer is simply moved downstream to point at the most recently arrived urgent byte. Programs must be constructed so as not to be confused by this asynchronous arrival of urgent messages.

A program that illustrates this phenomenon (albeit one that is not too useful) would invoke the **SIOCATMARK ioctl** twice without an intervening read operation. If urgent data is waiting at the head of the queue before the first invocation, and a second urgent message arrives between the first and second invocations, the first call would return TRUE, but the second would return FALSE.

Using the SIGURG Signal and Process Groups

Each process has an associated process group. **SIGURG** is sent to all processes within a process group. **SIGURG** signals are initialized to the process group of their creator, but can be redefined at a later time by using the **ioctl**(2) call with **SIOCSPGRP** as an argument. The following example illustrates such a call.

```
#include <sys/socket_ioctl.h>
int socket_des, pgrp;
ioctl(socket_des, SIOCSPGRP, &pgrp);
```

Use **SIOCGPGRP** with the **ioctl** call to determine the current process group of a socket.

Some Sample Programs

This section contains two programs, one client and one server, both written in C. The programs use TCP to access a remote service.

The programs, **serv.c** and **client.c**, are designed to show how to use the system calls to create a socket, establish a connection, and send messages back and forth across the connection. **serv.c** accepts any number of letters and converts them to uppercase. **client.c** takes any number of letters from a local terminal, sends them to the server for conversion, reads the response from the server, and displays the response to a terminal. The client program terminates when it reads a % from **serv.c**.

Both programs use **gethostbyname** to return a *hostent* structure for mapping the host address to the hostname supplied. If the system is running NFS, the entry may come from the NIS database. If the system is running DNS, the entry may come from a name server.

Both programs also use **getservbyname** to request the service name to the port number. The service specifications are in **/etc/services**. The client and server programs use **getservbyname** to map their names to a port number. If your system is running NFS, this entry may be in the Network Information Service (NIS). For more information, see Appendix B.

The client.c Program

This program requests a connection to the server named by its second argument, receives alphabetic characters from a terminal and sends them to the remote **serv.c** process.

```
extern int errno;
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>

main( argc, argv)
     int argc;
     char * argv[];
{

     int s, len;
     char c;
     struct sockaddr_in addr_base;
     struct sockaddr_in *addr = &addr_base;
     struct hostent * hp;
     struct servent *sp;
     if( argc != 2 ) {
          fprintf( stderr, "Usage:\t client hostname \n" );
          exit (1);
     }
     hp = gethostbyname( argv[1], NULL );

     if ( hp == NULL ) {
          fprintf( stderr, "No host named %s\n", argv[1] );
          exit( 1 );
     }
     addr_base.sin_family = hp->h_addrtype ;
     addr_base.sin_addr.S_un.S_addr = *((int *) hp->h_addr);
/*
 *    Service "tcp_example" must be in the /etc/services file
 *    with a unique port number.
 */

     sp = getservbyname( "tcp_example", NULL );
     if( sp == NULL ) {
          fprintf( stderr, "Can't find tcp_example in /etc/services \n");
          exit( 1 );
     }
     addr_base.sin_port = sp->s_port;
     s = socket( AF_INET, SOCK_STREAM, 0 );
     if( s == -1 ) {
          fprintf( stderr, "create failed with errno %d\n", errno );
          exit(1);
     }
/*
 *    Implicit bind takes place.
 */

     if( connect( s, addr, sizeof(struct sockaddr_in) ) == -1 ) {
          fprintf( stderr, "connect failed with errno %d\n", errno );
          exit(1);
     }
```

```
        do {
             c = getchar();
             if( c == '\n' ) {
                  continue;
             }
             if( write( s, &c, 1 ) != 1 ) {
                  fprintf( stderr, "client write failed\n" );
                  break;
             }
             if( read( s, &c, 1 ) != 1 ) {
                  fprintf( stderr, "client read failed\n" );
                  break;
             }
             putchar( c );
             putchar( '\n' );
        } while( c != '%' );
}
```

The serv.c Program

This program listens for requests for its service from a remote process, establishes a connection, accepts alphabetic characters from **client.c** and converts them to uppercase.

```
extern int errno;
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <stdio.h>

main( argc, argv )
     int argc;
     char *argv[];
{
     int s, ns, addrlen, i;
     char c, *cp;
     struct sockaddr_in addr_base;
     struct sockaddr_in *addr = &addr_base;
     struct servent *sp;


     if( argc != 1 ) {
          fprintf( stderr, "Usage: serv \n" );
          exit (1);
     }
/*
*     Service "tcp_example" must be in the /etc/services file
*     with a unique port number.
*/

     sp = getservbyname( "tcp_example", NULL );
     if( sp == NULL ) {
          fprintf( stderr, "Service tcp_example not in /etc/services \n");
          exit( 1 );
     }
     addr_base.sin_port = sp->s_port;
     addr_base.sin_family = AF_INET;
     addr_base.sin_addr.s_addr = INADDR_ANY;
```

```
        s = socket( AF_INET, SOCK_STREAM, 0 );
        if( s == -1 ) {
             fprintf( stderr, "create failed with errno %d\n", errno );
             exit(1);
        }

        if( bind( s, addr, sizeof(struct sockaddr_in) ) == -1 ) {
             fprintf( stderr, "bind failed with errno %d\n", errno );
             exit(1);
        }

        if( listen( s, 3 ) == -1 ) {
             fprintf( stderr, "listen failed with errno %d\n", errno );
             exit(1);
        }
        printf( "The server is up.  Place it in the background.\n" );
        for( ;; ) {
             addrlen = sizeof( struct sockaddr_in );
             ns = accept( s, addr, &addrlen );
             if( ns == -1 ) {
                  fprintf(stderr, "accept failed with errno %d\n", errno);
                  continue;
             } else {
                  fprintf( stderr, "Connection accepted from " );
                  cp = (char *) addr;
                  for( i = 0; i < 16; i++ ) {
                       fprintf( stderr, "%d ", *cp++ );
                  }
                  fprintf( stderr, "\n" );
             }
             do {
                  if ( read( ns, &c, 1 ) != 1 ) {
                       fprintf( stderr, "Broken read with errno %d\n", errno);
                       break;
                  }

                  if ( 'a' <= c && c <= 'z' ) {
                       c += 'A' - 'a';
                  }

                  if ( write( ns, &c, 1 ) != 1 ) {
                       fprintf( stderr, "Broken write with errno %d\n", errno);
                       break;
                  }
             } while( c != '%' );
             fprintf( stderr, "server done\n" );
             close( ns );
        }
   }
```

End of Chapter

Chapter 5
Programming with the User Datagram Protocol

This chapter discusses how to program with datagram sockets to access the User Datagram Protocol (UDP). It describes the system calls you use to communicate through datagram sockets. It also includes an example of a client and a server program.

For a thorough discussion of UDP, see *Internet Request for Comments (RFC) 768 (User Datagram Protocol)*. Also, see *RFC 1122 (Requirements for Internet Hosts -- Communication Layers)* for requirements for host system implementations of UDP.

Communicating Through Datagram Sockets

Sockets created in UDP require no connections. Even so, they are created much the same way as sockets in TCP. Once created through the **socket** call and associated with a port number through the **bind** call, a UDP socket is ready for use by a program.

System calls associated with connection-oriented sockets, such as **listen**(2) and **accept**(2) are not allowed. Instead, a program uses the **sendto**(2) and **recvfrom**(2) system calls to send and receive data between peer processes. Arguments to these calls provide the destination address for the data.

Figure 5-1 illustrates the syntax of the **sendto** call and Figure 5-2 illustrates the syntax of the **recvfrom** call.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

int sock_desc;              /* Socket descriptor */
int retcode;                /* Return code from system calls */
char buf[ BUFSIZE ];        /* Data buffer */
int buflen;                 /* Number of elements in data buffer */
char *dest_name;            /* Pointer to remote hostname */
struct hostent *dest_addr;  /* Address for destination host */
struct servent *service;    /* Port number for server  */
struct sockaddr_in to_addr; /* Socket name for the destination peer */

sock_desc = socket(AF_INET,SOCK_DGRAM,0); /* Open a socket */
if( -1 == sock_desc ) {
    fprintf(stderr,"Cannot open socket, errno %d\n", errno);
    exit(1);
   }

/* Get port number for service from destination */
service = getservbyname("myservice", "udp");
if ( NULL == service ) {
    fprintf(stderr, "Cannot get port number from getservbyname\n");
    exit(1);
    }

/* Get Internet address for destination host */
dest_addr = gethostbyname(dest_name);
if ( NULL == dest_addr ) {
    fprintf(stderr, "Cannot get destination address %s \n", dest_name);
    }

/* Setup the sockaddr structure with info about peer */
to_addr.sin_family = dest_addr->h_addrtype;
to_addr.sin_addr.s_addr = *( (int *) host->h_addr ) ;
to_addr.sin_port = service->s_port;

/* Put data in buffer (buf) and compute length of buffer (buflen) */
.
.

retcode = sendto(sock_desc, buf, buflen, 0, &to_addr, sizeof(to_addr)); /* Send the data */
if ( -1 == retcode ) {
   fprintf(stderr, "Error in send %d \n", errno );
   }
else {
   printf ("Sent %d bytes\n", retcode);
   }
```

Figure 5-1 *Syntax of the sendto System Call*

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>
#include <memory.h>
#include <netdb.h>

int sock_desc;                  /* Socket descriptor*/
int retval;                     /* Return value from system calls */
char buf[ BUFSIZE ];            /* Data buffer */
struct sockaddr_in from_addr; /* Socket name for the destination peer */
int from_addr_len;              /* Size of socket name */
struct servent *service;
struct sockaddr_in bind_addr;

/* Open a socket */
sock_desc = socket(AF_INET,SOCK_DGRAM,0);
if( -1 == sock_desc )
    {
    fprintf(stderr,"Cannot open socket, errno %d\n", errno);
    exit(1);
    }

service = getservbyname("myservice", "udp");

if ( NULL == service )
    {
    fprintf(stderr, "Cannot get port number from getservbyname\n");
    exit(1);
    }

memset((caddr_t)&bind_addr, 0, sizeof(struct sockaddr_in));

bind_addr.sin_family = AF_INET;
bind_addr.sin_port = service->s_port;

result = bind(sock_desc,(struct sockaddr *)&bind_addr,sizeof(struct sockaddr));
if (result == -1)
    {
    fprintf(stderr, "bind failed\n");
    exit(1);
    }

/* Compute the length of buffer for the socket name of the destination peer */
from_addr_len = sizeof(from_addr);


/* Read the data from peer */
retval = recvfrom(sock_desc, buf, sizeof(buf), 0, &from_addr, &from_addr_len);
if ( -1 == retval )
    {
    fprintf(stderr, "Error in recvfrom %d \n", errno );
    }
else
    {
    printf("Read %d bytes\n", retval);
    }
```

Figure 5-2 *Syntax of the recvfrom System Call*

For both calls, the fourth argument is a *flags* argument, through which you can pass special options to manipulate data. The two program fragments above passed **0** as the *flags* argument; no special options were desired.

Using the connect System Call with Datagram Sockets

Datagram sockets can also use the **connect**(2) call to associate a socket with a specific address. Data sent on the socket is automatically addressed to the connected peer. Only one connected address is honored for each socket (that is, no multicasting).

When you use the **connect**(2) call on datagram sockets, requests return immediately because the system has only to record the peer's address. These requests do not call the peer and establish a connection.

Connected UDP sockets allow you to use the **write/read**, **writev/readv**, and **send/recv** system calls to transfer data.

Broadcasting and Datagram Sockets

Datagram sockets can be used to send broadcast packets on those types of networks that can broadcast, such as Ethernet networks. Broadcast messages can place a high load on a network since they force every host on the network to service them.

There are two ways to send broadcast packets: become the superuser or, as an ordinary user, specify the socket option (**SO_BROADCAST**). To send a broadcast message, you must follow these steps:

1. Create an Internet datagram socket.
2. Bind at least a port number to the socket (you can bind the host number as well, which specifies a complete Internet address).
3. Determine Internet broadcast address.
4. Address the message.
5. Issue a **sendto** call.

Figure 5-3 shows a code fragment that sends a broadcast message.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>

/* Must be superuser to execute code */

int socket_des, cc;
char buf[128];
int buflen;
struct sockaddr_in dst, sin;
struct ifreq  ifr;

/* Fill in buf and buflen here */

socket_des = socket(AF_INET, SOCK_DGRAM, 0);

sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;

bind(socket_des, (char *)&sin, sizeof (sin));
strcpy(ifr.ifr_name,"inen0");
ioctl(socket_des, SIOCGIFBRDADDR, (caddr_t) &ifr);

dst.sin_family = AF_INET;
dst.sin_addr.s_addr = ((struct sockaddr_in *) &ifr.ifr_broadcast)->
                        sin_addr.s_addr;
dst.sin_port = DESTPORT;

cc = sendto(socket_des, buf, sizeof(buf), 0, &dst, sizeof (dst));
```

Figure 5-3 *Sending a Broadcast Message*

Received broadcast messages contain the sender's address and port (datagram sockets must be bound to an address before a message is allowed to go out). In this example, the device is set to inen0. Normally, you would use **SIOCGIFCONF ioctl** to get a list of interfaces, and you would check whether any given interface is capable of broadcasting.

Some Sample Programs

This section contains two programs, **are_you_there.c** and **i_am_here.c**, both written in C. The programs use UDP to access a remote service.

The server program, **i_am_here.c**, is written to be put in the background. After you invoke it, the server will accept a message from any client sending one to its port and running in the Internet domain. When the server receives a message, it sends a message to the client process indicating that it is alive.

The client program, **are_you_there.c**, binds to any address, sends a message to the server program **i_am_here.c**, and waits for a reply. When it receives a reply, the client program prints a message to the screen indicating that the server is running.

Both the client and server programs use the **gethostbyname** routine to return a **hostent** structure for mapping the hostname supplied on the command line to the host address. If your system is running NFS, this entry may be in an NIS database. If your system is running the domain name system, the entry may come from a name server. For more information, see Appendix B.

Both the client and server programs also use **getservbyname** to request the service specification indicated in **/etc/services**. If your system is running NFS, this entry may be in an NIS database. For more information, see Appendix B.

The are_you_there.c Program

This is a client program that sends a message to a server program to see whether that server is running. If this program receives a response, it prints a message to the terminal indicating that the server is running.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <netinet/in.h>
#include <stdio.h>
#include <signal.h>

extern int errno;
void alarmed();
main ( argc, argv )
     int argc;
     char *argv[];
{
     int s, ns, i, cc, flags, fromlen;
     char c, *cp, buf[1024], msg[17];
     char *name = "tcp_example";
     struct sockaddr_in addr_base;
     struct sockaddr_in *addr= &addr_base;
     struct sockaddr_in from;
     struct sockaddr_in to;
     struct servent *sp;
     struct hostent *hp;

     strcpy(&msg[0],"Hello!");
```

```
     if ( argc != 2 ) {
          fprintf ( stderr, "Usage:\t are_you_there hostname \n" );
          exit (1);
     }

     hp = gethostbyname ( argv[1], NULL );
     if ( hp == NULL ) {
          printf ( "no host named %s\n", argv[1] );
          exit (1);
     }
/*
*Service name must be in /etc/services.
*/

     sp = getservbyname ( name, NULL );
     if ( sp == NULL ) {
          printf ( "no known service named %s\n", name );
          exit (1);
     }
/*
*     We are a client, so we ask to be bound to any port
*     ( addr_base.sin_port = 0 ).  We don't care what port
*     we are on, only what port the server is on.  Bind
*     will assign us a port.
*/

     bzero( (char *)addr, sizeof(struct sockaddr_in) );
          addr_base.sin_family = hp->h_addrtype;

     s = socket ( AF_INET, SOCK_DGRAM, 0 );
     if ( s == -1 ) {
          perror( "socket" );
          exit (1);
     }

     if ( bind ( s, addr, sizeof(struct sockaddr_in) ) == -1 ) {
          perror( "bind" );
          exit(1);
     }
/*
*     Next we want to send a message to the server.  The
*     theory is that he will send a message back and we'll
*     know that he is alive.
*
*     In preparation for this, we'll zero out the "to" struct.
*     This makes sure that there is no garbage to interfere
*     with our call.  We set up the "to" struct with the
*     correct family and the port number and address
*     of the server.  We got the port number from the service
*     name and the address from the hostname.  Both
*     of these parameters came from the command line.
*/

     flags = 0;

     bzero( (char *)&to, sizeof(to) );
     to.sin_family = AF_INET;                /* Assign family */
     to.sin_port = sp->s_port;               /* Assign port */
     to.sin_addr.s_addr = *(int *)hp->h_addr; /* Specify address of server */

     cc = sendto ( s, msg, sizeof(msg), flags, &to, sizeof(to) );
     if ( cc == -1 ) {
          perror( "sendto" );
          exit(1);
     }
```

```
/*
*     Finally, we'll wait for the server to return our call.
*
*     Once again we start by zeroing out the 'from' structure
*     and setting 'fromlen' to the length of that struct.  Then
*     we let recvfrom() do the rest.
*/

     printf( "waiting for response from %s\n", hp->h_name );
     signal( SIGALRM, alarmed );
     alarm( 5 );

     bzero( (char *)&from, sizeof(from) );
     fromlen = sizeof(from);
     cc = recvfrom ( s, buf, sizeof(buf), flags, &from, &fromlen );
     if ( cc == -1 ) {
          perror( "recvfrom" );
          exit(1);
     }

     printf ( "Other machine is alive and kicking \n" );
}

void     alarmed()
{
     printf( "Whoops - no answer!\n" );
     exit(0);
}
```

The i_am_here.c Program

This is a server program that receives messages from a remote client process and sends a message to that process, informing the client that it is up and running.

```
#include <sys/types.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/socket.h>
#include <stdio.h>

extern int errno;
main ( argc, argv )
     int argc;
     char *argv[];
{
     int s, ns, buflen, i, cc, flags, *fromlen, tolen;
     char c, *cp, buf[1024], *msg;
     char *name = "tcp_example";
     char hostname[14];
     struct sockaddr_in addr_base;
     struct sockaddr_in *addr;
     struct sockaddr_in *from;
     struct sockaddr_in *to;
     struct servent *sp;
     struct hostent *hp;
     struct sockaddr_in from_container;
     int fromlen_container;
```

```
     if ( argc != 1 ) {
          fprintf ( stderr, "usage: \t i_am_here \n" );
          exit (1);
     }

     if( -1 == gethostname(hostname,sizeof(hostname))){
          printf("Can't gethostname() errno %d",errno);
          exit(1);
     }

     hp = gethostbyname ( hostname, NULL );
     if ( hp == NULL ) {
          printf ( "can't find host %s\n", hostname );
          exit (1);
     }
/*
*Service name must be in /etc/services.
*/

     sp = getservbyname ( name, NULL );
     if ( sp == NULL ) {
          printf ( "can't find %s\n", name );
          exit (1);
     }

     addr_base.sin_port = sp->s_port;
     addr_base.sin_family = AF_INET;
     addr_base.sin_addr.s_addr = INADDR_ANY;
     s = socket ( AF_INET, SOCK_DGRAM,0 );
     if ( s == -1 ) {
          fprintf ( stderr, "create failed with errno %d\n", errno );
          exit(1);
     }

     addr = &addr_base;
     if ( bind ( s, addr, sizeof (struct sockaddr_in) ) == -1 ) {
          fprintf ( stderr, "bind failed with errno %d \n", errno );
          exit (1);
     }
     from = &from_container;
     fromlen_container = sizeof (from_container);
     fromlen = &fromlen_container;
     buflen = sizeof(buf);
     for ( ;; ) {
          cc = recvfrom ( s, buf, buflen, flags, from, fromlen );

          if ( cc == -1 ) {
               fprintf ( stderr, "receive failed with errno %d\n", errno );
               exit (1);
          }
          cc = sendto ( s, buf, buflen, flags, from, *fromlen );
          if ( cc == -1 ) {
               fprintf ( stderr, "send failed %d\n", errno );
               exit (1);
          }
     }

}
```

End of Chapter

Chapter 6
Programming with the Internet Protocol and Internet Control Message Protocol

This chapter discusses programming with the Internet Protocol (IP) and the Internet Control Message Protocol (ICMP). It describes why and how to use raw sockets. It tells how to set and read socket options at the IP level. It also includes a sample program that uses raw sockets to communicate with other hosts.

For a thorough discussion of IP, see *Internet Request for Comments (RFC) 791 (Internet Protocol)*. Also, see *RFC 1122 (Requirements for Internet Hosts -- Communication Layers)* for requirements for host system implementations of IP. For a thorough discussion of ICMP, see *RFC 792 (Internet Control Message Protocol)*.

Programming at this level is not for inexperienced programmers. Access to this level of programming is limited to superusers, typically system programmers. System programmers use this level to write programs that use IP to create and experiment with new protocols. Programmers can also use this level to gain access to facilities provided by the Internet Control Message Protocol (ICMP). ICMP provides error reporting, an echoing facility, and access to gateways.

Creating Raw Sockets for the Internet Protocol

As with programming at the TCP or UDP level, you use sockets to access the IP level. As with TCP and UDP, you create sockets in IP through the **socket**(2) system call. To program at the IP level, you must use the Internet domain and raw socket type. The following example shows how to create a raw socket.

```
int socket_des;
socket_des = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);
```

where **AF_INET** represents the Internet domain, **SOCK_RAW** represents the raw socket type, and **IPPROTO_RAW** represents the specific protocol in the domain specified.

IP delivers datagrams to sockets based on the following:

- If incoming datagrams are addressed to an existing kernel level protocol (that is, TCP or UDP), they are given to the specified protocol.

- If incoming datagrams are not addressed to a specified protocol, they are given to all raw sockets that can accept them.

Typically you use the **sendto/recvfrom** or **sendmsg/recvmsg** system calls to transfer data through raw sockets. If you use the **connect** system call to specify an address for a raw socket, then you may use the **send/recv** system calls to transfer data.

Communicating Through IP

Table 6-1 shows how communication might begin using IP and ICMP. Note that the procedures for the **bind** and **connect** calls are optional. Data can be sent without binding or connecting the processes to specific addresses.

Table 6-1 How Communication Begins with IP and ICMP

Calling Process	Action
gethostbyname(...)	The calling process uses this library routine to turn the name of the foreign host into a host address.
gethostname(...)	The calling process uses this call to find the name of the local host and look up the Internet address corresponding to that name. This address is used for the **bind** call.
s1=socket(AF_INET,SOCK_RAW,0)	The calling process creates a raw socket. The **socket** call creates an endpoint for communicating between the processes. Arguments to the call specify the socket domain (Internet), type of socket (raw), and the protocol to use (IP is the default).
bind(s1,...)	The calling process binds its socket to an address on the local host. Arguments to the call specify the socket, name to be bound to the socket, and the length of the name (in bytes).
Build an IP header.	The calling process builds an IP header to accompany the data transferred. IP headers may contain IP options. For more information, see "Specifying an IP Header."
sendto(s1,...)	The calling process sends a datagram that includes the IP header and data. Arguments to the **sendto** call specify the socket to which to send the message, the message buffer, the length of the message (in bytes), the flags to use when sending the message, the name of the destination, and the length of the destination name (in bytes).
recvfrom(s1,...)	Arguments to the **recvfrom** call specify the socket to receive the message from, the buffer of the message, the length of the buffer, the flags for transfer, the structure to hold the sender's name, and the number of bytes returned.

Setting and Reading Socket Options at the IP Level

At the IP level, sockets have options that can be adjusted after the socket has been created. These options can be set and read with the **setsockopt**(2) and **getsockopt**(2) system calls. Here is the synopsis of these two system calls.

```
int socket_des setval getval;
int level;
int optname;
char *optval;
int optlen;
int * optleng;
setval = setsockopt (socket_des, level, optname, optval, optlen)

getval = getsockopt (socket_des, level, optname, optval, optleng)
```

To manipulate options for IP, specify the *level* as **IPPROTO_IP**.

Here are the valid socket options for IP.

IP_TX_OPTIONS *optval* is a pointer to the option string.

optlen is the size of the option string.

Default *optval* is NULL.

Allows all IP-specific options to be set in one option management call for outgoing datagrams. The following option strings are recognized:

IPOPT_EOL (End of option list);
IPOPT_NOP (No operation);
IPOPT_LSRR (Loose source and record route);
IPOPT_SSRR (Strict source and record route);
IPOPT_TS (Internet timestamp); and
IPOPT_SECURITY (Security – some environments may require this).

IP_RX_OPTIONS *optval* is a pointer to the option string.

optlen is the size of the option string.

There is no default *optval*; the value depends on the options received in the last IP packet for the endpoint.

Gives an application the ability to obtain the IP options in incoming datagrams. Recognizes the same option strings as **IP_TX_OPTIONS**.

IP_TOS

optval is an **int** variable.

optlen is 4.

Default *optval* is 0.

Specifies the Type of Service field for all subsequent IP transmissions from the socket. The leftmost three bits of the most significant byte of the option value (bits 7-5) indicate the minimum acceptable IP precedence level for the transport endpoint. The next leftmost bit of the option value (bit 4) specifies the Delay characteristic for all subsequent IP transmissions associated with the transport endpoint. The next leftmost bit of the option value (bit 3) specifies the Throughput characteristic for all subsequent IP transmissions associated with the transport endpoint. The next leftmost bit (bit 2) specifies the Reliability characteristic for all subsequent IP transmissions associated with the endpoint. The rightmost two bits (bits 0-1) are reserved.

IP_TTL

optval is an **int** variable.

optlen is 4.

There is no default *optval*; the value depends on the protocol.

Specifies the Time to Live field for all subsequent IP transmissions from the socket.

IP_DONTFRAG

optval is an **int** variable.

optlen is 4.

Default *optval* is 0 (fragmentation is allowed).

Prohibits fragmentation of IP datagrams.

Introduction to IP Message Formats

A typical IP message contains an IP header and the data to be transferred. The data section can be further divided when you use other protocols with IP, such as ICMP. In this case, the data section contains an ICMP header and the data to be transferred.

Specifying an IP Header

IP only sends data from host to host. IP sends the necessary address information along with the data. This information is included in a header. When programming with IP, you must provide information for the IP header.

TCP/IP for AViiON Systems supports IP options. These options would follow the destination address field on the header. Adding options will affect the value of the IHL field. Figure 6-1 shows the header format.

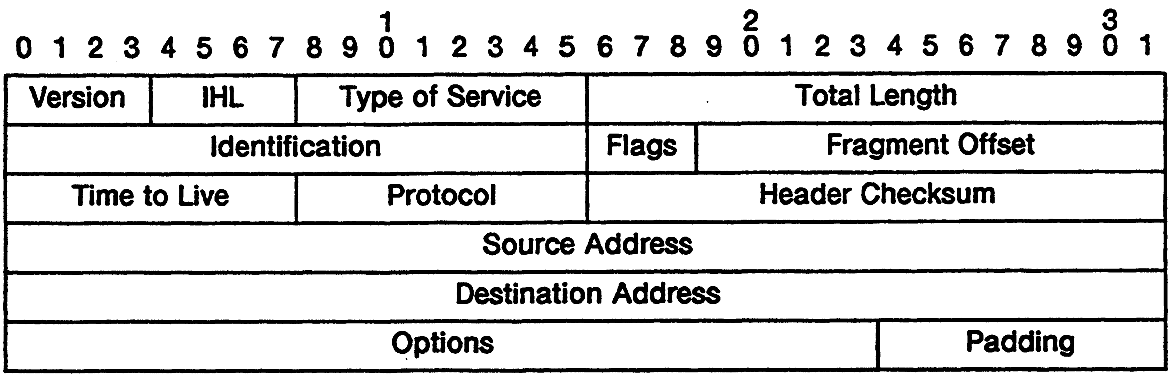


Figure 6-1 *A Sample Internet Datagram Header*

Table 6-2 describes each element in the header and provides the number of bits in the element's field.

Table 6-2 Elements in an Internet Datagram Header

Element	Description	Bits
Version	Indicates the version of the Internet header your system is running. Current Internet version is 4.	4
IHL	Internet Header Length (IHL). Indicates the length of the header in 32-bit words. The minimum value for a correct header is 5. If this value is more than 5, everything after the fifth word will be options (see MIL-STD-1777).	4
Type of Service	Specifies service parameters to use when transmitting a datagram through a particular network. Use a 0 or see MIL-STD-1777.	8
Total length	Indicates the total length of a datagram measured in octets (8-bit quantities), including the Internet header and data. This field allows the length of a datagram to be as many as 65,535 octets.	16
Identification	Indicates the value assigned by the sending process that helps in assembling fragments. This field is used internally; set it to 0.	16
Flags	Indicates whether fragmenting is allowed. If fragmentation is allowed, this element indicates whether more fragments exist. Bit 0 is reserved. Bit 1 controls whether the datagram can split into fragments (0 indicates the datagram can be fragmented, 1 indicates it cannot). Bit 2 indicates whether the fragment received is the last one (0 indicates this is the last fragment, 1 indicates more fragments exist).	3
Fragment Offset	Indicates where each fragment belongs in the datagram. Fragment offset is measured in units of 8 octets (64 bits). The first fragment has offset 0. Use 0 in this field.	13

(continued)

Table 6-2 Elements in an Internet Datagram Header

Element	Description	Bits
Time to Live	Specifies the maximum number of hops that a datagram is allowed to travel in the Internet system before it is destroyed. If this field contains the value 0, the datagram is destroyed. This field is decremented by at least 1 whenever the Internet header is processed. Set this field to at least one greater than the number of gateways through which the datagram travels.	8
Protocols	Indicates the protocol above IP (for example, ICMP=1). See the file **/usr/include/netinet/in.h.**	8
Header Checksum	Checks the header for errors at each point that the Internet header is processed. The algorithm is the 16-bit one's complement of the one's complement sum of all 16-bit words in the header. Clear this field before calculating checksum.	16
Source Address	Indicates the Internet address of the sender.	32
Destination Address	Indicates the Internet address of the intended receiver.	32

(concluded)

Specifying an IP Header When Using ICMP

When specifying an IP header for an ICMP message, you must fill in values for the fields. To fill in these values, create a header based on the information given earlier in this chapter in Table 6-2 but set Protocol to the constant **IPPROTO_ICMP**, defined in the file **/usr/include/netinet/in.h**, and Header Checksum to 0.

Introduction to ICMP Message Formats

ICMP is used for reporting errors to hosts. The ICMP messages provide feedback about problems in the communication environment. These messages can be sent when a datagram cannot reach its destination, the gateway does not have the buffering capacity to forward a datagram, or the gateway can direct the host to send traffic through a shorter route. A *gateway* is an intermediate host that allows other hosts that do not have direct connections to communicate through a system of interconnected networks.

When programming with ICMP, use the proper ICMP format. Since ICMP messages are sent as part of the IP datagram, you must specify the IP header. In addition, you must specify the ICMP header for each message used. For example, if you use the ICMP echo message, then you must specify a basic IP header and an ICMP header for the echo message.

Figure 6-2 shows the ICMP message format.

```
                    1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
```

Type	Code	Checksum
Unused		
IP Header + 64 bits of Original Data Datagram		

Figure 6-2 *A Sample ICMP Message*

For detailed information on how to fill in headers for ICMP messages, see *RFC 792 Internet Control Message Protocol*.

Specifying an ICMP Message Header

ICMP message headers start with the following three fields: Type, Code, and Checksum. Table 6-3 describes each field and indicates and the number of bits in each field.

Table 6-3 Elements in the ICMP Message Header

Field	Description	Bits
Type	Indicates the specific message being used and determines the format of the remaining data.	8
Code	Indicates the particular reason for the message. For error messages, it indicates, for example, the particular reason that data did not reach its destination.	8
Checksum	Checks the header elements for errors at each point that the ICMP header is processed. The algorithm is the 16-bit one's complement of the one's complement sum of all 16-bit words of the ICMP message, starting with the Type.	16

ICMP can send messages and replies. Table 6-4 lists and describes ICMP messages.

Table 6-4 Description of ICMP Messages

Message	Description
`Source Quench`	Requests that the host send messages to the Internet destination at a slower rate.
`Echo`	Sends a message to a host.
`Subnet Mask Request`	Requests that a host send a reply containing its address mask.
`Information Request`	Requests that a host send a reply containing the number of the network it is on.
`Destination Unreachable`	Indicates that the network specified in the Internet destination field of a datagram is unreachable or that the datagram could not be delivered.
`Redirect`	Indicates to the host that a path shorter than the one indicated exists for the specified destination.
`Time Exceeded`	Indicates that the Time to Live is 0 or that a host did not receive all the fragments in time to complete reassembly of the datagram.
`Parameter Problem`	Indicates that a host or gateway encountered a problem with the IP header parameters.
`Timestamp`	Sends a message to a foreign host indicating the time the sender last touched the message before sending it.

Table 6-5 lists and describes ICMP replies.

Table 6-5 Description of ICMP Replies

Reply	Description
`Echo Reply`	Returns the same message to the host that sent an echo message.
`Subnet Mask Reply`	Sends a reply containing an address mask to a host that has sent a subnet mask request.
`Information Reply`	Sends a reply to a host that has sent an information request.
`Timestamp Reply`	Returns the message to a host that sent a timestamp and includes the time the recipient first touched it.

A Sample Program: pong.c

This section contains a program, **pong.c**, that uses the raw socket interface and the ICMP netmask request to communicate with remote machines. The **pong** program illustrates how to specify headers for IP, ICMP, and the ICMP netmask request.

The **pong** program sends an ICMP netmask request message to a host using a raw socket interface. If the ICMP packet is sent and received correctly, then a message is printed indicating the network mask of the requested host. If there are errors locating the host, creating the socket, sending the message, or receiving the message, then an error message is printed.

The program continues testing the network until **timeout** seconds have elapsed, or an answer is received. The default timeout is 20 seconds. The program accepts either a hostname argument or an Internet address.

```
/**********************************************************
* Copyright (C) Data General Corporation, 1988 - 1989     *
* All Rights Reserved.                                    *
* Licensed Material-Property of Data General Corporation. *
**********************************************************/

    /***************************************************
    * This software is made available solely pursuant *
    * to the terms of a DGC license                   *
    * agreement that governs its use.                 *
    ***************************************************/

/*
 * pong host [timeout]
 *
 *   attempts to see if machine is alive by ponging it for
 *   timeout seconds (default is 20)
 */
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/time.h>
#include <sys/param.h>
#include <netinet/ip.h>
#include <netinet/ip_icmp.h>
#include <signal.h>
#include <sys/wait.h>
#include <ctype.h>
#include <malloc.h>
#include <memory.h>
char  *address_to_string();
char  *host;
int   noresponse();
int   end_it_all();
int   my_pid;

#define DEFTIMEOUT 20
#define MAXALARM 2147483647   /* max arg to alarm() */

struct in_addr inet_addr();
main(argc, argv)
        int  argc;
        char *argv[];
{
        char *buf_ptr;
        char *buf_icmp_ptr;
        char *buf_ip_ptr;
        char *buf_time_ptr;
        char mysys[512];
        struct icmp icmp_hdr;
        struct icmp *icmp_hdr_ptr = &icmp_hdr;
        struct in_addr address;
        struct ip ip_hdr;
        struct ip *ip_hdr_ptr = &ip_hdr;
        struct timeval time;
        struct hostent *hp;
        struct sockaddr_in to, from;
        union wait status;
        int  len, cc, packetsize;
        int  timeout, s;
        if ( argc < 2 ) {   /*   usage message   */
           fprintf( stderr, "usage: pong host [timeout]\n" );
           exit(1);
        }
```

```
/*   Determine Internet address of remote host to pong  */
/*   The parameter to pong could be the Internet        */
/*   address or the hostname                            */

        host = argv[1];

/* Test for how address was specified.  */

        if ( isdigit(host[0]) )  {
           /*  Address specified in digits */
           address = inet_addr( host );
        }
        else {
           /*  Address specified as hostname  */
              if ( (hp = gethostbyname( host )) == NULL) {
                 fprintf( stderr, "can't find host %s\n", host );
                 exit(1);
              }
              address = *((struct in_addr *)hp->h_addr);
        }
/*  If third parameter was specified, use it as a timeout value. */
        if ( argc == 3 ) {
           timeout = atoi( argv[2] );
           if ( timeout < 0 || timeout > MAXALARM ) {
              fprintf( stderr, "invalid timeout\n" );
              exit(1);
           }
        }
        else {  /* Otherwise use the default timeout value.  */
             timeout = DEFTIMEOUT;
        }

/*  Get hostname of own system and look up the Internet */
/*  address corresponding to that hostname.             */

        gethostname( mysys, sizeof(mysys) );
        if ( (hp = gethostbyname( mysys )) == NULL ) {
           fprintf( stderr, "can't find host %s\n", mysys );
           exit(1);
        }
/* Allocate socket to make ICMP request */
        if ( (s = socket(AF_INET, SOCK_RAW, IPPROTO_ICMP) ) < 0) {
           perror( "pong: socket" );
           exit(1);
        }

/* Packet holds IP and ICMP information */
        packetsize = sizeof(struct ip) + sizeof(struct icmp);
        buf_ptr = malloc( packetsize );
/* Set socket type and address for sending */
        memset( (char *)&to, '\0', sizeof(struct sockaddr_in) );
        to.sin_family = AF_INET;
        to.sin_addr = address;

/* Initialize IP data in packet */
        memset( (char *)ip_hdr_ptr,'\0', sizeof(struct ip) );
        ip_hdr_ptr->ip_v = 4;
        ip_hdr_ptr->ip_hl = 5;
        ip_hdr_ptr->ip_len = packetsize;
        ip_hdr_ptr->ip_ttl = 0xff;
        ip_hdr_ptr->ip_p = 1;
        ip_hdr_ptr->ip_src = INADDR_ANY;
        ip_hdr_ptr->ip_dst = address;
        buf_ip_ptr = buf_ptr;
        memcpy( buf_ip_ptr, (char *)ip_hdr_ptr, sizeof(struct ip) );
```

```
/* Initialize ICMP data in packet */
        memset( (char *)icmp_hdr_ptr, ' ', sizeof(struct icmp) );
        icmp_hdr_ptr->icmp_type = ICMP_AMREQ;
        icmp_hdr_ptr->icmp_id = 1;
        icmp_hdr_ptr->icmp_seq = 1;
        buf_icmp_ptr = buf_ip_ptr + sizeof(struct ip);
        memcpy( buf_icmp_ptr, (char *)icmp_hdr_ptr, sizeof(struct icmp) );

/* Calculate checksum and place it in the packet */
        ((struct icmp *)buf_icmp_ptr)->icmp_cksum =
              in_checksum( (short *)buf_icmp_ptr,
              sizeof(struct icmp) );
/* Fork a child process to receive response from the ICMP request.  */
/* If there is no response within 20 seconds, the process           */
/* will terminate.                                                   */

       my_pid = fork();
       if ( my_pid  < 0 ) {
           perror( "pong: fork" );
           exit(1);
       }
       if (my_pid != 0) {                      /* parent */
          signal( SIGINT, end_it_all );
          for (;;) {
                   if ( sendto(s, buf_ptr, packetsize, 0, &to, sizeof(to) ) != packetsize ) {
                       perror( "pong: sendto" );
                       kill ( my_pid, SIGKILL );
                       exit(1);
                   }

                   sleep(1);
                   if ( wait3(&status, WNOHANG, 0) == my_pid )

                       if ( status.w_termsig == 0 )
                          exit(status.w_retcode);
                       else
                      exit(-1);
          }                      /* end of for loop */
       }                         /* end of if */
       if ( my_pid == 0 ) {                    /* child */
          alarm( timeout );
          signal( SIGALRM, noresponse );
          for (;;) {
              len = sizeof(from);
              if ( (cc = recvfrom(s, buf_ptr, packetsize, 0, &from, &len)) < 0 ) {
                 perror( "pong: recvfrom" );
                 continue;
              }
              if ( cc != packetsize ) {
                 continue;
              }
              if ( ((struct icmp *)buf_icmp_ptr)->icmp_type != ICMP_AMREPLY ) {
                 continue;
              }

              printf( "%s has address mask %x\n",
                      address_to_string(from.sin_addr),
                      ((struct icmp *)buf_icmp_ptr)->icmp_address_mask );
              exit(0);
          }
       }
                   /* NOTREACHED */
}
```

```
/*  kill process  */
end_it_all()
{
        kill ( my_pid, SIGKILL );
        exit(1);
}

noresponse()
{
        printf( "no response from %s\n", host );
        exit(1);
}
/*  Calculate checksum */
in_checksum( addr, length )
        u_short *addr;
        int length;
{
        register u_short *ptr;
        register int sum;
        u_short *lastptr;

        sum = 0;
        ptr = (u_short *)addr;
        lastptr = ptr + ( length/2 );
        for ( ; ptr < lastptr; ptr++ ) {
                sum += *ptr;
                if ( sum & 0x10000 ) {
                        sum &= 0xffff;
                        sum++;
                }
        }
        return (~sum & 0xffff) ;
}
char *
address_to_string( address )
        struct in_addr address;
{
        struct hostent *hp;
        char buf[100];

        hp = gethostbyaddr( (char *)&address, sizeof(address), AF_INET );
        if ( hp == NULL ) {
           sprintf( buf, "0x%x", address.s_addr );
           return buf;
        }
        else {
                return hp->h_name;
        }
}
```

End of Chapter

Chapter 7
Using the Transport Layer Interface to Access TCP/IP

Previous chapters have described how to use the socket family of system calls in networking applications. The socket calls provide an interface to the TCP/IP protocol stack that directly accesses kernel services. Alternatively, you can use the Transport Layer Interface (TLI) to access TCP/IP. The TLI is a library of routines that uses STREAMS mechanisms to access transport-level services in the kernel.

Chapter 3 generally describes the system calls you use to open, use, and close a socket. This chapter describes the routines you use to establish, use, and close a transport connection through the TLI. It contrasts the use of socket calls with the use of TLI routines to perform specific communications functions. For detailed information about each of the routines covered in this chapter, see the appropriate manual page.

This chapter contrasts socket calls with TLI routines because many network programmers already use sockets and are familiar with them. In providing communication facilities between peer processes, the TLI and sockets are much alike. Thus, it should be easier to learn TLI if you already know sockets.

As you write networking applications, you may find that there are times that you want to use TLI routines, times that you want to use sockets, and times that either interface would do. If you want a program to be portable or to run on a system compatible with System V Release 4, use the TLI. If you want complete access to TCP/IP functionality, use sockets. Socket-based applications do not run on an OSI-based stack.

Chapter 3 is organized around the sequence of events when a client and server communicate through a socket. This chapter is organized around the sequence of events when a client and server communicate through a TLI-based transport endpoint. A local program that uses the TLI to access TCP/IP has to create a communication endpoint and bind an address/name to it. The TLI routines that act on a communication endpoint expect certain data structures. If the program is a server, it has to listen for and accept a connection request. If the program is a client, it has to place a request for a connection. After endpoints are connected, clients and servers need to send and receive data. When data transmission is complete, a program has to close the endpoint. The following sections cover these events in detail.

Opening a Communication Endpoint

Recall that one definition of the term socket is conceptual: it is simply a communication endpoint that you can give a name. In the socket realm, you create this communication endpoint through the **socket**(2) system call.

In the TLI realm, the communication endpoint is called a transport endpoint. To create a transport endpoint with the TLI, you must use the **t_open** routine. Figure 7-1 shows the syntax of the **t_open** routine.

```
#include <tiuser.h>

int fd;
char *path;     /* Read only */
int oflag;      /* Read only */
struct t_info protocol_info;    /* Write only */

fd = t_open(path, oflag, &protocol_info);
```

Figure 7-1 *Syntax of the t_open Routine*

You do not specify the protocol family type of service or optional protocol ID for **t_open** as you do for the **socket** system call. Instead, you use **t_open** to open a special file that identifies a particular transport provider. The *path* argument points to the pathname of the transport provider to open. When you use TLI to access TCP/IP, have *path* point to a file system entry for a STREAMS-based clonable driver such as **/dev/tcp** or **/dev/udp**.

The *oflag* argument of **t_open** identifies open flags. These flags indicate the open intent (read, write, or both) and, optionally, open behavior (wait for a carrier before return, and so on) of the connection. You can construct the value for *oflag* by performing the OR function with an open intent flag and an open behavior flag, or you can simply specify an open intent flag.

When you use TLI to access TCP/IP, you can use the **O_RDWR** open intent flag for *oflag*. This indicates that you intend to read and write through the connection. You can use the **O_NONBLOCK** optional open behavior flag.

For details about the open flags, see **open**(2).

You can use the *protocol_info* argument to return various characteristics of the underlying transport protocol (TCP or UDP). The argument points to a structure of type **t_info**. A **t_info** structure contains the following members:

```
long    addr;
long    options;
long    tsdu;
long    etsdu;
long    connect;
long    discon;
long    servtype;
```

The **addr** member of a **t_info** structure specifies the maximum size of a protocol-specific address. The **options** member specifies the size in bytes of protocol-specific options.

If the value of the **tsdu** member is greater than 0, it specifies the maximum size in bytes of a *transport service data unit (TSDU)*. A value of **0** means that the transport provider does not support TSDUs, but it does support sending a byte stream of data. A value of **−1** indicates no limit to the size of a TSDU. A value of **−2** indicates that the transport of normal data is not supported. Here are the values of **tsdu** for TCP, UDP, and IP:

Protocol	tsdu Value
TCP	0
UDP	65507
IP	65515

If the value of the **etsdu** member is greater than 0, it specifies the maximum size in bytes of an *expedited transport service data unit (ETSDU)*, or in the language of socket-based applications, a unit of urgent data. A value of **0** indicates that the transport provider does not support ETSDUs, but it does support sending an expedited data stream with no logical boundaries preserved across the connection. A value of **−1** indicates no limit to the size of the ETSDU. A value of **−2** indicates that expedited data is not supported. Here are the values of **etsdu** for TCP, UDP, and IP:

Protocol	etsdu Value
TCP	0
UDP	−2
IP	−2

The **connect** member specifies the maximum amount of user data that can be sent with connection primitives. This is needed because some protocols support the transfer of user data with a connection request. The **discon** member specifies the maximum amount of data that can be sent with disconnect primitives. For TCP, the values of these members depend on the TCP Maximum Segment size, which is based on the Maximum Transmission Unit (MTU) of the interfaces used by peers to communicate. For example, two locally connected Ethernet peers have a Maximum Segment size of 1420. For UDP and IP, these members have the value of -2, since the protocols don't support connection-oriented service.

The **servtype** member specifies the provider service type. There are three legal values:

T_COTS	Connection-oriented service without orderly release
T_COTS_ORD	Connection-oriented service with orderly release
T_CLTS	Connectionless service

For TCP, **servtype** would have the value **T_COTS_ORD**. For UDP and IP, it would have the value **T_CLTS**.

The **t_open** routine returns a file descriptor **(fd)** that identifies the new transport endpoint. You would use this descriptor in later TLI calls to associate an address to the endpoint.

Specific examples should make the contrast between the **socket** call and the **t_open** routine clearer. Here's how you would use the **socket** call to open a TCP endpoint.

```
#include <sys/socket.h> /* defines AF_INET, SOCK_STREAM, and IPPROTO_TCP */
int net_fd;

net_fd = socket(AF_INET,SOCK_STREAM,IPPROTO_TCP);
if(net_fd < 0){
     perror("Socket failed\n");
     exit(1);
     }
```

In this call, the arguments assume the constant values **AF_INET** (Internet domain), **SOCK_STREAM** (stream sockets), and **IPPROTO_TCP** (TCP protocol). The code returns an error message if the **socket** call fails.

Here's how you would use **t_open** to open a TCP transport endpoint.

```
#include <tiuser.h>    /* defines T_CALL and other T_* structures */
#include <stdio.h>     /* defines print routines and NULL */
#include <fcntl.h>     /* defines O_RDWR used in t_open call */
int net_fd;

net_fd = t_open("/dev/tcp",O_RDWR, NULL);
if(net_fd < 0){
     t_error("t_open failed\n");
     exit(1);
     }
```

The first argument to the **t_open** routine is **/dev/tcp**, which is a STREAMS-based clonable driver that accesses the transport provider for a stream connection in the Internet domain. The second argument specifies the open intent flag **O_RDWR**: this means that the connection is intended for read and write operations. The third argument specifies NULL: this means that **t_open** returns no protocol information through the **protocol_info** argument. If you want an application to use the information returned by the **protocol_info** argument, declare the third argument as a **t_info** structure. As with the **socket** system call, an error message is returned if the call fails.

Allocating Data Structures

Most of the data structures passed between a transport user and transport provider contain one or more **netbuf** structures, each of which contains a pointer to a buffer used to send and receive data or addresses. The **netbuf** structure is covered later in this section.

You must explicitly tell the TLI routines that operate on a transport endpoint what kind of data to expect and in what format to expect it. One way to do this is through the **t_alloc** routine.

When TLI routines operate on a transport endpoint, they use a specified set of data structures to manipulate data. You use the **t_alloc** routine to allocate dynamically a particular data structure for the task that you want to perform. The specific data structures that **t_alloc** allocates are covered later in this section. Alternatively, you can specify statically the data structures that the TLI routines should expect.

You also use **t_alloc** to allocate memory for buffers referenced by a data structure. The maximum buffer sizes are all available in the **t_info** structure returned by **t_open**. This point is covered in a little more detail later.

In the socket realm, you would either use **malloc**(3) to dynamically allocate the data structures, or explicitly declare the structures in the user program. For more information about **malloc**, see the manual page.

Figure 7-2 shows the syntax of the **t_alloc** routine.

```
#include <tiuser.h>

int fd;
int struct_type;
int fields;
char *t_alloc(fd, struct_type, fields)
```

Figure 7-2 *Syntax of the t_alloc Routine*

The **fd** is the file descriptor. Each of the six allowed values of *struct_type* specifies the allocation of a specific type of structure:

T_BIND	allocates a **t_bind** structure
T_CALL	allocates a **t_call** structure
T_DIS	allocates a **t_discon** structure
T_UNITDATA	allocates a **t_unitdata** structure
T_UDERROR	allocates a **t_uderr** structure
T_INFO	allocates a **t_info** structure

Each of these structures is described in more detail in the section about the TLI routine that uses them. For example, the **t_bind** structure is discussed in the section that explains how to use the **t_bind** routine.

You use **t_alloc**'s *fields* argument to allocate memory for the buffer associated with the particular data structure specified. The arguments that you can pass are the bitwise-OR of any of the following:

T_ADDR	Allocate the **addr** member of the **t_bind**, **t_call**, **t_unitdata**, or **t_uderr** structures.
T_UDATA	Allocate the **udata** member of the **t_call**, **t_discon**, or **t_unitdata** structures.
T_ALL	Allocate all relevant members of the given structure.

Here is an example that uses **t_alloc** to allocate address information to which an endpoint is bound.

```
int result;
struct t_bind* bind_info_ptr;
struct sockaddr_in *sin_ptr;
struct servent *service_ptr;
.
.
.
/* Assume that the file descriptor is open */

bind_info_ptr = (struct t_bind*)t_alloc(lstn_fd,T_BIND,T_ADDR);
if(bind_info_ptr == NULL){
     t_error("t_alloc of T_BIND packet failed\n");
     exit(1);
     }
bind_info_ptr->addr.len = sizeof(struct sockaddr_in);
bind_info_ptr->qlen = 2;
sin_ptr = (struct sockaddr_in*)bind_info_ptr->addr.buf;
memset((char *)sin_ptr, 0, sizeof(*sin_ptr));
sin_ptr->sin_port = service_ptr->s_port;
sin_ptr->sin_family = AF_INET;
sin_ptr->sin_addr.s_addr = INADDR_ANY;
```

In this example, the **t_alloc** routine allocates a **t_bind** data structure for the descriptor named **bind_info_ptr** and allocates memory for the **addr** member of the structure. Then, the address structure pointed to by the **addr** member is initialized.

To release an allocated data structure, use the **t_free** routine. Figure 7-3 shows the syntax of the routine.

```
#include <tiuser.h>

int retcode;
char *ptr;
int struct_type;

retcode = t_free(ptr, struct_type);
```

Figure 7-3 *Syntax of the t_free Routine*

The **t_free** routine frees memory for the specified structure, and also frees memory for buffers referenced by the structure. The *ptr* argument points to one of the six structure types described for **t_alloc**, and *struct_type* identifies the type of the structure.

The data structures allocated by **t_alloc** most often contain one or more structures of type **netbuf**. The **netbuf** structure consists of the following members:

```
unsigned int maxlen;
unsigned int len;
char    *buf;
```

The **buf** member points to a data buffer. The **maxlen** member has meaning only when you use **buf** to receive data from a TLI routine; then, it specifies the amount of data that can be copied into the buffer. When you pass data from a read-only buffer to a TLI routine, **maxlen** is ignored.

When you use **buf** to receive data, the value of **len** is specified on return to be the amount of data actually copied into the buffer. When you use **buf** to send data, the **len** argument specifies the number of valid bytes in the buffer. If you use **buf** for both input and output, the calling routine replaces the value of **len** on return.

The layout of **buf** depends on whether you use it to pass an address, option information, or user data. Figure 7-4 shows how you could use a **netbuf** structure to send the address of a transport endpoint to another TLI routine. The figure shows the address in the form of a **sockaddr_in** structure, which is defined in **/usr/include/sys/socket.h** and is discussed in detail in Chapter 3. The maximum length of the buffer is specified as 1024 bytes, but because you are sending data to a TLI routine, this specification is ignored. The **len** is 16, which is the length of a **sockaddr_in** structure.

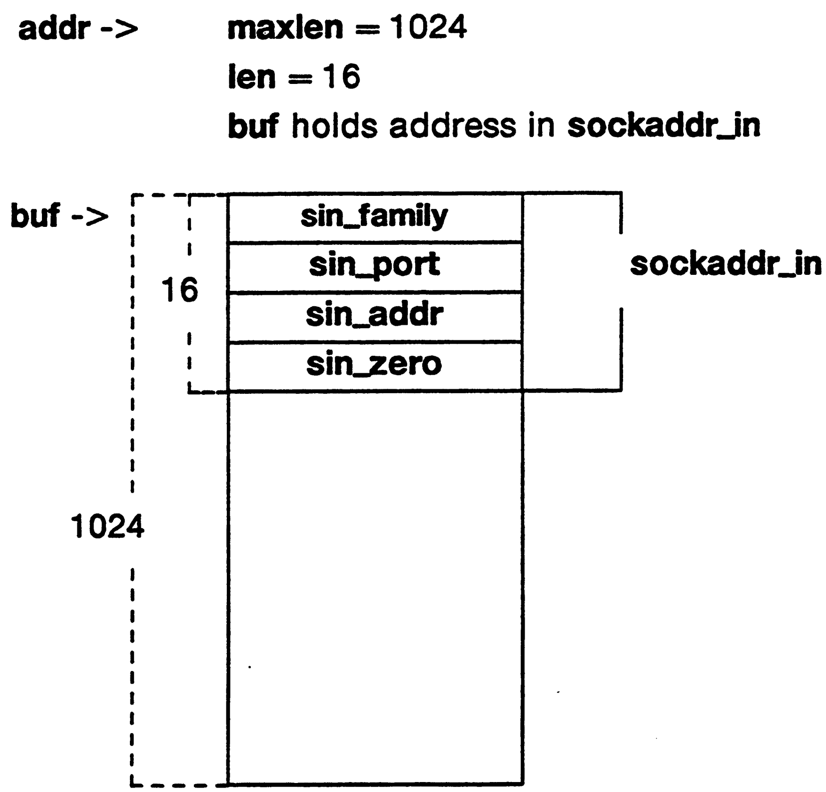


Figure 7-4 *Sending an Internet Address Through netbuf*

Sections of Chapters 3, 4, and 6 describe how to set and get protocol-specific options through the socket system calls. When you set and get protocol-specific options through a TLI routine, you put **opthdr-value** pairs into **netbuf's** data buffer. The **opthdr** is a fixed-length structure that specifies the protocol-specific option you wish to set or get. The **value** specifies the option value itself.

The **opthdr** structure contains three members: **level, name,** and **length**. Valid **level-name** pairs in the **opthdr** structure are as follows. The **length** specifies the length of the option value.

Table 7-1 Valid level-name Pairs for the opthdr Structure

level	name
IPPROTO_IP	IP_TX_OPTIONS IP_TOS IP_TTL IP_DONTFRAG IP_RX_OPTIONS
IPPROTO_TCP	TCP_NODELAY TCP_MAXSEG TCP_URGENT_INLINE TCP_PEER_ADDRESS TCP_ACCEPT_QUEUE_LENGTH

Figure 7-5 shows two options (32 bytes) being passed through a **netbuf** structure.

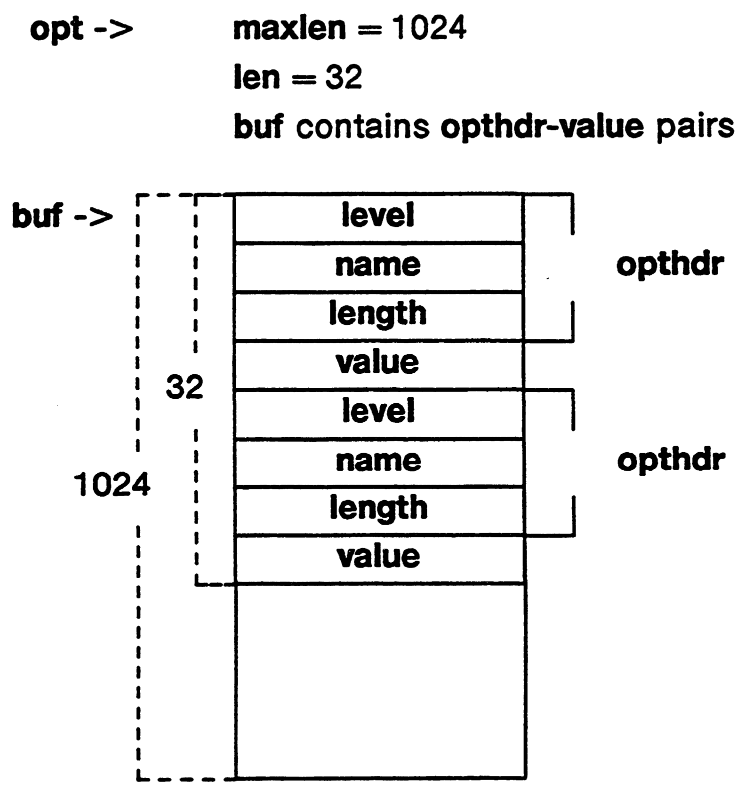


Figure 7-5 *Passing Protocol-Specific Options Through netbuf*

For more information about IP options, see Chapter 6. For more information about TCP options, see Chapter 4.

Figure 7-6 shows how you could use **buf** to receive data from a TLI routine. Assume that you are using a buffer of 1024 bytes, and that the routine returns with 32 bytes of data.

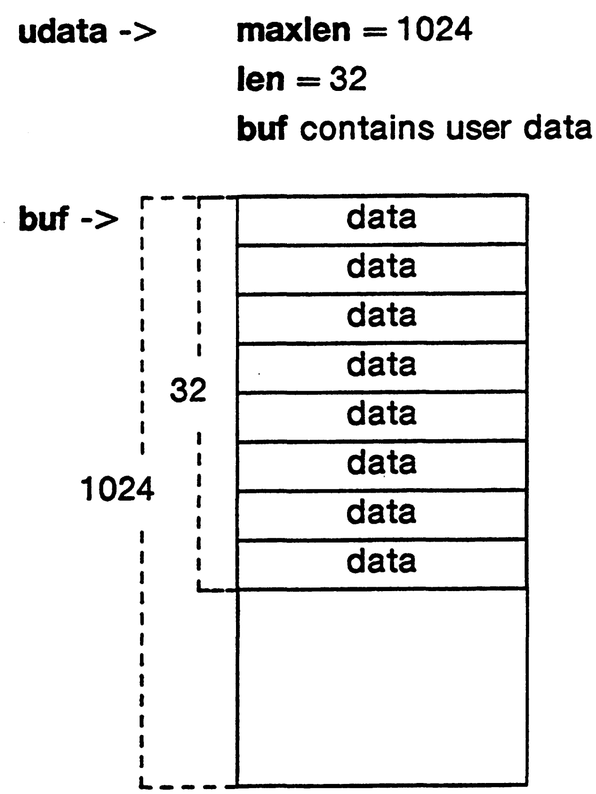


Figure 7-6 *Receiving Data Through netbuf*

Binding an Address to an Endpoint

In the socket realm, the **bind** system call assigns a name to a communication endpoint. In the TLI realm, the analog to the **bind** system call is the **t_bind** routine. Figure 7-7 shows the syntax of the routine.

```
#include <tiuser.h>

int fd;
int retcode;
struct t_bind *req;
struct t_bind *ret;

retcode = t_bind(fd, req, ret);
```

Figure 7-7 *Syntax of the t_bind Routine*

The **t_bind** routine associates a local address with the transport endpoint that you specify (here **fd**) and activates that endpoint. The *req* and *ret* arguments each point to a **t_bind** structure, which contains the following members:

```
struct netbuf addr;
unsigned qlen;
```

The **addr** member of the **t_bind** structure is a **netbuf** structure that contains an address. The **qlen** member of the **t_bind** structure, which is meaningful only for connection-oriented servers, indicates the maximum number of outstanding connection requests.

Use *req* to request that an address be bound to the transport endpoint specified by **fd**. When you use the TLI to access TCP/IP, the address is conveyed in a **sockaddr_in** structure. The **len** member of the *req* structure specifies the number of bytes in the address, and the **buf** member points to the address. The **maxlen** member has no meaning for *req*.

If *req* is NULL or if *req* is not null and the length of the address is 0, it does not matter to the transport user what address gets assigned to the endpoint, and **qlen** is assumed to be 0. The provider uses a wildcard IP address and an unused port number in the range 1024 to 5000. If *req* is not null and the length of the address is greater than 0, the transport user specifies an address for the transport provider to assign to the endpoint.

On return, *ret* contains the address that the transport provider actually bound to the transport endpoint; this may be different from the address specified in **req**. Again, the address is conveyed in a **sockaddr_in** structure. You specify the maximum size of the address buffer in the **maxlen** member of the *ret* structure. Specify the buffer where the address is to be placed in the **buf** member of *ret*. On return, **len** specifies the number of bytes in the bound address and **buf** points to the bound address.

If the **qlen** member has a value greater than 0, the endpoint is passive, accepting connections. Then, the value of **qlen** specifies how many connection requests can be enqueued at that particular transport endpoint.

What happens after **t_bind** activates the endpoint depends on whether the transport user is a server or a client, and on whether it is connection-oriented or connectionless. A connection-oriented server may begin accepting connections on the transport endpoint immediately after **t_bind** activates the endpoint. In this case, then, **t_bind** handles the functionality of the **bind** and **listen** calls in the socket realm.

For a connection-oriented client, the **t_bind** routine handles only the functionality associated with the **bind** call. The transport user must issue a **t_connect** after **t_bind** to initiate a connection with a server.

A connectionless user (server or client) may begin sending or receiving data through the transport endpoint immediately after the **t_bind** is issued.

Here's how you use the **bind** call to associate an address with a socket.

```
#include <sys/socket.h>     /* defines AF_INET and SOCK_STREAM */
#include <arpa/inet.h>      /* defines inet_ntoa */

int sock_fd;
struct sockaddr_in addr_base;
struct sockaddr_in *addr = &addr_base;
.
.
.
addr_base.sin_port = 0;
addr_base.sin_family = AF_INET;
addr_base.sin_addr.s_addr = INADDR_ANY;

if( bind( sock_fd, addr, sizeof(struct sockaddr_in) ) == -1 ) {
   fprintf( stderr, "bind failed with errno %d\n", errno );
   exit(1);
   }
```

Lines of code before the **bind** call explicitly fill in the members of the address structure bound to the endpoint.

Here's how you can use **t_bind** routine to associate a local address with a transport endpoint and activate the endpoint.

```
/*  Assume that net_fd is a transport endpoint previously
    created by a call to t_open.  */

     result = t_bind(net_fd, NULL, NULL);
     if(result < 0){
     t_error("t_bind failed");
     exit(1);
     }
```

Notice that a NULL value is specified for the **req** argument. This means that it does not matter to the transport user what address gets assigned to the endpoint.

Listening for and Accepting a Connection Request

In the socket realm, a server program uses the **listen** call to specify the length of a queue of connection requests on a particular socket, and the **accept** call to extract a connection request from the queue and establish the connection. In the TLI realm, the maximum length of a queue of connection requests is specified through the **t_bind** routine. A server uses **t_listen** to obtain information about a pending connection. When **t_listen** returns an indication of a connection, a server typically uses **t_open** to open a new endpoint and **t_bind** to bind to it. Finally, a server accepts the connection through the **t_accept** routine.

Figure 7-8 shows the syntax of the **t_listen** routine, which a server routine uses to listen for a client's request for service.

```
#include <tiuser.h>

int retcode;
int fd;
struct t_call *call;

retcode = t_listen(fd, call);
```

Figure 7-8 *Syntax of the t_listen Routine*

As you have seen, in the TLI realm the **t_bind** routine associates a protocol address with a particular transport endpoint and activates that endpoint. A connection-oriented server may begin accepting connections on an endpoint immediately after **t_bind** activates the endpoint. The **t_bind** routine also specifies the number of outstanding connection indications that the transport provider should support for the given endpoint. Thus, the **t_bind** routine combines the functionality of the **bind** and **listen** calls in the socket realm.

The **t_listen** routine listens for a connection indication (**T_CONN_IND**) from a calling transport user. A server process needs this indication to accept a connection; it cannot accept a new connection without this indication.

The **fd** argument specifies the descriptor of the transport endpoint where connection indications arrive. On return, *call* contains information about the connection indication. The *call* argument points to a **t_call** structure.

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
int sequence;
```

In *call*, the **addr** member returns the protocol address of the calling transport user, the **opt** member returns protocol-specific parameters associated with the connect request, and the **udata** member returns any user data sent by the caller on the connect request. The **sequence** is a number that uniquely identifies the returned connection indication. The **sequence** value enables the transport user process to listen for multiple connect indications before responding to any of them.

Thus, the *call* argument of the **t_listen** routine passes information about the connection indication. To a large extent, it provides the functionality of the **accept** call in the socket realm.

Since **t_listen** returns values for the **addr**, **opt**, and **udata** members of **call**, the **maxlen** member of each structure must be set before issuing the **t_listen** to indicate the maximum size of the buffer for each.

By default, **t_listen** waits for a connection indication to arrive before returning to the transport user. However, if you set **O_NDELAY** (through **t_open** or **fcntl**), **t_listen** polls for existing connection indications. If there are none, **t_listen** returns **−1** and sets **t_errno** to the value **TNODATA**.

A server accepts a connection through the **t_accept** routine. Figure 7-9 shows the syntax of this routine.

```
#include <tiuser.h>

int retcode;
int fd;
int resfd;
struct t_call *call;

retcode = t_accept(fd, resfd, call);
```

Figure 7-9 *Syntax of the t_accept Routine*

The **fd** argument of the **t_accept** routine identifies the local transport endpoint where the connection indication arrived. The *resfd* argument specifies the local transport endpoint where the connection is to be established. The *call* argument is a **t_call** structure that contains information required by the transport provider to complete the connection. The **t_call** structure, as you have seen, contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

In this case, **addr** is the address of the caller, **opt** contains any protocol-specific parameters associated with the connection, **udata** points to any user data to be returned to the caller, and **sequence** is the value returned by **t_listen** that uniquely associates the response with a previously received connection indication.

The following program fragment from Chapter 4 uses the **listen** and **accept** system calls. Assume that the local socket has been opened and named.

```
#include <stdio.h>
#include <errno.h>
#include <sys/socket.h>
#include <netinet/in.h>

int sock_desc, new_sock;          /* Two socket descriptors */
int retcode;                      /* Return code from system calls */
struct sockaddr_in cname;         /* Client socket name */
int cname_len = sizeof(cname);    /* Size of client socket name */

.
.
.

/* Get into the listen state */
retcode = listen(sock_desc, 1);
if( -1 == retcode ) {
    fprintf(stderr,"Cannot set socket to listen state, errno %d\n", errno);
    exit(1);
    }

/* Wait for a connection and return the first connection on the queue */
new_sock = accept(sock_desc, &cname, &cname_len);
if( new_sock < 0 ) {
    fprintf(stderr," Error in accept, errno %d\n",errno);
    exit(1);
    }
```

Here's a program fragment that uses the **t_listen** and **t_accept** calls to accept a connection from a remote host. Again, assume that the local endpoint has been opened and named.

```
#define SERVER_PORT 5001

int lstn_fd;            /* Descriptor for endpoint to listen for connections  */
int new_con_fd;         /* Descriptor for new connection */
int result;
struct t_call * lstn_info_ptr;
struct sockaddr_in *sin_ptr;
.
.
.
/* Allocate T_CALL structure for t_listen.  */
lstn_info_ptr = (struct t_call *)t_alloc(lstn_fd,T_CALL,T_ALL);
if(lstn_info_ptr == NULL){
     t_error("t_alloc of T_CALL packet failed\n");
     exit(1);
     }

/* Wait for a connection to arrive.  */

result = t_listen(lstn_fd,lstn_info_ptr);
if(result < 0){
     t_error("t_listen failed");
     exit(1);
     }
con_bind_ptr = (struct t_bind*)t_alloc(new_con_fd,T_BIND,T_ALL);
if(con_bind_ptr == NULL){
      t_error("t_alloc of T_BIND packet for new con failed");
      exit(1);
      }
/* Open the new file descriptor.  */

new_con_fd = t_open("/dev/tcp", O_RDWR, NULL);
if(new_cond_fd < 0){
     t_error("t_open failed");
     exit(1);
     }
```

```
/* Do a bind on the new file descriptor.
   Address information is provided, but not required.  */

con_bind_ptr->addr.len = sizeof(*sin_ptr);
con_bind_ptr->qlen = 0;
sin_ptr = (struct sockaddr_in*)con_bind_ptr->addr.buf;
memset((char *)sin_ptr, 0, sizeof(*sin_ptr));
sin_ptr->sin_family = AF_INET;

result = t_bind(new_con_fd,con_bind_ptr,NULL);
if(result <0){
     t_error("new connection t_bind failed");
     exit(1);
     }

lstn_info_ptr->opt.len = 0;
/* Associate connection with new file descriptor.
   Use t_rcvdis to identify the cause of a disconnect if
   it occurs.  Use t_close to close the endpoint.  */

result = t_accept(lstn_fd,new_con_fd,lstn_info_ptr);
if(result < 0){
     t_error("t_accept failed");
     if(t_errno == TLOOK){
          result = t_rcvdis(lstn_fd,NULL);
          if(result < 0){
               t_error("t_rcvdis failed");
               exit(1);
          }
          result = t_close(new_con_fd);
          if(result < 0){
               t_error("t_close failed");
               exit(1);
          }
     continue;
     }
```

First, address information in a **t_bind** structure is allocated for **t_bind** through the **t_alloc** routine. Next, **t_bind** associates the listening endpoint with the allocated address information. Then, all relevant members of a **t_call** structure are allocated for **t_listen** through **t_alloc**. After the members are allocated, **t_listen** waits for a connection to arrive. When it does arrive, **t_open** opens a new connection, **t_bind** binds the connection to the new file descriptor, and **t_accept** associates the connection with the new file descriptor. If a disconnect occurs, **t_rcvdis** identifies the cause. Finally, **t_close** closes the endpoint. The **t_rcvdis** and **t_close** are discussed later in the chapter.

Thus, for the listening endpoint, the action of the **t_bind** routine mirrors the action of the **bind** and **listen** system calls. For the descriptor of the accepting endpoint, the action of the **t_open**, **t_bind**, **t_listen**, and **t_accept** routines mirrors the action of the **accept** system call.

Requesting a Connection

Once a socket is created and bound, it can communicate with another socket. Precisely how this happens depends on whether a process uses stream sockets or datagram sockets. With datagram sockets, you can send and receive data as soon as sockets at both ends of the connection are bound to an address. With stream sockets, a client program must first use the **connect** system call to establish a connection with a server.

With the TLI, a client process initiates a connection with a server through the **t_connect** routine. Figure 7-10 shows the syntax of the routine.

```
#include <tiuser.h>

int retcode;
int fd;
struct t_call *sndcall;
struct t_call *rcvcall;

retcode = t_connect(fd, sndcall, rcvcall);
```

Figure 7-10 *Syntax of the t_connect Routine*

The **t_connect** routine is valid only for connection-oriented transport endpoints.

The **fd** argument identifies the descriptor of the transport endpoint where the connection is established. The *sndcall* and *rcvcall* arguments point to **t_call** structures. Remember from the discussions of **t_listen** and **t_accept** that a **t_call** structure has the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

Here, **addr** specifies the caller's address (again, a **sockaddr_in** address), **opt** specifies call options, and **udata** contains user data. The **sequence** member has no meaning for the **t_connect** routine.

The *sndcall* argument specifies information needed by the transport provider to establish a connection. The *rcvcall* argument specifies the location of information about the new connection passed from the transport provider.

In *sndcall*, the **addr** member specifies the address of the peer's communication endpoint. The *opt* member presents any protocol-specific information that might be needed by the transport provider. The *udata* member points to optional user data that may be passed to the destination transport user during the establishment of a connection.

On return in *rcvcall*, **addr** returns the address associated with the responding transport endpoint. The *opt* member contains any protocol-specific information associated with the connection. The *udata* member points to optional user data that may be returned by the peer process during connection establishment.

By default, **t_connect** waits for the destination process's response before returning control to the local process. A successful return indicates that the connection has been established.

If you set the **O_NDELAY** option (through the **t_open** routine or the **fcntl** system call), **t_connect** does not wait for the remote process's response. Instead, it returns control immediately to the local process and returns **−1** with **t_errno** set to **TNODATA** to indicate that the connection has not yet been established.

Here's a code fragment that uses the **connect** system call.

```
int sock_fd;
int result;
struct sockaddr_in sin;
.
.
.
/* Assume socket is open and bound */

result = connect(sock_fd,&sin,sizeof(sin));
if(result < 0){
    perror("connect failed");
    exit(1);
    }
printf("Connection established\n");
```

Here's a code fragment that uses **t_connect** to connect to a TCP endpoint.

```
int result;
int net_fd;
struct t_call *call_info_ptr;
struct sockaddr_in *sin_ptr;
.
.
.

/* Assume transport endpoint has been opened and named.
   Allocate data structures for new endpoint. */

call_info_ptr = (struct t_call *)t_alloc(net_fd,T_CALL,T_ADDR);
if(call_info_ptr == NULL){
     t_error("t_alloc of T_CALL packet failed\n");
     exit(1);
}
call_info_ptr->addr.len = sizeof(struct sockaddr_in);
sin_ptr = (struct sockaddr_in*)call_info_ptr->addr.buf;
sin_ptr->sin_family = AF_INET;
sin_ptr->sin_port = service_ptr->s_port;
sin_ptr->sin_addr = host_address;

printf("Connecting to address=%s (%X), port number=%d\n",
      inet_ntoa(host_address),
      host_address.s_addr,
      service_ptr->s_port);
```

```
result = t_connect(net_fd,call_info_ptr,NULL);
if(result < 0){
    if(t_errno == TLOOK && t_look(net_fd) == T_DISCONNECT){
       printf("Connection not established\n");
       printf("Disconnection indication received\n");
       exit(1);
       }
    t_error("t_connect failed\n");
    exit(1);
    }
printf("Connection established\n");
```

To connect to the remote host, this code first declares, allocates, and initializes address information. Then, it asks the transport provider (TCP) to establish the connection.

Sending and Receiving Data over a Transport Connection

Like sockets, the TLI provides two kinds of communication service: connection-oriented and connectionless. How an application sends and receives data through a transport connection depends on the type of communication service it uses.

Connection-oriented communication allows the reliable transmission of data through an established connection. Connectionless communication transfers data in self-contained units. This kind of communication does not require an established connection.

Sending Data with Connection-Oriented Service

In the socket realm, an application uses the **write**, **writev**, or **send** system calls to send data after it establishes a connection. With connection-oriented service, a TLI-based application uses the **t_snd** routine to send data. Figure 7-11 shows the syntax of the routine.

```
#include <tiuser.h>

int retcode;
int fd;
char *buf;
unsigned nbytes;
int flags;

retcode = t_snd(fd, buf, nbytes, flags);
```

Figure 7-11 *Syntax of the t_snd Routine*

Use this routine to send either normal or expedited data. The **fd** argument identifies the local transport endpoint over which data should be sent. The *buf* argument points to the user data, *nbytes* specifies the number of bytes of user data to be sent, and *flags* specifies any optional flags.

By default, **t_snd** may wait if flow-control restrictions prevent the data from being accepted by the local transport provider when the call is made. However, if you set **O_NDELAY** (through **t_open** or **fcntl**) and flow-control restrictions, **t_snd** fails immediately.

Receiving Data with Connection-Oriented Service

In the socket realm an application uses the **read**, **readv**, or **recv** system calls to receive data after it establishes a connection. With connection-oriented service, a TLI-based application uses the **t_rcv** routine to receive data.

Figure 7-12 shows the syntax of the **t_rcv** routine.

```
int retcode;
int fd;
char *buf;
unsigned nbytes;
int *flags;

retcode = t_rcv(fd, buf, nbytes, flags);
```

Figure 7-12 *Syntax of the t_rcv Routine*

The **fd** identifies the local transport endpoint through which to expect data. The *buf* argument points to a buffer where user data is placed, and *nbytes* specifies the size of the buffer. The *flags* argument may be set on return from **t_rcv**; for details, see the manual page.

By default, **t_rcv** waits for data to arrive if none are currently available. However, if you have set **O_NDELAY** (through **t_open** or **fcntl**), **t_rcv** fails if no data are available.

Here's a socket-based example of sending and receiving data with connection-oriented service.

```
#define FROM_STDIN 0
#define FROM_NET 1
typedef int data_direction_type;
data_direction_type wait_data();
.
.
.
for(;;) {
     int nbytes;           /* Holds number of bytes moved */
     char buffer[80];      /* Holds data for/from network. */
     int result;

     switch (wait_data(fileno(stdin),net_fd)){
```



```
        case FROM_STDIN: {
             if (gets(buffer) == NULL){
                  printf("   Sending EOF    \n");
                  result = shutdown(net_fd,1);
                  if(result < 0){
                       perror("shutdown failed\n");
                  }
                  break;
             }
             strcat(buffer,"\r\n");

             nbytes = send(net_fd,buffer,strlen(buffer),0);
             if(nbytes < 0){
                  perror("send failed\n");
                  exit(1);
             }
             break;
        } /* end FROM_STDIN case */
        case FROM_NET: {
             nbytes = recv(net_fd,buffer,sizeof(buffer),0);
             if(nbytes <0){
                  perror("recv failed\n");
                  exit(1);
             }
             if(nbytes == 0){
                  printf("Received end of file\n");
                  goto END_OF_FILE;
             }

            write(fileno(stdout),buffer,nbytes);
            break;
        } /* end FROM_NET case */

        } /* end switch */

   } /* end data movement loop */

   END_OF_FILE:
```

Here's a TLI-based example of sending and receiving data with connection-oriented service.

```
   #define FROM_STDIN 0
   #define FROM_NET 1
   typedef int data_direction_type;
   data_direction_type wait_data();
   .
   .
   .
   for(;;) {
        int nbytes;           /* Holds number of bytes moved */
        char buffer[80];      /* Holds data for/from network. */
        int flags;            /* Used with t_* calls. */

        switch (wait_data(fileno(stdin),net_fd)){
```

```
        case FROM_STDIN: {
            if (gets(buffer) == NULL){
                int result;
                printf("   Sending EOF    \n");
                result = t_sndrel(net_fd);
                if(result < 0){
                     t_error("t_sndrel failed\n");
                }
                break;
            }
            strcat(buffer,"\r\n");

            flags = 0;
            nbytes = t_snd(net_fd,buffer,strlen(buffer),&flags);
            if(nbytes < 0){
                 t_error("t_snd failed\n");
                 exit(1);
            }
            break;
        } /* end FROM_STDIN case */
        case FROM_NET: {
            flags = 0;
            nbytes = t_rcv(net_fd,buffer,sizeof(buffer),&flags);
            if(nbytes <0){
                 t_error("t_rcv failed\n");
                 exit(1);
            }
            if(nbytes == 0){
                 printf("Received end of file\n");
                 goto END_OF_FILE;
            }

            write(fileno(stdout),buffer,nbytes);
            break;
        } /* end FROM_NET case */

        } /* end switch */

    } /* end data movement loop */

    END_OF_FILE:
```

In the first case, the code reads data from standard input. At the end of the file, the code does an orderly release. (The **t_sndrel** routine is covered later in this chapter.) A Carriage Return character and a New Line character are added to permit communication through FTP or SMTP. Finally, data is written to the network. In the other case, the code gets data from the network. It exits a loop when it gets an End of File character, and then writes data to standard output.

Sending Data with Connectionless Service

With sockets, you use the **sendto** or **sendmsg** system calls to send data with a connectionless protocol such as UDP. With connectionless service, an application uses the **t_sndudata** routine to send data. Figure 7-13 shows the syntax of this routine.

```
#include <tiuser.h>

int fd;
int retcode;
struct t_unitdata *unitdata;

retcode = t_sndudata(fd, unitdata);
```

Figure 7-13 *Syntax of the t_sndudata Routine*

Use this routine to send a data unit to another connectionless transport user. The **fd** argument identifies the local transport endpoint through which to send data, and *unitdata* points to a **t_unitdata** structure that contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

In the *unitdata* argument, **addr** specifies the protocol address of the destination user, **opt** identifies protocol-specific options that you want associated with this request, and **udata** specifies the data to be sent. You may choose not to specify which protocol options are associated with the transfer by setting the **len** member of **opt** to zero. In this case, the provider uses default options.

By default, **t_sndudata** may wait if flow-control restrictions prevent the data from being accepted by the local transport provider at the time the call is made. However, if you have set **O_NDELAY** (through **t_open** or **fcntl**), **t_sndudata** fails under such conditions.

Receiving Data with Connectionless Service

With sockets, you use the **recvfrom** or **recvmsg** system calls to receive data with a connectionless protocol such as UDP. With connectionless service, a TLI-based application uses the **t_rcvudata** routine to read data. Figure 7-14 shows the syntax of this routine.

```
#include <tiuser.h>

int retcode;
int fd;
struct t_unitdata *unitdata;
int *flags;

retcode = t_rcvudata(fd, unitdata, flags);
```

Figure 7-14 *Syntax of the t_rcvudata Routine*

Use this routine to receive a datagram from another connectionless transport user. The **fd** argument identifies the local transport endpoint through which to expect data. The *unitdata* points to a **t_unitdata** structure that holds data associated with the received datagram. If the *flags* argument is set to **T_MORE** on return, the application's buffer was not large enough to hold the entire datagram. The rest of the datagram may be retrieved with subsequent **t_rcvudata** calls.

The **t_unitdata** structure contains the following members:

```
struct netbuf addr;
struct netbuf opt;
struct netbuf udata;
```

The **addr** member specifies the address of the incoming or outgoing datagram. The **opt** member specifies options. The **udata** member specifies user data. The **maxlen** member of **addr**, **opt**, and **udata** must be set before issuing this routine to indicate the maximum size of the buffer for each.

By default, **t_rcvudata** waits for a datagram to arrive if none is currently available. However, if you have set **O_NDELAY** (through **t_open** or **fcntl**), **t_rcvudata** fails if no datagrams are available.

Releasing a Transport Connection

A connection-oriented TLI-based application can release a transport connection abruptly or gracefully. Figure 7-15 shows the syntax of the **t_snddis** routine, which you use to initiate an abrupt release on an already established connection or to reject a connect request.

```
#include <tiuser.h>

int retcode;
int fd;
struct t_call *call;

retcode = t_snddis(fd, call);
```

Figure 7-15 *Syntax of the t_snddis Routine*

The **fd** argument identifies the local transport endpoint of the connection. The *call* argument is a structure of type **t_call** that specifies information associated with the abortive release. As you have read, the **t_call** structure has the following members:

```
struct netbuf   addr;
struct netbuf   opt;
struct netbuf   udata;
int             sequence;
```

The meaning of the values in *call* differ depending on the context of the call to **t_snddis**. When rejecting a connect request, *call* must be non-NULL and contain a valid value of **sequence** to identify uniquely the rejected connection indication to the transport provider. In this case, the **addr** and **opt** members of the **t_call** structure are ignored.

In all other cases, *call* need only be used when data is being sent with the disconnect request. The **addr**, **opt**, and **sequence** members of the **t_call** structure are ignored. If you do not wish to send data to the remote user, the value of call may be **NULL**.

If the process that called **t_snddis** sends data with the disconnect, that is, if it sends data through the **udata** structure to which the *call* argument points, the receiving process has to pass a non-NULL *discon* argument to the **t_rcvdis** routine to retrieve the data. You use the **t_rcvdis** routine to identify the cause of a disconnect and to retrieve any user data sent with the disconnect.

Here is the syntax of the **t_rcvdis** routine:

```
#include <tiuser.h>

int retcode;
int fd;
struct t_discon *discon;

retcode = t_rcvdis(fd, discon);
```

Figure 7-16 *Syntax of the t_rcvdis Routine*

The **fd** argument identifies the local transport endpoint. The *discon* argument points to a structure of the type **t_discon**. The **t_discon** structure has the following members.

```
struct netbuf   udata;
int             reason;
int             sequence;
```

The **udata** member contains user data. The **reason** member specifies the error number of the disconnection (for example, **EACCES** to indicate that the caller has inadequate privileges to do what it tried to do). The **sequence** member specifies the sequence number. If the **sequence** is **−1**, the connection is associated with the Stream; if the **sequence** is not **−1**, one of the enqueued connections is yet to be accepted.

Figure 7-17 shows the syntax of the **t_sndrel** routine, which you use to release a connection gracefully.

```
#include <tiuser.h>

int retcode;
int fd;

retcode = t_sndrel(fd);
```

Figure 7-17 *Syntax of the t_sndrel Routine*

Use **t_sndrel** to initiate an orderly release of a transport connection and indicate to the transport provider that the transport user has no more data to send. The **fd** argument identifies the local transport endpoint where the connection exists.

After issuing **t_sndrel**, you may not send any more data over the connection. However, you may continue to receive data if an orderly release indication has been received.

This routine is an optional service of the transport provider, and is only supported if the transport provider returned service type **T_COTS_ORD** on **t_open** or **t_getinfo**(3).

A connection-oriented or connectionless TLI-based application can release a transport connection abruptly through the **t_close** routine. Figure 7-18 shows the syntax of this routine.

```
#include <tiuser.h>

int retcode;
int fd;

retcode = t_close(fd);
```

Figure 7-18 *Syntax of the t_close Routine*

Use the **t_close** routine to tell the transport provider that you are finished with the transport endpoint specified by **fd**. The routine frees any local resources associated with the endpoint. Also, **t_close** closes the file associated with an endpoint.

You should call **t_close** from the **T_UNBND** state. For details, see the **t_getstate**(3N) manual page. However, **t_close** does not check state information, so it may be called from any state to close a transport endpoint. If this occurs, the local library resources associated with the endpoint are freed automatically. In addition, **close**(2) is issued for that file descriptor. If no other process has the file open, the close breaks any transport connection that may be associated with that endpoint.

Handling Errors

A socket system call returns an error through the **errno** variable. A TLI routine sets the variable **t_errno** when it returns an error (like **errno**, **t_errno** is not cleared on successful calls). The array named **t_errlist** can be indexed by **t_errno** to get an ASCII error message for a particular value of **t_errno**.

Figure 7-19 shows how to call the global **t_error** routine, which writes a message to standard error that describes the last error encountered during a call to a particular routine.

```
#include <tiuser.h>

char *errmsg;
extern int t_errno;
extern char *t_errlist[];
extern int t_nerr;
void t_error(errmsg);
```

Figure 7-19 *Syntax of the t_error Routine*

The argument string **errmsg** is a user-supplied error message. **t_nerr** is the largest message number provided for in the **t_errlist** table.

Suppose a datagram is transmitted correctly by the transport provider but an error is detected in the datagram somewhere else in the network. For example, suppose the datagram has an invalid address. The provider needs a way to tell the user that an error has occurred. Also, the transport user needs some way of determining the cause of the error.

The TLI provides such a way by setting the return code of **t_rcvudata** to **−1** and by setting **t_errno** to **T_LOOK**. A program can then call the **t_rcvuderr** routine to determine what happened and to clear the error status. Figure 7-20 shows the syntax of the **t_rcvuderr** routine.

```
#include <tiuser.h>

int retcode;
int fd;
struct t_uderr *uderr;

retcode = t_rcvuderr(fd, uderr);
```

Figure 7-20 *Syntax of the t_rcvuderr Routine*

The **fd** argument identifies the local transport endpoint through which the error report is received. The *uderr* argument points to a **t_uderr** structure, which contains the following members.

```
struct netbuf   addr;
struct netbuf   opt;
long            error;
```

The **addr** member specifies the destination address of the packet that caused the error. The **opt** member specifies any options contained in the message that caused the error. The **error** member passes an error code.

If you do not care about the error indication, set the **uderr** argument to NULL. This clears the error status without returning information about the error. A datagram error may arrive at any time after the datagram was sent.

Opening, Using, and Closing a Connection

The previous sections have described how you would create, use, and close a transport connection through TLI routines. The following five figures depict a simple scenario of just that. Figure 7-21 shows the creation of a passive endpoint on a server system. Once created, this passive endpoint waits for a connection request from a client.

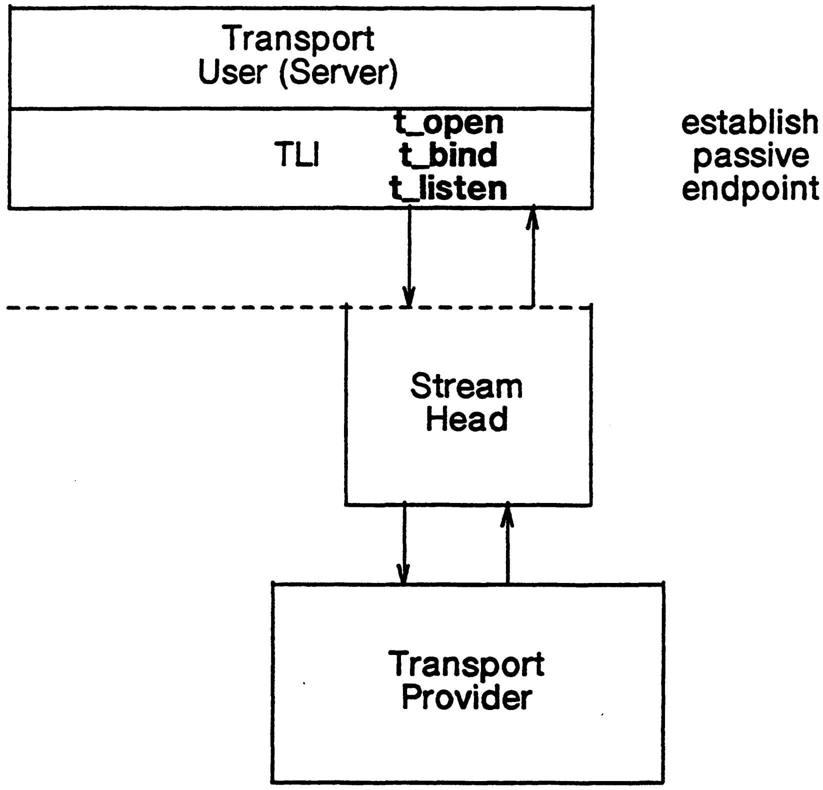


Figure 7-21 *Establishing a Passive Endpoint*

The server program (the transport user) uses **t_open** to open a STREAMS connection to the transport provider (let's assume in this case that it is TCP). It then uses **t_bind** to associate an address with a passive endpoint. It then uses **t_listen** to listen for connection requests on the endpoint.

A client program also uses **t_open** to open a STREAMS connection to the transport provider (TCP). It then uses **t_bind** to associate an address with an active endpoint. Then, it initiates a connection with a server through the **t_connect** routine. Figure 7-22 shows these events.

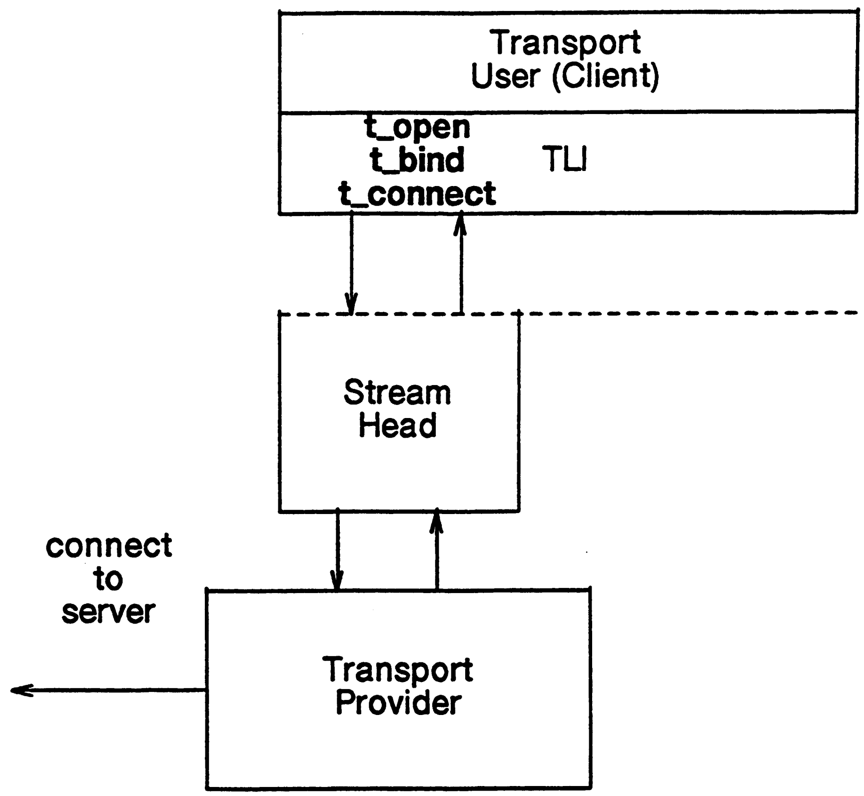


Figure 7-22 *Establishing an Active Endpoint*

Figure 7-21 showed the server program using **t_listen** to listen for connection requests on a passive endpoint. Figure 7-22 showed the client telling the server that it wanted a connection. This causes the transport provider to send the **T_CONN_IND** signal (connection indication) to the TLI streams head. The provider then opens a new communications endpoint. Figure 7-23 illustrates these events.

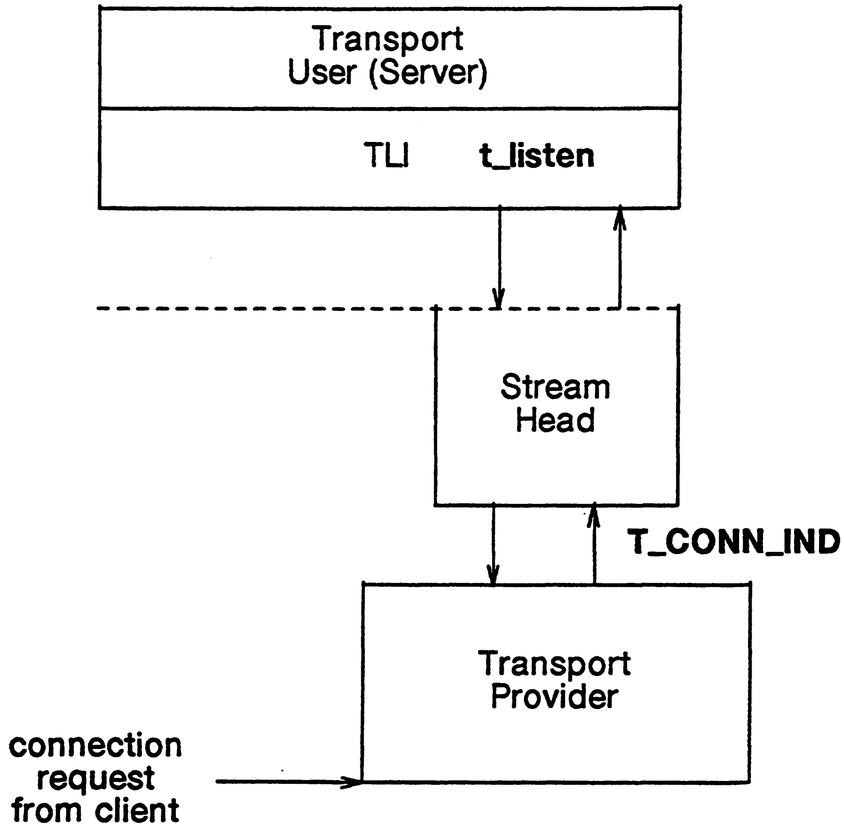


Figure 7-23 *Listening for a Connection*

Figure 7-24 shows that when the server program gets the connection indication signal, it uses **t_open** to create a new Stream head for a new connection. It then uses **t_bind** to associate an address with the new connection.

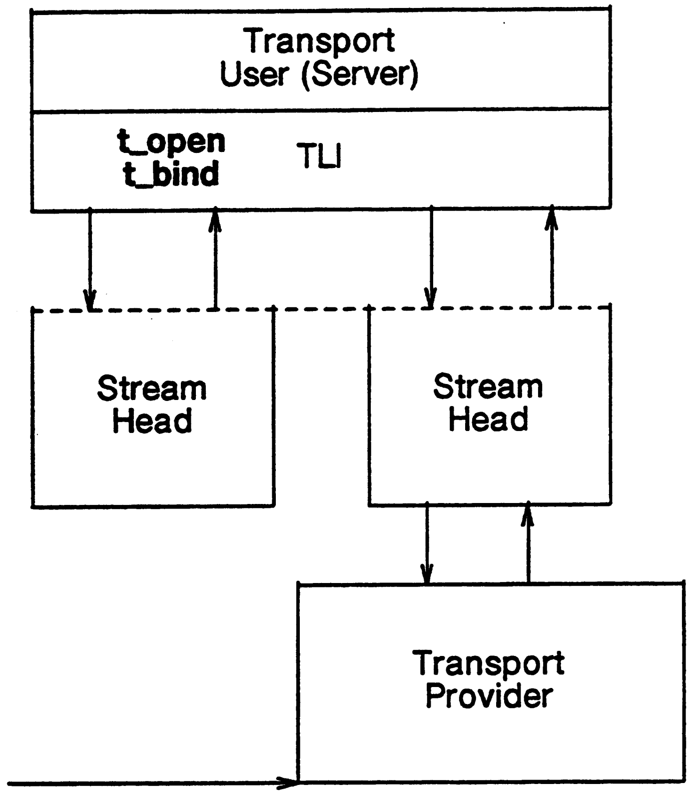


Figure 7-24 *Opening a New Connection*

Figure 7-25 shows that the server program uses **t_accept** through the original stream to accept the new connection.

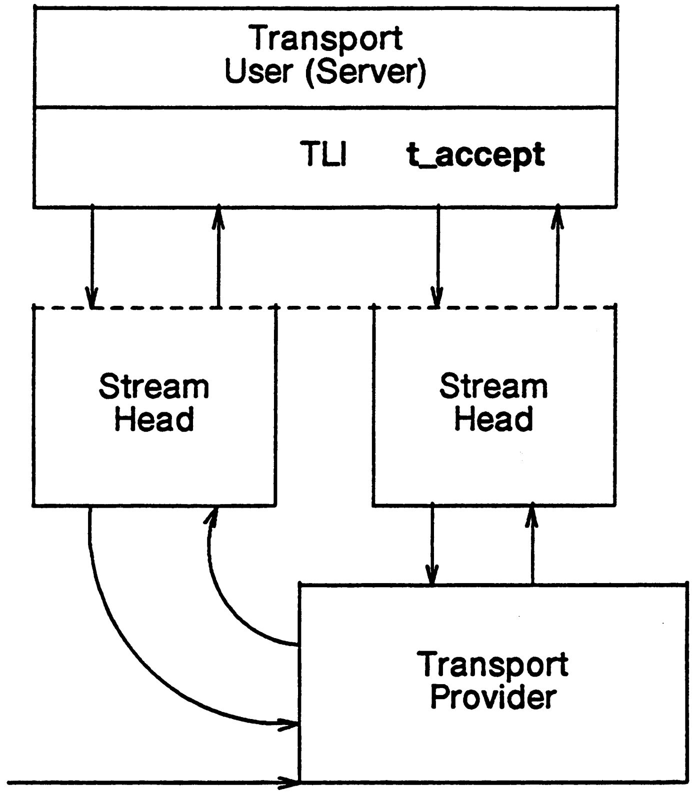


Figure 7-25 *Accepting the New Connection*

Finally, Figure 7-26 depicts how data is sent and received through the new Stream with **t_snd** and **t_rcv**. The passive endpoint waits for a new connection request.

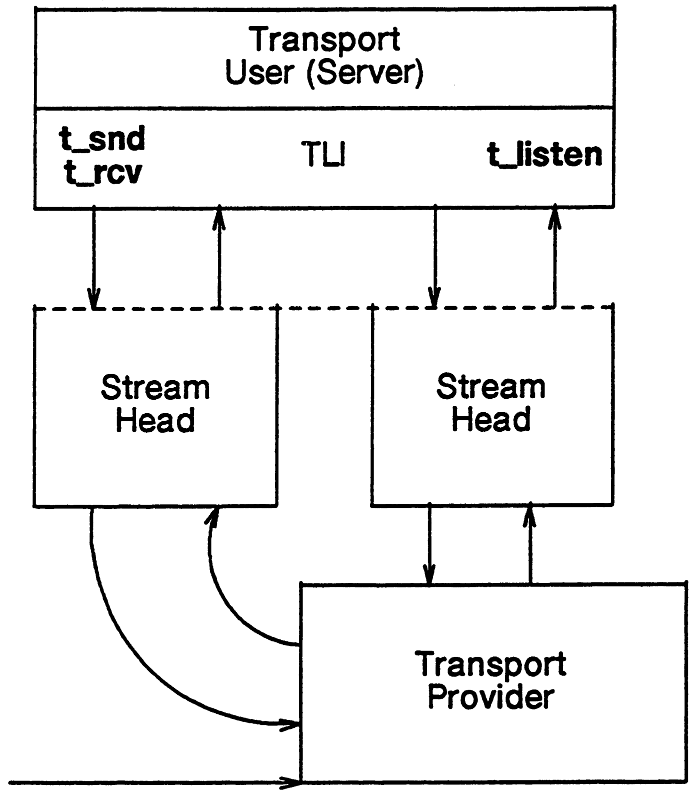


Figure 7-26 *Sending and Receiving Data Through the New Connection*

Comparison of Sockets to TLI Routines

Table 7-2 summarizes the comparison of sockets to TLI routines. It shows the sequence of socket calls and TLI routines that a server program, a client program, or either type would use to open, use, and close a communication endpoint.

Table 7-2 Comparison of Sockets and TLI Routines

Initiator	Activity	Sockets	TLI
Server or Client	Set endpoint data structure requirements	Data structure requirements depend on socket domain	**t_alloc()**
Server or Client	Create communication endpoint	**socket()**	**t_open()**
Server or Client	Bind address to communication endpoint	**bind()**	**t_bind()**
Server	Specify queue	**listen()**	For a connection-oriented server, **t_bind()** does the same work as **bind()** and **listen()**
Client	Connect to server	**connect()**	**t_connect()**
Server	Wait for a connection and get a new descriptor for the incoming connection	**accept()**	**t_listen()** **t_open()** **t_bind()** **t_accept()**
Server or Client	Send and receive data	**read()** **write()** **recv()** **send()**	**t_rcv()** **t_snd()**
Server or Client	Send and receive datagrams	**recvfrom()** **sendto()**	**t_rcvudata()** **t_sndudata()**
Server or Client	Close the connection	**close()** **shutdown()**	**t_close()** **t_sndrel()** **t_snddis()**

Compiling a Program to Use the TLI Library

To have a program use the TLI, link in the TLI library when you compile the program. For example, if you wanted a program named **user.c** to use the TLI, you would compile it with the following command line:

```
% cc user.c -lnsl ⮐
```

What follows are three sample programs: two that use TLI routines and one that uses sockets. The first program is a TLI-based server that passively receives a connection and transfers data. The second program is a TLI-based client that establishes a connection and transfers data. The last program is the same client application, but written to use sockets.

A TLI-Based Server Program

Here is a sample server program that uses TLI routines to passively receive a connection and transfer data. Invoke the program as follows:

% *program* [*service_name*] ⮐

The *service_name* can be either a name of a well-known service (the default is **echo**) or a port number. Once the connection has been established, the program reads data from the network and echos it back to the network.

```
#include <stdio.h>       /* Defines print functions and NULL. */
#include <memory.h>      /* Defines memcpy. */
#include <netdb.h>       /* Defines types for gethostbyname and getservbyname. */
#include <sys/types.h>   /* Defines u_long types used by following file. */
#include <netinet/in.h>  /* Defines in_addr. */
#include <arpa/inet.h>   /* Defines inet_ntoa. */
#include <fcntl.h>       /* Defines O_RDWR used in t_open call. */
#include <sys/socket.h>  /* Defines AF_INET. */
#include <tiuser.h>      /* Defines T_CALL and other T_* things for TLI. */

extern int t_errno;      /* Make TLI error codes available. */
main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
     char *portname;                   /* Holds service name to listen on. */
     struct servent* service_ptr;      /* Used to lookup port number. */
     struct servent  atoi_servent;     /* Used if getservbyname fails. */
     int lstn_fd;                      /* Holds listening file descriptor. */
     struct t_call * lstn_info_ptr;
     int result;
```

```
/*
 * Check if argument specifies service to listen on.
 */
if (argc > 1){
     portname = argv[1];
} else {
     portname = "echo";
}

if (argc >2 ){
     printf("Too many arguments\n");
     exit(1);
}

/*
 * Find port number for named service.
 * (This should be replaced with netdir_getbyname
 * (see netdir(3N)) to be transport independent.)
 */
service_ptr = getservbyname(portname,"tcp");
if(service_ptr == NULL){
     service_ptr = &atoi_servent;
     service_ptr->s_port = atoi(portname);
     if(service_ptr->s_port == 0){
          printf("Can't resolve portname %s\n",portname);
          exit(1);
     }
}

/*
 * Print banner with port number.
 */
printf("Starting %s with port number=%d\n",
     argv[0],
     service_ptr->s_port);

/*
 * Open tcp.
 */
lstn_fd = t_open("/dev/tcp",O_RDWR, NULL);
if(lstn_fd < 0){
     t_error("listen t_open failed");
     exit(1);
}

/*
 * Bind local port to listening file descriptor.
 *
 * First declare, allocate, and initialize address information.
 * Then ask Transport Provider to bind local port number.
 */
{
     struct t_bind* bind_info_ptr;
     struct sockaddr_in *sin_ptr;

     bind_info_ptr = (struct t_bind*)t_alloc(lstn_fd,T_BIND,T_ADDR);
     if(bind_info_ptr == NULL){
          t_error("t_alloc of T_BIND packet failed");
          exit(1);
     }
     bind_info_ptr->addr.len = sizeof(struct sockaddr_in);
     bind_info_ptr->qlen = 2;
     sin_ptr = (struct sockaddr_in*)bind_info_ptr->addr.buf;
     memset((char *)sin_ptr, NULL, sizeof(*sin_ptr));
     sin_ptr->sin_family = AF_INET;
     sin_ptr->sin_port = service_ptr->s_port;
```



```
        result = t_bind(lstn_fd, bind_info_ptr, NULL);
        if(result < 0){
             t_error("t_bind for lstn_fd failed");
             exit(1);
        }

        result = t_free(bind_info_ptr,T_BIND);
        if(result < 0){
             t_error("t_free bind_info_ptr failed");
             exit(1);
        }
   }
   /*
    * Allocate T_CALL structure to hold t_listen information.
    */
   lstn_info_ptr = (struct t_call *)t_alloc(lstn_fd,T_CALL,T_ALL);
   if(lstn_info_ptr == NULL){
        t_error("t_alloc of T_CALL packet failed");
        exit(1);
   }

   /*
    * Loop Accepting connections from the remote host.
    *
    */
   for(;;){
        int new_con_fd;     /* file descriptor for new connection. */
        struct t_bind * con_bind_ptr;
        struct sockaddr_in *sin_ptr;


        /*
         * Wait for a connection to arrive.
         */
        result = t_listen(lstn_fd,lstn_info_ptr);
        if(result < 0){
             t_error("t_listen failed");
             exit(1);
        }

        /*
         * Get a file descriptor for the new connection.
         */
        new_con_fd = t_open("/dev/tcp",O_RDWR, NULL);
        if(new_con_fd < 0){
             t_error("connection t_open failed");
             exit(1);
        }
        /*
         * Do a bind on the new file descriptor.
         */
        con_bind_ptr = (struct t_bind*)t_alloc(new_con_fd,T_BIND,T_ALL);
        if(con_bind_ptr == NULL){
             t_error("t_alloc of T_BIND packet for new con failed");
             exit(1);
        }
        con_bind_ptr->addr.len = sizeof(*sin_ptr);
        con_bind_ptr->qlen = 0;
        sin_ptr = (struct sockaddr_in*)con_bind_ptr->addr.buf;
        memset((char *)sin_ptr, NULL, sizeof(*sin_ptr));
        sin_ptr->sin_family = AF_INET;
```

```
result = t_bind(new_con_fd,con_bind_ptr,NULL);
if(result <0){
     t_error("new connection t_bind failed");
     exit(1);
}

/*
 * Free bind structure.
 */
result = t_free(con_bind_ptr,T_BIND);
if(result < 0){
     t_error("t_free con_bind_ptr failed");
     exit(1);
}

lstn_info_ptr->opt.len = 0;
/*
 * Associate connection with new file descriptor.
 * If two connections are pended, this fails.
 * This code should be prepared to do multiple
 * calls to t_listen before doing the t_accept.
 */
result = t_accept(lstn_fd,new_con_fd,lstn_info_ptr);
if(result < 0){
     t_error("t_accept failed");
     if(t_errno == TLOOK){
          result = t_rcvdis(lstn_fd,NULL);
          if(result < 0){
               t_error("t_rcvdis failed");
               exit(1);
          }
          result = t_close(new_con_fd);
          if(result < 0){
               t_error("t_close failed");
               exit(1);
          }
          continue;
     }
     exit(1);;
}
printf("Accepted a connection.\n");

/*
 * Loop echoing data back to network.
 * A real server should probably do a fork
 * to handle the new connection in parallel
 * with accepting new connections.
 */
for(;;) {
     int nbytes;     /* Holds number of bytes moved */
     char buffer[80];/* Holds data for/from network. */
     int flags;     /* Used with t_* calls. */

     /*
      * Get data from network.
      * Exit loop on disconnect.
      */
     flags = 0;
     nbytes = t_rcv(new_con_fd,buffer,sizeof(buffer),&flags);
     if(nbytes <0){
          t_error("error return from t_rcv");
          break;
     }
     /*
      * Ignore zero length reads.
      */
     if(nbytes == 0){
          continue;
     }
```

```
                /*
                 * Deliver data to network.
                 */
                flags = 0;
                nbytes = t_snd(new_con_fd,buffer,nbytes,&flags);
                if(nbytes < 0){
                     t_error("t_snd failed");
                     break;
                }
           } /* end loop to move network data. */
           /*
            * Handle end condition.
            */
           if(t_errno == TLOOK){
                result = t_look(new_con_fd);
                if(result < 0){
                     t_error("t_look failed");
                     exit(1);
                }
                switch (result) {
                case T_DISCONNECT: {
                     struct t_discon discon_info = {0};

                     result = t_rcvdis(new_con_fd, &discon_info);
                     if(result < 0){
                          t_error("t_rcvdis failed");
                     }
                     printf("Disconnect indication: (reason= %d) %s\n",
                               discon_info.reason,
                               strerror(discon_info.reason));
                     break;
                }
                case T_ORDREL: {
                     result = t_rcvrel(new_con_fd);
                     if(result < 0){
                          perror("t_rcvrel failed");
                          exit(1);
                     }
                     printf("Received orderly release.\n");

                     result = t_sndrel(new_con_fd);
                     if(result < 0){
                          perror("T_sndrel failed");
                     }
                     break;
                }
                default:
                     printf("Unknown result from t_look: %s\n",
                          result);
                } /* end switch */
           }
           /*
            * Close the data connection.
            */
           result = t_close(new_con_fd);
           if( result < 0){
                t_error("t_close failed");
                exit(1);
           }
      } /* end loop to process new connections */
}
```

A TLI-Based Client Program

Here is a sample client program that uses TLI routines to access TCP to establish a connection and transfer data. Invoke the program as follows:

% *program* [*hostname* [*service_name*]] ↵

The default *hostname* is **localhost**. The default *service_name* is **echo**, but *service_name* can also be a decimal number. Once the connection has been established, the program reads data from standard input and writes it to the network, and reads data from the network and writes it on standard output.

```
#include <stdio.h>       /* defines print functions and NULL */
#include <memory.h>      /* defines memcpy */
#include <netdb.h>       /* defines types for gethostbyname and getservbyname */
#include <sys/types.h>   /* defines u_long types used by next include file */
#include <netinet/in.h>  /* defines in_addr */
#include <arpa/inet.h>   /* defines inet_ntoa */
#include <fcntl.h>       /* defines O_RDWR, used in t_open call */
#include <sys/socket.h>  /* defines AF_INET used in t_bind structure */
#include <tiuser.h>      /* defines T_CALL and other T_* TLI things */
#include <errno.h>       /* defines EINTR used by select */
#include <macros.h>      /* defines max */

extern int t_errno;      /* Make TLI error codes available. */
/*
 * Declare function that does select.
 */
#define FROM_STDIN 0
#define FROM_NET 1
typedef int data_direction_type;
data_direction_type wait_data();

main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
     char *hostname;                  /* Holds name of remote host. */
     char *portname;                  /* Holds service name to call. */
     struct hostent* hostent_ptr;     /* Used to look up host address. */
     struct servent* service_ptr;     /* Used to lookup port number. */
     struct servent atoi_servent;     /* Used if getservbyname fails. */
     struct in_addr host_address;     /* Holds internet address to call. */
     int net_fd;                      /* Holds file descriptor for tcp. */
     int valid_connection;            /* Flag for data movement loop. */
     int result;                      /* Result of last system call. */
     /*
      * Check if arguments specify hostname and service to call.
      */
     if (argc > 1){
          hostname = argv[1];
     } else {
          hostname = "localhost";
     }

     if (argc >2){
          portname = argv[2];
     } else {
          portname = "echo";
     }

     if (argc >3 ){
          printf("Too many arguments\n");
```

```
     exit(1);
}
/*
 * Print banner message to say program has started.
 */
printf("Starting %s with hostname=%s, portname=%s\n",
               argv[0],hostname,portname);

/*
 * Look up hostname and port number to get remote address.
 * (AT&T has introduced the netdir facility, which includes
 * netdir_getbyname, as a transport independent way to
 * manipulate addresses.  Programmers who wish to write
 * transport independent code should consider using it.
 * See "Network Selection and Name-to-Address Mapping"
 * in "UNIX System V Release 4 Programmer's Guide:
 * Networking Interfaces" for more details.)
 */
hostent_ptr = gethostbyname(hostname);
if(hostent_ptr == NULL){
     printf("Can't resolve hostname %s\n",hostname);
     exit(1);
}
(void) memcpy((char *)&host_address,
               hostent_ptr->h_addr_list[0],
               sizeof(host_address));
service_ptr = getservbyname(portname,"tcp");
   if(service_ptr == NULL){
           service_ptr = &atoi_servent;
           service_ptr->s_port = htons(atoi(portname));
           if(service_ptr->s_port == 0){
          printf("Can't resolve portname %s\n",portname);
          exit(1);
     }
}
/*
 * Open tcp.
 */
net_fd = t_open("/dev/tcp",O_RDWR, NULL);
if(net_fd < 0){
     t_error("t_open failed");
     exit(1);
}
/*
 * Bind local port of connection.
 *
 * Ask Transport Provider to bind local port.
 */
{
     result = t_bind(net_fd, NULL, NULL);
     if(result < 0){
          t_error("t_bind failed");
          exit(1);
     }

}
```

```
/*
 * Connect to remote host.
 *
 * First declare, allocate, and initialize address information.
 * Then ask Transport Provided to establish the connection.
 */
{
     struct t_call *call_info_ptr;
     struct sockaddr_in *sin_ptr;

     call_info_ptr = (struct t_call *)t_alloc(net_fd,T_CALL,T_ADDR);
     if(call_info_ptr == NULL){
          t_error("t_alloc of T_CALL packet failed");
          exit(1);
     }

     call_info_ptr->addr.len = sizeof(struct sockaddr_in);
     sin_ptr = (struct sockaddr_in*)call_info_ptr->addr.buf;
     sin_ptr->sin_family = AF_INET;
     sin_ptr->sin_port = service_ptr->s_port;
     sin_ptr->sin_addr = host_address;

     printf("Connecting to address=%s (%X), port number=%d\n",
          inet_ntoa(host_address),
          ntohl(host_address.s_addr),
          ntohs(service_ptr->s_port));
     result = t_connect(net_fd,call_info_ptr,NULL);

     if(result == 0){
          printf("Connection established\n");
     } else {
          t_error("Connection NOT established");
     }

     {     int result2;
          result2 = t_free(call_info_ptr,T_CALL);
          if(result2 < 0){
               t_error("t_free call_info failed");
               exit(1);
          }
     }
}
/*
 * Loop moving data between network and stdin/stdout.
 */
while(result >= 0) {
     char buffer[80];      /* Holds data for/from network. */
     int flags;           /* Used with t_* calls. */

     switch (wait_data(fileno(stdin),net_fd)){
     case FROM_STDIN: {
          /*
           * Read from stdin.
           * On end of file, send orderly release.
           * Add a CR NL to talk with FTP or SMTP.
           * Write data to network.
           */
          if (gets(buffer) == NULL){
               printf("   Sending EOF    \n");
               result = t_sndrel(net_fd);
               if(result < 0){
                    t_error("t_sndrel failed");
               }
               break;
          }
          strcat(buffer,"\r\n");
```

```
                flags = 0;
                result = t_snd(net_fd,buffer,strlen(buffer),&flags);
                if(result < 0){
                     t_error("t_snd failed");
                     exit(1);
                }
                break;
           } /* end FROM_STDIN case */

           case FROM_NET: {
                /*
                 * Get data from network.
                 * Write data to stdout.
                 */
                flags = 0;
                result = t_rcv(net_fd,buffer,sizeof(buffer),&flags);
                if(result > 0){
                     fwrite(buffer,result,1,stdout);
                } else {
                     /*
                      * Ignore zero length reads.
                      */
                     if(result == 0){
                          continue;
                     } else {
                          t_error("t_rcv failed");
                     }
                }
                break;
           } /* end FROM_NET case */

           } /* end switch */

      } /* end data movement loop */

      /*
       * Try to print out message saying why the connection was closed.
       * Close connection.
       */
      if(t_errno == TLOOK){
           result = t_look(net_fd);
           if(result < 0){
                t_error("t_look failed");
                exit(1);
           }
           switch (result) {
           case T_DISCONNECT: {
                struct t_discon discon_info = {0};

                result = t_rcvdis(net_fd, &discon_info);
                if(result < 0){
                     t_error("t_rcvdis failed");
                }
                printf("Disconnect indication: (reason= %d) %s\n",
                          discon_info.reason,
                          strerror(discon_info.reason));
                break;
           }
```

```
            case T_ORDREL: {
                result = t_rcvrel(net_fd);
                if(result < 0){
                    perror("t_rcvrel failed");
                    exit(1);
                }
                printf("Received orderly release.\n");
                break;
            }
            default:
                printf("Unknown result from t_look: %s\n",
                    result);
            } /* end switch */
        }

        result = t_close(net_fd);
        if( result < 0){
            t_error("t_close failed");
            exit(1);
        }
}
/*
 * This function determines when either of two files has data
 * ready to read.  The return value indicates which file descriptor
 * has data.  If both files have data, the file descriptor
 * returned is arbitrary.

 * This function uses select(2) to wait for data to arrive.
 * TLI applications typically use poll(2) rather than select.
 * Using poll could increase portability.  Programmers should
 * use the function most appropriate for their application.
 * Using select as here illustrated also works on an AT&T 3B2
 * running System V release 4.
 */
data_direction_type wait_data(stdin_fd,net_fd)
int stdin_fd;
int net_fd;
{
    int result;
    fd_set ibits, obits, ebits;
    /*
     * Loop to ignore interrupts.
     */
    do {
        FD_ZERO(&ibits);
        FD_ZERO(&obits);
        FD_ZERO(&ebits);

        FD_SET(stdin_fd,&ibits);
        FD_SET(net_fd,&ibits);

        result = select(max(stdin_fd,net_fd)+1,&ibits,&obits,&ebits,0);
        if(result < 0){
            if(errno != EINTR){
                perror("select failed");
            }
        }
    } while( result <= 0);
```

```
        if(FD_ISSET(stdin_fd,&ibits)){
             return FROM_STDIN;
        } else {
             return FROM_NET;
        }
   }
```

A Socket-Based Client Program

Here is the same client program written to use sockets to establish a connection and transfer data. Invoke the program as follows:

% *program* [*hostname* [*service_name*]] ↵

As with the previous program, the default hostname is *localhost*. The default *service_name* is **echo**, but *service_name* can also be a decimal number.

```
#include <stdio.h>       /* defines print routines and NULL */
#include <memory.h>      /* defines memcpy */
#include <netdb.h>       /* defines types for gethostbyname and getservbyname */
#include <sys/types.h>   /* defines u_long types used by next include file */
#include <netinet/in.h>  /* defines in_addr */
#include <arpa/inet.h>   /* defines inet_ntoa */
#include <sys/socket.h>  /* defines AF_INET used in socket call */
#include <errno.h>       /* defines EINTR used by select */
#include <macros.h>      /* defines max */

/*
 * Declare routine that does select.
 */
#define FROM_STDIN 0
#define FROM_NET 1
typedef int data_direction_type;
data_direction_type wait_data();

main(argc,argv,envp)
int argc;
char *argv[];
char *envp[];
{
     char *hostname;                  /* Holds name of remote host. */
     char *portname;                  /* Holds service name to call. */
     struct hostent* hostent_ptr;     /* Used to look up host address. */
     struct servent* service_ptr;     /* Used to lookup port number. */
     struct servent atoi_servent;     /* Used if getservbyname fails. */
     struct in_addr host_address;     /* Holds internet address to call. */
     int net_fd;                      /* Holds file descriptor for tcp. */
     /*
      * Check if arguments specify hostname and service to call.
      */
     if (argc > 1){
          hostname = argv[1];
     } else {
          hostname = "localhost";
     }

     if (argc >2){
          portname = argv[2];
     } else {
          portname = "echo";
     }

     if (argc >3 ){
          printf("Too many arguments\n");
          exit(1);
     }
```

```
/*
 * Print banner message to say program has started.
 */
printf("Starting %s with hostname=%s, portname=%s\n",
               argv[0],hostname,portname);


/*
 * Find the internet address for the remote machine.
 */
hostent_ptr = gethostbyname(hostname);
if(hostent_ptr == NULL){
     printf("Can't resolve hostname %s\n",hostname);
     exit(1);
}
(void) memcpy((char *)&host_address,
               hostent_ptr->h_addr_list[0],
               sizeof(host_address));
/*
 * Find binary port number for named service.
 */
service_ptr = getservbyname(portname,"tcp");
   if(service_ptr == NULL){
           service_ptr = &atoi_servent;
           service_ptr->s_port = atoi(portname);
           if(service_ptr->s_port == 0){
          printf("Can't resolve portname %s\n",portname);
          exit(1);
     }
}

/*
 * Open tcp.
 */
net_fd = socket(AF_INET,SOCK_STREAM,0);
if(net_fd < 0){
     perror("socket failed");
     exit(1);
}

/*
 * Connect to remote host.
 *
 */
{    int result;
     struct sockaddr_in sin;
     int length;

     sin.sin_family = AF_INET;
     sin.sin_port = service_ptr->s_port;
     sin.sin_addr = host_address;

     printf("Connecting to address=%s (%X), port number=%d\n",
          inet_ntoa(host_address),
          host_address.s_addr,
          service_ptr->s_port);

     result = connect(net_fd,&sin,sizeof(sin));
     if(result < 0){
          perror("connect failed");
          exit(1);
     }
     printf("Connection established\n");
}
```

```
/*
 * Loop moving data between network and stdin/stdout.
 */
for(;;) {
     int nbytes;          /* Holds number of bytes moved */
     char buffer[80];     /* Holds data for/from network. */
     int result;

     switch (wait_data(fileno(stdin),net_fd)){

     case FROM_STDIN: {
          /*
           * Read from stdin.
           * On end of file, Shutdown network.
           * Add a CR NL to talk with FTP or SMTP.
           * Write data to network.
           */
          if (gets(buffer) == NULL){
               printf("   Sending EOF    \n");
               result = shutdown(net_fd,1);
               if(result < 0){
                    perror("shutdown failed");
               }
               break;
          }
          strcat(buffer,"\r\n");

          nbytes = send(net_fd,buffer,strlen(buffer),0);
          if(nbytes < 0){
               perror("send failed");
               exit(1);
          }
          break;
     } /* end FROM_STDIN case */
     case FROM_NET: {
          /*
           * Get data from network.
           * Exit loop on end of file.
           * Write data to stdout.
           */
          nbytes = recv(net_fd,buffer,sizeof(buffer),0);
          if(nbytes <0){
               perror("recv failed");
               exit(1);
          }
          if(nbytes == 0){
               printf("Received end of file\n");
               goto END_OF_FILE;
          }

          fwrite(buffer,nbytes,1,stdout);
          break;
     } /* end FROM_NET case */

     } /* end switch */

} /* end data movement loop */
```



```
     END_OF_FILE:
     /*
      * Close connection.
      */
     {
          int result;

          result = close(net_fd);
          if( result < 0){
               perror("close failed");
               exit(1);
          }
     }
}
/*
 * The following routine uses select to determine when
 * either of the two file descriptors has data.
 * The return value indicates which file descriptor has data.

 */

data_direction_type wait_data(stdin_fd,net_fd)
int stdin_fd;
int net_fd;
{
     int result;
     fd_set ibits, obits, ebits;
     /*
      * Loop to ignore interrupts.
      */
     do {
          FD_ZERO(&ibits);
          FD_ZERO(&obits);
          FD_ZERO(&ebits);

          FD_SET(stdin_fd,&ibits);
          FD_SET(net_fd,&ibits);

          result = select(max(stdin_fd,net_fd)+1,&ibits,&obits,&ebits,0);
          if(result < 0){
               if(errno != EINTR){
                    perror("select failed");
               }
          }
     } while( result <= 0);
     if(FD_ISSET(stdin_fd,&ibits)){
          return FROM_STDIN;
     } else {
          return FROM_NET;
     }
}
```

End of Chapter

Chapter 8
TCP/IP for AViiON Systems Manual Pages

Here are manual pages that are useful to a network programmer using the TCP/IP for AViiON Systems package. These manual pages are also available online through the **man**(1) command.

The following manual pages are included:

Table 8-1 List of TCP/IP Manual Pages

Name	Description
intro(6)	Introduces the TCP/IP protocol family
inet(6f)	Provides more detail about the TCP/IP protocol family
ip(6)	Internet protocol
loop(6)	Loopback interface
tcp(6)	Transport control protocol
udp(6)	User datagram protocol

NAME

intro – Communications Protocols introduction to networking facilities

INCLUDE FILES

#include <netinet/tcp.h>
#include <netinet/udp.h>
#include <netinet/ip.h.
#include <netinet/ip_icmp.h>
#include <net/if.h>

DESCRIPTION

This section briefly describes the DG/UX system networking facilities. Documentation in this section covers three areas: the Internet protocol family, the available protocols, and the network interfaces. The Internet protocol family is described on the **inet**(6F) manual page, whereas entries describing the protocols are on manual pages marked *6P*. Network interfaces are described on manual pages marked *6*.

The Internet family includes the Transmission Control Protocol (TCP), User Datagram Protocol (UDP), Internet Protocol (IP), and Internet Control Message Protocol (ICMP). These protocols are communications facilities implemented in the DG/UX system kernel that transfer information from user programs to the network and back. Programmers writing user-level programs can access TCP, IP, and UDP with the **socket**(2) family of system calls.

The Transmission Control Protocol (TCP) fits into the layered networking architecture just above IP. Application programs, such as remote terminal agents and file transfer agents, usually run on top of TCP, using its services.

TCP assures reliable end-to-end delivery of a data byte stream. TCP deals with user data copied to the protocol's buffers. It packages the data into segments and passes this information to IP, which then breaks the information into packets that can be easily transmitted across the network. IP then determines the next hop on a path through the network for the packet being transmitted and transfers the packet to the first host on the path. A gateway host would receive the packet and route it to the destination host. When packets arrive at the destination host, TCP reconstructs the entire message, checking to ensure that the data is complete and correctly ordered before sending it to application programs. If there is a problem, TCP requests that the message be retransmitted.

Like TCP, the User Datagram Protocol (UDP) fits into the layered networking architecture just above IP. It provides procedures for application programs to send messages to other programs with a minimum of protocol mechanism. UDP is a simple datagram protocol. Unlike TCP, it neither guarantees reliable delivery nor does it provide protection from duplicate messages.

The Internet Protocol (IP) is primarily concerned with getting a *datagram* to the next host on the route to the datagram's final destination. A datagram is a self contained package of data carrying sufficient information for hosts to deliver it to its destination. Since host availability changes, the packets that make up a complete message may have different routes and may end up at the destination out of their original order. The TCP layer is responsible for reordering the packets correctly. Some packets may be lost or garbled in transmission. IP frequently notifies higher level protocols when packets are lost or damaged, but sometimes does not.

The Internet Control Message Protocol (ICMP) is used to report errors in datagram processing. ICMP is an integral part of IP and must be implemented by every IP module. ICMP messages are sent to report problems in the communication environment, not to make IP a reliable protocol.

ADDRESSING

Associated with each protocol family is an address format. The following address formats are used by the system:

```
#define AF_UNIX    1    /* local to host (pipes) */
#define AF_INET    2    /* internetwork: UDP, TCP, etc. */
```

INTERFACES

Each network interface in a system corresponds to a path through which messages may be sent and received. A network interface usually has a hardware device associated with it, though certain interfaces such as the loopback interface, **loop (6)**, do not.

The following **ioctl** calls may be used to manipulate network interfaces. See *Programming with TCP/IP on the DG/UX™ System* for details.

SIOCSIFADDR
: Set interface address. Following the address assignment, the "initialization" routine for the interface is called.

SIOCGIFADDR
: Get interface address.

SIOCSIFBRDADDR
: Set interface broadcast address. This is address is used to send IP broadcast packets on broadcast capbable interfaces.

SIOCGIFBRDADDR
: Get interface broadcast address.

SIOCSIFDSTADDR
: Set the destination address for point-to-point network interfaces.

SIOCGIFDSTADDR
: Get interface destination address.

SIOCSIFMETRIC
: Set the interface routing metric. This information is used by routing applications.

SIOCGIFMETRIC
: Get the interface routing metric.

SIOCSIFNETMASK
: Set the interface subnetwork mask.

SIOCGIFNETMASK
: Get the interface subnetwork mask.

SIOCSIFFLAGS
: Set interface flags field. If the interface is marked as down, any processes currently routing packets through the interface are notified.

SIOCGIFFLAGS
: Get interface flags.

SIOCGIFCONF
Get interface configuration list.

SEE ALSO
socket(2), **ioctl**(2), *Programming with TCP/IP on the DG/UX™ System*.

NAME

inet – Communications Protocol Internet protocol family

INCLUDE FILES

#include <netinet/in.h>

DESCRIPTION

The Internet protocol family is a collection of protocols based on and including the Internet Protocol (IP), the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP). Each of these protocols uses the Internet address format. The Internet family provides protocol support for the SOCK_STREAM, SOCK_DGRAM, and SOCK_RAW socket types; the SOCK_RAW interface provides access to the IP.

ADDRESSING

Internet addresses are four-byte quantities, stored in network standard format. The include file **netinet/in.h** defines this address as a discriminated union.

Sockets bound to the Internet protocol family utilize the following addressing structure:

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

Sockets may be created with the address INADDR_ANY to affect wildcard matching on incoming messages.

PROTOCOLS

The Internet protocol family consists of the Internet Protocol (IP), Internet Control Message Protocol (ICMP), Transmission Control Protocol (TCP), and User Datagram Protocol (UDP). TCP is used to support the SOCK_STREAM abstraction, whereas UDP is used to support the SOCK_DGRAM abstraction. A raw interface to IP is available by creating an Internet socket of type SOCK_RAW. The ICMP is not directly accessible.

SEE ALSO

tcp(6P), **udp**(6P), **ip**(6P).

NAME

IP – Communications Protocol Internet Protocol

INCLUDE FILES

#include <sys/socket.h>
#include <netinet/ip.h>

SYNTAX

This is an example of how you would create an endpoint for the IP connection.

s = socket(AF_INET, SOCK_RAW, 0);

DESCRIPTION

IP is the network/internetwork layer protocol used by the Internet protocol family. It may be accessed through a raw socket when developing special-purpose applications. A raw socket can be opened only by the superuser.

IP sockets are connectionless, and are normally used with the **sendto** and **recvfrom** calls, though the **connect**(2) call may also be used to fix the destination for future packets (in which case the **read**(2) or **recv**(2) and **write**(2) or **send**(2) system calls may be used).

Outgoing packets must have an IP header prepended to them.

OPTIONS

IPPROTO_IP options recognized by IP:

IP_TX_OPTIONS	IP transmit options. When setting, the system will verify that the option string is well formed.
IP_RX_OPTIONS	IP receive options. When setting, the system will verify that the option string is well formed.
IP_TOS	IP Type Of Service.
IP_TTL	IP Time To Live. Number of routing hops a packet may make before reaching its destination.
IP_DONTFRAG	IP Dont Fragment flag. When non-zero, IP will try to send a packet without fragmenting. If a packet is too large to send without fragmenting, the packet is dropped.

SEE ALSO

connect(2), **send**(2), **recv**(2).
intro(6), **inet**(6f), *Programming with TCP/IP on the DG/UX™ System.*

NAME

loop – Communications Interface software loopback network interface

SYNOPSIS

loop

DESCRIPTION

The **loop**(7) interface is a software loopback mechanism that may be used for performance analysis, software testing, and/or local communication. By default, the loopback interface is accessible at address 127.0.0.1; this address may be changed with the SIOCSIFADDR **ioctl**. It is usually called **localhost** in the DG/UX system **/etc/hosts** file.

The **loop** interface will be configured into the DG/UX system only if the appropriate one-word entry is included in the system configuration file.

EXAMPLE

loop()

This entry in a system file will define the loop device.

DIAGNOSTICS

Use the **–i** switch with **netstat**(1C).

SEE ALSO

system(4).
intro(6), **inet**(6F).

NAME

TCP – Network Protocol Internet Transmission Control Protocol

INCLUDE FILES

#include <sys/socket.h>
#include <netinet/tcp.h>

SYNTAX

This is an example of how you would create an endpoint for the TCP connection:

```
s = socket(AF_INET, SOCK_STREAM, 0);
```

DESCRIPTION

Transmission Control Protocol (TCP) provides reliable, flow-controlled, two-way transmission of data. It is a byte-stream protocol used to support the SOCK_STREAM abstraction. TCP provides a per-host collection of port addresses on top of the standard Internet address format. Thus, each address is composed of an Internet address specifying the host and network, with a specific TCP port on the host identifying the peer entity.

Sockets utilizing the TCP are either active or passive. Active sockets initiate connections to passive sockets. By default TCP sockets are created active; only active sockets may use the **connect**(2) call to initiate connections. To create a passive socket, the **listen**(2) system call must be used after binding the socket with the **bind**(2) system call. Only passive sockets may use the **accept**(2) call to accept incoming connections.

Passive sockets may underspecify their location to match incoming connection requests from multiple networks. This technique, termed wildcard addressing, allows a single server to provide service to clients on multiple networks. To create a socket that listens on all networks, the Internet address INADDR_ANY must be bound to the socket. The TCP port may still be specified at this time; if the port is not specified, the system will assign one. Once a connection has been established, the socket's address is fixed by the peer entity's location. The address assigned to the socket is the address associated with the network interface through which packets are being transmitted and received.

OPTIONS

IPPROTO_TCP level options recognized by TCP:

TCP_NODELAY
: When the option value is non-zero, the system does not delay sending data to coalesce small packets. When the option value is zero, the system may defer sending data to coalesce small packets to conserve network bandwidth.

TCP_MAXSEG
: When set prior to a connect(2) call, TCP will use the option value to negociate the maximum size of TCP packets sent and received during the life of the connection. Values for the TCP Maximum Segment Size are between 1 and 65,535. This option is only valid prior to establishing a connection. The result of segment size negociation is less than or equal to the option value.

TCP_URGENT_INLINE
: This option has no effect in the DG/UX system. Use the SO_OOBINLIN socket level option.

TCP_PEER_ADDRESS
: Restricts the passive TCP endpoint to only accept connections initiated by the address supplied in the option value. The option value must contain a pointer to a sockaddr_in structure.

TCP_ACCEPT_QUEUE_LENGTH
: Sets the number of outstanding connections allowed at the TCP passive endpoint.

SEE ALSO

intro(6), **inet**(6F), *Programming with TCP/IP on the DG/UX™ System.*
getsockopt(2), **setsockopt**(2).

NAME

UDP – Communications Protocol Internet User Datagram Protocol

INCLUDE FILES

#include <sys/socket.h>
#include <netinet/udp.h>

SYNTAX

This is an example of how you would create an endpoint for the UDP connection:

```
s = socket(AF_INET, SOCK_DGRAM, 0);
```

DESCRIPTION

UDP is a simple, unreliable datagram protocol that is used to support the SOCK_DGRAM abstraction for the Internet protocol family.

UDP sockets are connectionless, and are normally used with the **sendto**(2) and **recvfrom**(2) calls. The **connect**(2) and **bind**(2) calls may also be used to fix the destination for future packets (in which case the **recv**(2) or **read**(2) and **send**(2) or **write**(2) system calls may be used). **Listen**(2) and **accept**(2) are not valid operations on datagram sockets.

SEE ALSO

send(2), **recv**(2), **sendto**(2), **recvfrom**(2).
intro(6), **inet**(6F), *Programming with TCP/IP on the DG/UX™ System.*

End of Chapter

Appendix A
Error Messages

This appendix describes the error messages that may appear when you use the socket family of system calls.

A **socket** call could fail for any of the following reasons:

Table A-1 Error Messages from the socket System Call

Error	Description
EAFNOSUPPORT	The specified address family is not supported in this version of the system. Check the address family specified in the **sockaddr_in** structure.
ESOCKTNOSUPPORT	The specified socket type is not supported in this address family.
EPROTONOSUPPORT	The specified protocol is not supported.
EMFILE	The per-process descriptor table is full.
ENOBUFS	No buffer space is available. The socket cannot be created. Try again; more memory may be available.
EPROTOTYPE	No default protocol could be found for the socket type. Check the socket type and specify the protocol.
ENOSR	The system is out of STREAMS resources, and could not create the protocol stream. Check the number of active sockets, reconfigure the kernel **NQUEUE** to be a larger number (each socket requires three queues).

The **setsockopt** or **getsockopt** could fail for any of the following reasons:

Table A-2 Error Messages from the setsockopt and getsockopt System Calls

Error	Description
EBADF	The argument *socket_des* is not a valid descriptor.
ENOTSOCK	The argument *socket_des* is not a socket.
ENOPROTOOOPT	The option is unknown at the level indicated.
EFAULT	The address to which *optval* points is not in a valid part of the process address space. For **getsockopt**, this error may also be returned if *optlen* is not in a valid part of the process address space.
EINVAL	The option value is invalid.
ENOBUFS	There are no internal buffers available.
EOPNOTSUPP	The option is unsupported.
EISCONN	The TCP option is invalid while in the connected state.
EACCES	Caller has inadequate privileges to set the option. Socket privilege is based on the euid of the process when the socket was created.

A **bind** call could fail for any of the following reasons:

Table A-3 Error Messages from the bind System Call

Error	Description
EBADF	*socket_des* is not an active valid descriptor.
EAFNOTSOCK	*socket_des* is not a socket.
EADDRNOTAVAIL	The address is not a valid address for the local machine.
EADDRINUSE	The address is already in use. Retry after a reasonable period.
EINVAL	The socket is already bound to an address. The size of the buffer pointed to by *name* is insufficient to form a valid address.
EFAULT	The *name* is not in a valid part of the user address space.
ENOBUFS	There are no internal buffers available.
EISCONN	The socket is already connected.
EPERM	The caller is not allowed to use the address.
EACCES	Caller has inadequate privilege to bind to a port in the reserved range. Socket privilege is based on the euid of the process when the socket was created.

The **shutdown** call could fail for any of the following reasons:

Table A-4 Error Messages from the shutdown System Call

Error	Description
EBADF	*socket_des* is not a valid descriptor.
ENOTSOCK	*socket_des* is not a socket.
ENOTCONN	The specified socket is not connected.
EINVAL	The *how* parameter is out of range.

Some of the more common errors returned when a **connect** call fails are as follows:

Table A-5 Error Messages from the connect System Call

Error	Description
EBADF	*socket_des* is not an active valid descriptor.
EAFNOTSOCK	*socket_des* is not a socket.
EADDRNOTAVAIL	The address is not a valid address for the local machine.
EAFNOSUPPORT	Addresses in the specified address family cannot be used with this socket.
EISCONN	This socket is already connected.
ETIMEDOUT	Connection establishment timed out without establishing a connection.
ECONNREFUSED	The attempt to connect was rejected by a foreign host.
ENETUNREACH	The network is not reachable from this host.
EADDRINUSE	This address is already in use. There is an existing connection using the same local and remote addresses.
EFAULT	The *name* parameter specifies an area outside the process address space.
EAGAIN	The socket is nonblocking and the connection cannot be completed before returning from the system call. The socket can be selected while it is connected by selecting it for writing.
ENOBUFS	There are no internal buffers available.
EINVAL	Invalid system call argument.
EALREADY	The connect operation has already been stated on this socket and has not yet finished.
EINTR	System call returned due to interrupt.
EOPNOTSUPPORT	The socket is in listen state.

The **listen** system call could fail for any of the following reasons:

Table A-6 Error Messages from the listen System Call

Error	Description
EBADF	The argument *socket_des* is not a valid descriptor.
EINVAL	The *backlog* parameter is a negative number.
ENOTSOCK	The argument *socket_des* is not a socket.
EOPNOTSUPP	The socket is not of a type that supports the **listen** operation.

The **accept** system call could fail for any of the following reasons:

Table A-7 Error Messages from the accept System Call

Error	Description
EBADF	The argument *socket_des* is not a valid descriptor.
ENOTSOCK	The argument *socket_des* references a file, not a socket.
EOPNOTSUPP	The referenced socket is not of type SOCK_STREAM.
EFAULT	The *from* or *fromlen* parameter is not in a writable part of the user address space.
EAGAIN	The socket is marked nonblocking and no connections are present to be accepted. Use select for reading.
EINVAL	The socket is not in the listen state.
EINTR	The call was interrupted by a signal.
EMFILE	Too many file descriptors were opened by the process.
ECONNABORTED	The listening socket was marked unreadable by the system. This is usually caused by a network failure.

The **send** system call could fail for any of the following reasons:

Table A-8 Error Messages from the send System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid file descriptor.
ENOTSOCK	The argument *socket_des* is not a socket.
EFAULT	The *buf* parameter points to an invalid portion of the process address space.
EMSGSIZE	The socket requires that messages be sent atomically, and the size of the message made this impossible.
EAGAIN	The socket is marked nonblocking and the requested operation would block.
EOPNOTSUPP	The *flags* argument included the MSG_OOB flag applied to a UDP socket.
ENOTCONN	The socket is an unconnected UDP socket.
EINTR	The call was interrupted by a signal.
EPIPE	An established connection on a SOCK_STREAM socket was closed by the remote peer.

The **recv** system call could fail for any of the following reasons:

Table A-9 Error Messages from the recv System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid file descriptor.
ENOTCONN	The socket is not connected.
ENOTSOCK	The argument *socket_des* is not a socket.
EAGAIN	The socket is marked nonblocking and the receive operation would block.
EINTR	The call was interrupted by a signal.
EFAULT	The data was specified to be received into a non-existent or protected part of the process address space.
EINVAL	Invalid argument.
EOPNOTSUPP	The *flags* argument included the MSG_OOB flag applied to a UDP socket.

The **sendto** system call could fail for any of the following reasons:

Table A-10 Error Messages from the sendto System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid descriptor.
ENOTSOCK	The argument *socket_des* is not a socket.
EFAULT	The *msg*, *to*, or *tolen* parameter points to an invalid portion of the process address space.
EMSGSIZE	The socket requires that messages be sent atomically, and the size of the message made this impossible.
EAGAIN	The socket is marked nonblocking and the receive operation would block.
EINTR	The call was interrupted by a signal.
EFAULT	One of the pointer parameters specified a non-existent or protected part of the process address space.
EISCONN	Cannot use **sendto** with connected socket.

The **recvfrom** system call could fail for any of the following reasons:

Table A-11 Error Messages from the recvfrom System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid descriptor.
ENOTSOCK	The argument *socket_des* is not a socket.
EAGAIN	The socket is marked nonblocking and the receive operation would block.
EINTR	The call was interrupted by a signal.
EFAULT	One of the pointer parameters specified a non-existent or protected part of the process address space.
EINVAL	An invalid argument has been specified.

The **sendmsg** system call could fail for any of the following reasons:

Table A-12 Error Messages from the sendmsg System Call

Error	Description
`EBADF`	The argument 2socket_des is not an active valid descriptor.
`ENOTSOCK`	The argument *socket_des* is not a socket.
`EFAULT`	The *buf* parameter points to an invalid portion of the process address space.
`EMSGSIZE`	The socket requires that messages be sent atomically, and the size of the message made this impossible.
`EAGAIN`	The socket is marked nonblocking and the requested operation would block.
`ENOTCONN`	The socket is an unconnected UDP socket.
`EISCONN`	The socket is connected and cannot accept a destination address.
`EINTR`	The call was interrupted by a signal.

The **recvmsg** system call could fail for any of the following reasons:

Table A-13 Error Messages from the recvmsg System Call

Error	Description
`EBADF`	The argument *socket_des* is not an active valid descriptor.
`ENOTSOCK`	The argument *socket_des* is not a socket.
`EAGAIN`	The socket is marked nonblocking and the requested operation would block.
`EINTR`	The call was interrupted by a signal.
`EFAULT`	One of the pointer parameters specified a non-existent or protected part of the process address space.
`EMSGSIZE`	Too many entries in the I/O array.

The **readv** system call could fail for any of the following reasons:

Table A-14 Error Messages from the readv System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid descriptor.
EINTR	The call was interrupted by a signal.
EFAULT	Part of the **iovec** points outside the process's allocated address space.
EINVAL	The **iovcnt** parameter was invalid, or the length of one of the values in the **iovec** array was negative, or the sum of the lengths in the **iovec** array overflowed a 32-bit integer.

The **writev** system call could fail for any of the following reasons:

Table A-15 Error Messages from the writev System Call

Error	Description
EBADF	The argument *socket_des* is not an active valid descriptor.
EPIPE	An attempt is made to write to a pipe not open for writing or a socket of type SOCK_STREAM that is not connected to a peer socket.
EFBIG	An attempt was made to write a file that exceeds the process's file size limit or the maximum file size.
EINTR	The call was interrupted by a signal.
EFAULT	Part of the **iovec** points outside the process's allocated address space.
EINVAL	The **iovcnt** parameter was invalid, or the length of one of the values in the **iovec** array was negative, or the sum of the lengths in the **iovec** array overflowed a 32-bit integer.

End of Appendix

Appendix B
Using the Network Library Routines

Although TCP/IP for AViiON Systems currently supports only the DARPA standard Internet protocols, the network library routines are flexible. They allow communication protocols to use the same interfaces when accessing network-related databases, regardless of protocol. If new protocols become available, they should differ only in the values returned. Because the values returned are usually supplied to the system, the communication protocol and naming conventions in use can be hidden from users.

The network library routines are designed to aid in mapping hostnames, addresses, and other information that is necessary to allow communication throughout a network. For a client and server to communicate, two levels of mapping must be done to locate a service on the remote host: (1) a service is assigned a name convenient for users (for example, the login server on host A); and (2) the assigned name and the name of the peer host are translated into network addresses. The DG/UX system provides standard routines for mapping the following items to one another:

- Hostnames to network addresses
- Network names to network numbers
- Protocol names to protocol numbers
- Service names to port numbers
- Appropriate protocols for communicating with the server process

Include the file **netdb.h** when using the routines.

Mapping Hostnames to Network Addresses

Three library routines aid in mapping hostnames to network addresses. These routines are **gethostent**(3N), **gethostbyname**(3N), and **gethostbyaddr**(3N). The **gethostent** routine is the primitive upon which **gethostbyname** and **gethostbyaddr** are built. It extracts a line from the network hosts database and returns a pointer to a **hostent** structure. The network hosts database could be provided by **/etc/hosts**, the Network Information Service (NIS), or the domain name system. The routines **gethostbyname** and **gethostbyaddr** use the **hostent** structure.

The **gethostbyname** routine takes a hostname and returns a pointer to a **hostent** structure (see below). The **gethostbyaddr** routine maps host addresses into a **hostent** structure. Since a host can have many addresses that have the same name, **gethostbyname** returns the first matching entry in the network hosts database.

The hostname to network address mapping is represented by the **hostent** structure, which contains the following fields:

```
struct hostent {
       char      *h_name;        /* official name of host */
       char      **h_aliases;    /* alias list */
       int       h_addrtype;     /* host address type */
       int       h_length;       /* length of address */
       char      *h_addr;        /* address */
};
```

The members of this structure are as follows:

h_name A pointer to the official name of the host.

h_aliases A pointer to a null-terminated array of alternate names for the host.

h_addrtype The type of address being returned; currently always **AF_INET**.

h_length The length, in bytes, of the address.

h_addr A pointer to the network address for the host. Host addresses are returned in network byte order (see "Additional Routines" later in this appendix for a description of network byte order).

If the entry returned is not the one wanted, you can use the lower level routine **gethostent**. Because **gethostbyname** returns only the first entry by that name, you will have to use **gethostent** for subsequent entries by that name.

For example, to obtain a **hostent** structure for a host on a particular network, you could use the following routine (this routine considers only Internet addresses):

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

struct hostent*
get_host_by_name_and_net(name, net)
        char  *name;
        int   net;

        struct hostent *host_ptr;
        char           **cp;

        sethostent(0);

       /*
        * We'll look at the host file one entry at a time until
        * we either run out of entries or find one that matches
        * name and net.
        */

       while  ((host_ptr + gethostent()) != NULL)  {

       /*
        * We're interested only in Internet addresses...
        */

       if(host_ptr->h_addrtype != AF_INET) {
          continue;

       }

       /*
        * If name matches either the h_name or any of
        * the h_aliases for this entry we'll goto
        * found: and decide if the net matches also.
        * If we don't match, we'll loop up and get
        * another host entry.
        */

       if  (strcmp(name, host_ptr->h_name)){
           for (cp = host_ptr->h_aliases; cp && *cp; cp++) {
               if (!strcmp(name, *cp) == 0) {
                   goto found;
               }
          }
          continue;
     }
     /* If our net matches, we can go ahead and return the
      * pointer to the host structure.
      */

      found:
            if(inet_netof(*(struct in_addr *)host_ptr->h_addr)) == net)
                   break;
        }

      /*
       * We didn't match anything, and since the last gethostent
       * failed, host_ptr is a NULL pointer.  Returning it will
       * let the caller know we failed.
       */

    endhostent();
    return(host_ptr);
}
```

The routines **gethostent** and **endhostent** open and close the network hosts database. The **sethostent** routine also rewinds the file. The **inet_netof** routine returns the network number of an Internet address.

Mapping Network Names to Network Numbers

Three library routines are provided to aid in mapping network names to network numbers. These routines are **getnetent**(3N), **getnetbyname**(3N), and **getnetbyaddr**(3N). The **getnetent** routine is the primitive upon which **getnetbyname** and **getnetbyaddr** are built. It extracts a line from the network database file and returns a pointer to a **netent** structure. The network database could be provided by **/etc/networks** or by NIS. The **getnetent** routine can be used to develop new routines for extracting a specific entry from the network database.

The **netent** structure returns the official name of the network and its public aliases. It also returns an address type and network number. The **netent** structure is as follows:

```
struct netent {
        char     *n_name;       /* official name of net */
        char     **n_aliases;   /* alias list */
        int      n_addrtype;    /* net address type */
        int      n_net;         /* network # */
};
```

The members of this structure are as follows:

n_name A pointer to the official name of the network.

n_aliases A pointer to a null-terminated list of alternate names for the network.

n_addrtype The type of the network number returned; currently only **AF_INET**.

n_net The network number. Network numbers are returned in network byte order.

The **getnetbyname** routine takes a network name and returns a pointer to a **netent** structure. The **getnetbynumber** routine takes a network number and returns a pointer to **netent** structure. Since a host can have many names that have the same number, **getnetbyname** returns the first matching entry in the network database.

Mapping Protocol Names to Protocol Numbers

Three library routines aid in mapping protocol names to protocol numbers. These routines are **getprotoent**(3N), **getprotobyname**(3N), and **getprotobynumber**(3N). The **getprotoent** routine is the primitive on which **getprotobyname** and **getprotobynumber** are built. It extracts a line from the protocol database and returns a pointer to the **protoent** structure. The protocol database could be provided by **/etc/protocols** or by NIS. You can use **getprotoent** to develop new routines for extracting a specific entry from the protocol database.

The **getprotobyname** and **getprotobynumber** routines also return a **protoent** structure. The **protoent** structure contains the official name of the protocol, the protocol's public aliases, and a port number. The **protoent** structure is as follows:

```
struct protoent {
        char        *p_name;       /* official protocol name */
        char        **p_aliases;   /* alias list */
        int         p_proto;       /* protocol to use */
};
```

The members of the **protoent** structure are as follows:

p_name A pointer to the official name of the protocol.

p_aliases A pointer to a null-terminated list of alternate names for the protocol.

p_proto The protocol number.

The **getprotobyname** routine takes a protocol name and returns a pointer to a **protoent** structure. The **getprotobynumber** routine maps network numbers into a **protoent** structure. Since a protocol can have many names that have the same number, **getprotobyname** returns the first matching entry in the protocol database.

Mapping Service Names to Port Numbers

Three library routines aid in mapping service names to port numbers. These routines are **getservent**(3N), **getservbyname**(3N), and **getservbynumber**(3N). The **getservent** routine is the primitive upon which **getservbyname** and **getservbynumber** are built. It extracts a line from the services database and returns a pointer to the **servent** structure. The services database could be provided by **/etc/services** or by NIS. You can use **getservent** to develop new routines for extracting a specific entry from the services database.

The **getservbyname** and **getservbynumber** routines return a **servent** structure, which is as follows:

```
struct servent {
       char       *s_name;      /* official protocol name */
       char       **s_aliases;  /* alias list */
       long       s_port;       /* port service resides at */
       char       *s_proto;     /* protocol to use */
};
```

The members of this structure are as follows:

s_name A pointer to the official name of the service.

s_aliases A pointer to a null-terminated list of alternate names for the service.

s_port The port number at which the service resides. Port numbers are returned in network byte order (see "Additional Routines" later in this chapter for a description of byte order).

s_proto A pointer to the name of the protocol to use when contacting the service.

The **getservbyname** routine maps service names to a **servent** structure by specifying a service name and a protocol. Although the protocol must be specified, it can be specified as NULL. For example, the following call returns the service specification for a chargen server using any protocol:

```
#include <netdb.h>

struct servent *server_ptr

     server_ptr = getservbyname("chargen", NULL);
```

On the other hand, the following call returns only the TELNET server that uses the TCP protocol:

```
server_ptr = getservbyname("chargen", "tcp");
```

The **getservbyport** routine functions much like the **getservbyname** routine. It maps a service port number to a **servent** structure by specifying a service port number and a protocol. The protocol must be specified, but can be supplied as NULL. For example, the following call returns the service specification for a chargen server using any protocol:

```
server_ptr = getservbyport("19", NULL);
```

In contrast, the call returns only the chargen server that uses the TCP protocol:

```
server_ptr = getservbyport("19", "tcp");
```

Port number specifications are listed in the services database.

Using Additional Routines

Because of the support routines described so far, application programs should rarely have to deal directly with addresses. Services, therefore, can be developed as independently as possible from the network on which they are used.

In addition to the address-related database routines, routines to handle byte-swapping of network addresses and values are also provided. These routines are defined in the file **/usr/include/netinet/in.h**. The DG/UX system expects addresses to be supplied in network order. Some machines, however, reverse this order. Thus, programs are sometimes required to byte-swap addresses.

Table B-1 summarizes the routines for handling byte-swapping of network addresses and values.

Table B-1 Routines for Byte-Swapping Network Addresses

Routine	Meaning	What it Does
htonl(val)	host to network long	Convert 32-bit value from host to network byte order.
htons(val)	host to network short	Convert 16-bit value from host to network byte order.
ntohl(val)	network to host long	Convert 32-bit value from network to host byte order.
ntohs(val)	network to host short	Convert 16-bit value from network to host byte order.

NOTE: We recommend that these routines be used in all programs that use the network. While these routines do not affect programs on Data General equipment, they are essential in arranging the byte order for programs on some other systems. Using these routines increases the portability of programs across systems with different architectures.

Users should encounter the byte-swapping problem only when interpreting network addresses. For example, to print out an Internet port, the following code is required:

```
struct servent *sp;
printf("port number %d\n", ntohs(sp->s_port));
```

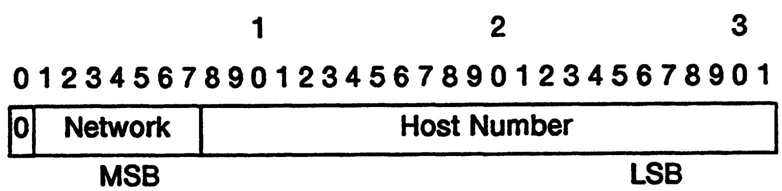
End of Appendix

Glossary

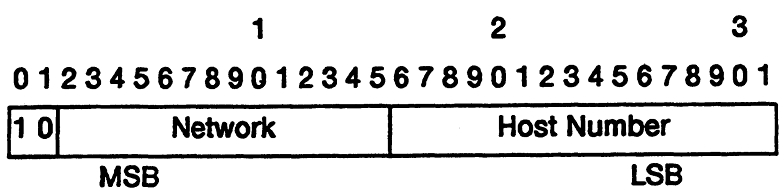
address class

A class of address recognized on the Internet network. There are three classes available: Class A, Class B, and Class C. The Defense Data Network — Network Information Center (NIC) assigns network numbers, and individual organizations assign host numbers. Conceptually, the 32-bit address consists of a class identifier, a network number, and a host number. The format of the address depends on the class as the diagram shows:

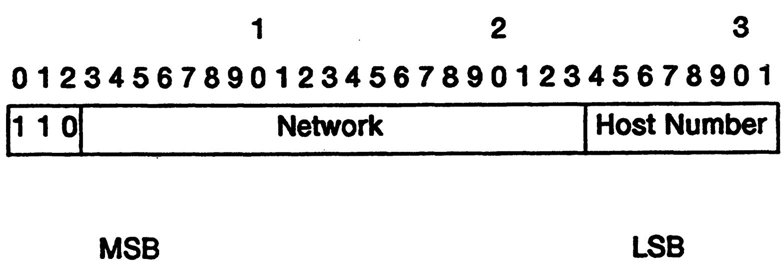
Class A Internet Address



Class B Internet Address



Class C Internet Address



See also Internet address.

Address Resolution Protocol (ARP)

A kernel-level protocol used to map an Internet address to a physical address (Ethernet address). ARP can be used only across a single physical network and can be used only over networks that support broadcasts over the communications medium.

architecture
See network architecture.

ARPANET A wide-area network funded by the Defense Advanced Projects Research Agency (DARPA). The ARPANET served as the basis for early networking research and as a center backbone during the development of the Internet network.

association Binds communicating processes to one another over a network. An association is composed of a local address bound to a local port and a remote address bound to a remote port.

Berkeley Software Distribution (BSD)
A general term for the version of UNIX created at the University of California at Berkeley. When DARPA first made TCP/IP widely available, they decided to encourage university researchers to use the protocols. Most university computer science departments were running BSD UNIX at that time. DARPA funded a company to implement TCP/IP under BSD UNIX and funded UC Berkeley to integrate them with its distribution. TCP/IP first appeared in the BSD 4.2 distribution, and was revised in the 4.3 and 4.3 Tahoe revisions. BSD UNIX offered more than the basic TCP/IP protocols; it also offered UNIX-like utilities to use the protocols, for example, **rcp**. BSD UNIX also provided the socket abstraction that allows application programmers to access the protocols. Data General's implementation of TCP/IP is based on the BSD implementation.

binding Assign a name to a socket so that a process can use the socket to communicate with another process. See the **bind**(2) manual page for details.

broadcast To send the same message to all systems on a network at the same time.

broadcast address
An Internet address used for all hosts on the network. Any Internet address with a host portion that consists of all 1s is reserved for broadcast for systems compatible with BSD 4.3. Any Internet address with a host portion that consists of all 0s is reserved for broadcast for systems compatible with BSD 4.2. On networks where both BSD 4.2 and BSD 4.3 software is used, a host portion of all 0s works best.

BSD *See* Berkeley Software Distribution (BSD).

client

1. An operating system (OS) client.
2. An executing program that sends a request to a server for services and waits for a response. Thus, there are network clients, Network Information Service (NIS) clients, Network File System clients, and clients of the domain name system.

client-server model
Refers to a pattern of interaction among application programs that communicate over the network. Server programs provide services (such as remote login facilities) and client programs consume them. The relationship describes which program initiates a connection, sends data first, and controls the communication link. *See also* client, server.

connection The path between two processes that provides reliable, stream-oriented, process-to-process delivery service.

connection-oriented communication
Characteristic of the reliable, stream-oriented, process-to-process service offered by the Transmission Control Protocol (TCP). System calls are used to connect to and communicate with remote processes.

connectionless communication
Characteristic of the packet delivery service offered by the Internet Protocol or the User Datagram Protocol. Treats each packet or datagram as a separate entity that contains the source and destination address. Connectionless services may drop or duplicate packets or deliver them out of sequence.

DARPA *See* ARPANET.

daemon An unattended background process, often perpetual, that performs a system-wide public function; for example, **inetd**. *See also* server.

datagram A self-contained package of data carrying the necessary information to route itself from source to destination. It is the unit of transmission in the IP protocol. To cross a particular network, a datagram is encapsulated inside a packet.

datagram socket
Sends datagrams in and receives datagrams from both directions simultaneously, preserving logical breaks in the data. Data travels in complete packets rather than streams or bytes. The packets may arrive out of order or may fail to be delivered. This service is known as connectionless communication. *See also* socket, User Datagram Protocol.

Defense Data Network (DDN)
Used loosely to refer to the MILNET and ARPANET networks and the TCP/IP protocols that these networks use.

Defense Data Network — Network Information Center (NIC)
The part of the Defense Data Network (DDN) with the authority to assign Internet addresses.

device driver
A set of software used to manage a peripheral device. For example, **hken** is a device driver used to manage a V/Ethernet 3207 Hawk Local Area Network (LAN) Controller. *See also* controller.

Ethernet A type of local area network developed by the Xerox Corporation. An Ethernet network consists of cable and interface hardware that connects hosts. Only one host can use the network at a time. Hosts send out packets of information over the network whenever they detect that other hosts are not using it.

Ethernet address
A number that identifies a specific host on an Ethernet-based local area network. Ethernet addresses are set on a host during manufacture with hardware switches and are guaranteed to be unique.

file system *See* logical disk-file system.

File Transfer Protocol (FTP)
A user-level protocol accessed through the **ftp** command. FTP allows you to transfer files from one host to another. The File Transfer Protocol uses TCP as the transport level protocol.

host A computer that is configured to share resources with other computers in a network. Refers to any computer: stand-alone, OS server, or OS client. *See also* local host, remote host.

host ID A unique number that identifies the host. In the DG/UX system, the host ID typically is the host's most commonly used Internet address. *See also* host number.

hostname A string that represents a host. Hostnames are associated with Internet addresses in the **/etc/hosts** file.

host number
The host portion of a computer system's Internet address. *See also* address class, Internet address.

Internet The collection of networks and gateways, including the ARPANET and the MILNET, that use the TCP/IP protocol suite and function as a single, cooperative virtual network.

Internet address
A unique 32-bit number that identifies a specific host on the Internet. Internet addresses are expressed in dot notation, and have the general form *a.b.c.d*, where each letter represents eight bits, or one octet. One part of the Internet address represents the network number, and one part represents the host number. There are three classes of Internet address: Class A, Class B, and Class C. The difference among classes depends on the length of the network number: Class A network numbers are one octet long, Class B network numbers are two octets long, and Class C network numbers are three octets long. Network numbers are assigned by the Defense Data Network — Network Information Center, or simply the NIC. *See also* address class.

Internet Control Message Protocol (ICMP)
The part of IP that handles error and control messages. Gateways and hosts use ICMP to tell the source of datagrams about problems

delivering the datagrams. ICMP also allows a host to test whether a destination is reachable and responding.

Internet Protocol (IP)
A kernel-level protocol that defines unreliable, connectionless delivery of datagrams. An IP datagram contains the addresses of its source and destination, and the data transmitted. Connectionless service means that the protocol treats each datagram as a separate entity; the protocol can deliver packets out of sequence, or can drop packets. IP defines the exact format of data as it travels through a network, but delivery of data is not guaranteed.

interface A common boundary between two devices, programs, or systems. More specifically, an interface consists of the types and forms of messages that each layer of a network architecture uses to communicate with the layer above or below it. An interface gives two systems that handle information differently a way to interact.

internetwork
A technology that allows the interconnection of disparate physical networks into a coordinated functional unit. An internetwork (for example, the Internet) accommodates different networking hardware by adding physical connections and by implementing a standard set of protocol conventions.

interoperability
The ability of diverse computing systems to cooperate in solving computational problems.

kernel The nucleus of the DG/UX operating system. It controls access to the computer, manages the computer's memory, maintains the file system, and allocates the computer's resources among users. The kernel is sometimes described as the DG/UX system proper; resident code that implements the system calls.

local area network (LAN)
A network within a small area or a common environment, such as within a building. Ethernet is a type of LAN.

local host The computer your terminal is connected to. A local host can send and receive information from a remote host through connection-oriented or connectionless communication.

local port *See* port.

logical network
A network that may consist of one or more physical networks or may be a subdivision of a single physical network. You set up a logical network through the addressing scheme you use.

mapping Associating the elements of two different representations of a system so that a correspondence exists between the two systems. Every element in one system can be mapped to an element in the other system.

MILNET Originally part of the ARPANET, the MILNET was partitioned in 1984 to give military installations reliable network service while the ARPANET continues to be a research network. The MILNET uses the same hardware and protocols as the ARPANET.

multiplexing
Using a device to handle several similar but separate operations simultaneously by alternating attention among them.

name server
Part of the domain name system. The name server runs as a daemon process called **named**, and responds to queries by consulting its database. If the answer is not in its database and the name server acts recursively, the name server forwards a query to other name servers. For more information, see *Managing TCP/IP on the DG/UX™ System*.

netmask *See* subnet mask.

network The hardware and software that constitute the interconnections between computer systems, permitting electronic communication between the systems and associated peripherals. Networking for computer systems means sending data from one system to another over some medium (such as coaxial cable or phone lines). Common networking services include file transfer, remote login, and remote execution.

network architecture
The set of layers, interfaces, and protocols that govern communication over a network.

Network File System (NFS)
A service that allows many users to share file systems over a network. For more information, see *Managing ONC™/NFS® and Its Facilities on the DG/UX™ System*.

Network Information Service (NIS)
A service that maintains a set of databases about hosts, networks, and services for an entire network. For more information, see *Managing ONC™/NFS® and Its Facilities on the DG/UX™ System*.

network number
The network portion of an Internet address. The length of the network number depends on the address class. *See also* address class, Internet address.

NIC *See* Defense Data Network — Network Information Center.

NIS domain

1. A named set of NIS maps, which are set of keys and associated values.

2. A logical grouping of hosts in a NIS environment. Each host in a domain relies on the same servers for certain resource sharing and

security services. Each domain has one master and zero or more slave servers.

For more information, see *Managing ONC™/NFS® and Its Facilities on the DG/UX™ System.*

NIS server A computer that creates and maintains the following information for hosts in a NIS domain: advertised resources, hostnames and passwords, names and addresses for name servers of other domains (optional), host user and group information used for ID mapping (optional). For more information, see *Managing ONC™/NFS® and Its Facilities on the DG/UX™ System.*

Open Network Computing/Network File System (ONC™/NFS®)
A package that consists of the Network Information Service (NIS), NFS, and other networking facilities. For more information, see *Managing ONC™/NFS® and Its Facilities on the DG/UX™ System.*

OS server

1. A host that is willing to share resources with another host in a NFS environment.
2. A host providing disk space for operating system software. OS servers can be stand-alone, homogeneous or heterogeneous.

See also server.

packet Refers to the unit of data sent across a packet-switching network. The format of a packet is typically defined by the protocol.

peer processes
Processes on different computer systems that run at the same level in the communications hierarchy. That is, both processes run at the user-level or both run at the kernel-level. Peer processes use protocols to exchange data that their peer can understand.

physical network
The hardware (computers, communication controllers, media) that makes up a network.

port

1. The point of connection between a device, such as a communications controller, and the CPU.
2. The number used to determine which process on a host receives information. Networking software uses ports to allow processes on different computers to communicate. A single process can use several ports, using each to communicate with the port of a different remote process. The *local port* exists on the local host. The *remote port* exists on the remote host.

process A program in execution. When a process is being executed by several people simultaneously, there are several processes, but only one program. Each process is cataloged in the system's process table.

protocol A set of rules that govern the transfer of data and communication between two or more devices in a network. Includes rules for handshaking and line discipline.

raw socket Allows access to the underlying communication protocols (such as IP) that support higher level protocols. These sockets normally send information in datagrams, but their characteristics depend on the interface provided by the protocol. *See also* socket.

remote host
The other computer that a local host sends information to and gets information from though connection-oriented or connectionless communication.

remote port
See port.

Request for Comments (RFC)
A series of technical papers that contain surveys, measurements, techniques, specifications, and proposed and accepted Internet protocol standards. RFCs are available from the Defense Data Network — Network Information Center (known as the NIC), SRI International, Menlo Park, CA, 94025.

Reverse Address Resolution Protocol (RARP)
A kernel-level protocol used by a diskless system at startup to find its Internet address. The diskless system broadcasts a request that contains its Ethernet address and the server responds by sending the machine its Internet address.

RFC *See* Request for Comments (RFC).

route The path that network traffic takes from its source to its destination.

router A box (a computer or special equipment) that forwards packets of a particular protocol type (for example, IP) from one network to another. It is possible to use a computer as a router as long as it has more than one network interface, and its software is prepared to forward datagrams.

sendmail A command that implements the Simple Mail Transfer Protocol (SMTP), which allows the dispatch of mail messages. The **sendmail** command uses TCP as the transport level protocol.

server

1. An OS server.
2. A server process that provides network services to a client process, for example, **telnetd**.
3. A Network Information Service (NIS) server, which provides NIS database information to NIS clients.

4. A Network File System (NFS) server, which provides file system access to remote NFS clients.

5. A name server, which is part of the domain name system. For more information about DNS, see *Managing TCP/IP on the DG/UX System*.

shell A command interpreter and programming language that acts as an interface to the UNIX® system. As a command interpreter, the shell accepts commands and acts on them. As a programming language, the shell's features include flow control and string-valued variable definition. When you log in to the system, you acquire a login shell. In this shell, you can run another shell program, which becomes a subshell to your login shell. The two most common shells are the Bourne shell and the C shell. For more information, see *Using the DG/UX™ System*.

socket An abstraction representing a communications endpoint. A socket is implemented as a mechanism in the kernel to act as an interface between processes on different computers. There are three types of sockets available with TCP/IP for AViiON Systems: stream sockets, datagram sockets, and raw sockets.

socket address
In the Internet domain, the concatenation of an Internet addresses with a port number. Also called a connection endpoint. *See also* socket.

stream A full duplex, processing and data transfer path in the kernel. It implements a connection between a driver in kernel space and a process in user space, providing a general character I/O interface for the user processes.

STREAMS A full-duplex character processing mechanism that regularizes the kernel's character I/O facilities. It defines a standard interface between modules, provides tools for managing buffers that modules have in common, and standardizes ways to pass control between modules.

stream socket
Used to access TCP to send and receive data in continuous streams of bytes without logical breaks or duplication. Data can pass through the socket in both directions simultaneously, guaranteeing delivery in the original order in which the data is sent. *See also* socket.

subnet An extension of the Internet addressing scheme that allows a site to use a portion of its host address field as a subnet field. Outside the site, routing divides the destination address into an Internet portion and a local portion. Routers and hosts inside a site that uses subnets interpret the local portion of the address by dividing it into a physical network portion and a host portion. Thus, a site can present a single local network number to the world, but still maintain distinct physical networks and routing internally.

subnet mask
A bit mask, associated with the network interface, that corresponds with the bits of the Internet address that determine the network portion of the address.

TELNET A user-level protocol accessed through the **telnet** command. The TELNET protocol allows a user on one host to interact with a remote host as if the terminal is directly connected to the remote host. TELNET uses TCP as the transport level protocol.

Transmission Control Protocol (TCP)
A kernel-level protocol that defines reliable, end-to-end delivery of datagrams. TCP is connection-oriented because it establishes a connection between communicating hosts before transmitting data. TCP allows a process on one host to send data to a process on another through a byte stream. TCP uses IP to transmit information along an Internet network. TCP messages include a protocol port number that allows the sender to distinguish multiple programs on the remote host.

Transport Layer Interface (TLI)
A library of routines that uses STREAMS mechanisms to access transport-level services in the kernel.

Trivial File Transfer Protocol (TFTP)
A user-level protocol accessed through the **tftp** command — allows file transfer with minimal capability and overhead. The **tftp** command depends on the UDP protocol.

During first stage boot with the AViiON station, the boot program, once it determines its Internet address, uses TFTP to transfer a file that contains the executable image of a second-stage boot program.

User Datagram Protocol (UDP)
A kernel-level protocol that allows a process on one host to send a datagram to a process on another. UDP is a connectionless transport protocol. UDP messages include a protocol port number that allows the sender to distinguish multiple programs on the remote host.

wide area network (WAN)
A network that extends across a wide area, such as across a street or across an ocean.

workstation A system with its own processor, its own graphics terminal, and graphics software (shared or host-dependent). A workstation could be an OS server, a diskless client, or a client with a disk.

Index

Note: Boldfaced page numbers (e.g., **1-5**) indicate definitions of terms or other key information.

H

I

T

U

Related Documents

The following list of related manuals gives titles of Data General manuals followed by nine-digit numbers used for ordering. You can order any of these manuals via mail or telephone (see the TIPS Order Form in the back of this manual).

For a complete list of AViiON® and DG/UX™ manuals, see *Guide to AViiON® and DG/UX™ System Documentation* (060-701085). The on-line version of this manual found in **/usr/release/doc_guide** contains the most current list.

Data General Software Manuals

Using TCP/IP on the DG/UX™ System (093-701023).
Introduces Data General's TCP/IP family of protocols and describes how to use the package.

Managing TCP/IP on the DG/UX™ System (093-701051).
Explains how to prepare for the installation of Data General's TCP/IP package on AViiON computer systems. Tells how to tailor the software for your site, use **sysadm** or **xsysadm** to manage the package, and troubleshoot system problems.

System Manager's Reference for the DG/UX™ System (093-701050).
Alphabetical listing of manual pages for commands relating to system administration or operation.

Managing ONC™/NFS® and Its Facilities on the DG/UX™ System (093-701049).
Shows how to install, manage, and use the DG/UX ONC/NFS product. Contains information on the Network File System (NFS), the Network Information Services (NIS), Remote Procedure Calls (RPC), and External Data Representation (XDR).

Data General Hardware Manuals

Ethernet/IEEE 802.3 Local Area Network Installation Guide (014-000793).
Explains how to install both the coaxial cable plant of an Ethernet local area network (LAN) and the transceivers that connect the network to a node communication controller.

DG/Token Ring Local Area Network Installation Guide (014-001730).
Tells how to install DG/Token Ring. Describes the physical and functional aspects of the DG/Token Ring network, identifies the network components and lists important physical, functional and environmental specifications.

AViiON 300 and 400 Series Stations: Programming System Control and I/O Registers (093-001800).
Describes the workstation architecture and explains how to program the system

control logic, monochrome and color graphics controller subsystems, keyboard port, mouse port, serial and parallel ports, LAN interface, and SCSI port.

Configuring the VME Token Ring Controller (VTRC) for AViiON Systems (014-001730). Provides information about programming and installing the VME Token Ring Controller (VTRC).

Setting Up and Installing VMEbus Options in AViiON Systems (014-001867). Provides information about how to insert the VME Token Ring Controller (VTRC) board in 2-slot VME chassis and how to jumper the board.

Expanding the AViiON® 5000 Series System (014-001850). Explains how to open and close the computer, plan a configuration, and install add-on boards. Includes a description of the computer subassemblies.

V/Ethernet 3207 Hawk Local Area Network Controller for Ethernet User's Guide (014-001818). Contains information about programming and installing the V/Ethernet 3207 Hawk Local Area Network Controller (VLC).

Request for Comments

Technical reports, protocol proposals, and protocol standards appear in a series of documents called *Request for Comments*, or *RFCs*. You can get copies of *RFCs* from the Defense Data Network — Network Information Center (known as the NIC), SRI International, Menlo Park, CA, 94025. The following *RFCs* may be helpful to you as you administer a TCP/IP-based network.

- *RFC 768 (User Datagram Protocol)*
- *RFC 783 (Trivial File Transfer Protocol)*
- *RFC 791 (Internet Protocol)*
- *RFC 792 (Internet Control Message Protocol)*
- *RFC 793 (Transmission Control Protocol)*
- *RFC 821 (Simple Mail Transfer Protocol)*
- *RFC 822 (Standard for the Format of ARPA Internet Text Messages)*
- *RFC 826 (An Ethernet Address Resolution Protocol)*
- *RFC 854 (Telnet Protocol)*
- *RFC 877 (A Standard for the Transmission of IP Datagrams Over Public Data Networks)*
- *RFC 903 (A Reverse Address Resolution Protocol)*

- *RFC 950 (Internet Standard Subnetting Procedures)*
- *RFC 952 (DOD Internet Host Table Specification)*
- *RFC 959 (File Transfer Protocol)*
- *RFC 974 (Mail Routing and the Domain System)*
- *RFC 1034 (Domain Names — Concepts and Facilities)*
- *RFC 1035 (Domain Names -- Implementation and Specification)*
- *RFC 1042 (A Standard for the Transmission of IP Datagrams over IEEE 802 Networks)*
- *RFC 1047 (Duplicate Messages and SMTP)*
- *RFC 1058 (Routing Information Protocol)*
- *RFC 1068 (Background File Transfer Program)*
- *RFC 1101 (DNS Encoding of Network Names and Other Types)*
- *RFC 1122 (Requirements for Internet Hosts -- Communication Layers)*
- *RFC 1123 (Requirements for Internet Hosts -- Application and Support)*
- *RFC 1155 (Structure and Identification of Management Information for TCP/IP-based Internets)*
- *RFC 1156 (Management Information Base for Network Management of TCP/IP-based Internets)*
- *RFC 1157 (Simple Network Management Protocol)*
- *RFC 1158 (Management Information Base for Network Management of TCP/IP-based internets: MIB-II)*

End of Related Documents

TIPS ORDERING PROCEDURES

TO ORDER

1. An order can be placed with the TIPS group in two ways:
   a) MAIL ORDER – Use the order form on the opposite page and fill in all requested information. Be sure to include shipping charges and local sales tax. If applicable, write in your tax exempt number in the space provided on the order form.

      Send your order form with payment to:

      Data General Corporation
      ATTN: Educational Services/TIPS G155
      4400 Computer Drive
      Westboro, MA 01581–9973

   b) TELEPHONE – Call TIPS at (508) 870–1600 for all orders that will be charged by credit card or paid for by purchase orders over $50.00. Operators are available from 8:30 AM to 5:00 PM EST.

METHOD OF PAYMENT

2. As a customer, you have several payment options:
   a) Purchase Order – Minimum of $50. If ordering by mail, a hard copy of the purchase order must accompany order.
   b) Check or Money Order – Make payable to Data General Corporation.
   c) Credit Card – A minimum order of $20 is required for Mastercard or Visa orders.

SHIPPING

3. To determine the charge for UPS shipping and handling, check the total quantity of units in your order and refer to the following chart:

Total Quantity	Shipping & Handling Charge
1–4 Units	$5.00
5–10 Units	$8.00
11–40 Units	$10.00
41–200 Units	$30.00
Over 200 Units	$100.00

If overnight or second day shipment is desired, this information should be indicated on the order form. A separate charge will be determined at time of shipment and added to your bill.

VOLUME DISCOUNTS

4. The TIPS discount schedule is based upon the total value of the order.

Order Amount	Discount
$1–$149.99	0%
$150–$499.99	10%
Over $500	20%

TERMS AND CONDITIONS

5. Read the TIPS terms and conditions on the reverse side of the order form carefully. These must be adhered to at all times.

DELIVERY

6. Allow at least two weeks for delivery.

RETURNS

7. Items ordered through the TIPS catalog may not be returned for credit.
8. Order discrepancies must be reported within 15 days of shipment date. Contact your TIPS Administrator at (508) 870–1600 to notify the TIPS department of any problems.

INTERNATIONAL ORDERS

9. Customers outside of the United States must obtain documentation from their local Data General Subsidiary or Representative. Any TIPS orders received by Data General U.S. Headquarters will be forwarded to the appropriate DG Subsidiary or Representative for processing.

TIPS ORDER FORM

Mail To: Data General Corporation
Attn: Educational Services/TIPS G155
4400 Computer Drive
Westboro, MA 01581 - 9973

BILL TO:	SHIP TO: (No P.O. Boxes - Complete Only If Different Address)
COMPANY NAME ________	COMPANY NAME ________
ATTN: ________	ATTN: ________
ADDRESS ________	ADDRESS (NO PO BOXES) ________
CITY ________	CITY ________
STATE ________ ZIP ________	STATE ________ ZIP ________

Priority Code ________ (See label on back of catalog)

Authorized Signature of Buyer (Agrees to terms & conditions on reverse side)	Title	Date	Phone (Area Code)	Ext.

ORDER #	QTY	DESCRIPTION	UNIT PRICE	TOTAL PRICE

A SHIPPING & HANDLING

V UPS		ADD
1-4 Items		$ 5.00
5-10 Items	$ 8.00	
11-40 Items	$ 10.00	
41-200 Items	$ 30.00	
200+ Items	$100.00	

Check for faster delivery

Additional charge to be determined at time of shipment and added to your bill.

V UPS Blue Label (2 day shipping)

V Red Label (overnight shipping)

B VOLUME DISCOUNTS

Order Amount	Save
$0 – $149.99	0%
$150 – $499.99	10%
Over $500.00	20%

Tax Exempt # or Sales Tax (if applicable) ________

ORDER TOTAL	
Less Discount See B	–
SUB TOTAL	
Your local* sales tax	+
Shipping and handling – See A	+
TOTAL – See C	

C PAYMENT METHOD

V Purchase Order Attached ($50 minimum)
P.O. number is ________. (Include hardcopy P.O.)
V Check or Money Order Enclosed
V Visa V MasterCard ($20 minimum on credit cards)

Account Number ☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐☐ Expiration Date ☐☐☐☐

____
Authorized Signature
(Credit card orders without signature and expiration date cannot be processed.)

THANK YOU FOR YOUR ORDER

PRICES SUBJECT TO CHANGE WITHOUT PRIOR NOTICE.
PLEASE ALLOW 2 WEEKS FOR DELIVERY.
NO REFUNDS NO RETURNS.

* Data General is required by law to collect applicable sales or use tax on all purchases shipped to states where DG maintains a place of business, which covers all 50 states. Please include your local taxes when determining the total value of your order. If you are uncertain about the correct tax amount, please call 508–870–1600.

Form 702
Rev. 8/87

DATA GENERAL CORPORATION TECHNICAL INFORMATION AND PUBLICATIONS SERVICE TERMS AND CONDITIONS

Data General Corporation ("DGC") provides its Technical Information and Publications Service (TIPS) solely in accordance with the following terms and conditions and more specifically to the Customer signing the Educational Services TIPS Order Form. These terms and conditions apply to all orders, telephone, telex, or mail. By accepting these products the Customer accepts and agrees to be bound by these terms and conditions.

1. CUSTOMER CERTIFICATION
Customer hereby certifies that it is the owner or lessee of the DGC equipment and/or licensee/sub-licensee of the software which is the subject matter of the publication(s) ordered hereunder.

2. TAXES
Customer shall be responsible for all taxes, including taxes paid or payable by DGC for products or services supplied under this Agreement, exclusive of taxes based on DGC's net income, unless Customer provides written proof of exemption.

3. DATA AND PROPRIETARY RIGHTS
Portions of the publications and materials supplied under this Agreement are proprietary and will be so marked. Customer shall abide by such markings. DGC retains for itself exclusively all proprietary rights (including manufacturing rights) in and to all designs, engineering details and other data pertaining to the products described in such publication. Licensed software materials are provided pursuant to the terms and conditions of the Program License Agreement (PLA) between the Customer and DGC and such PLA is made a part of and incorporated into this Agreement by reference. A copyright notice on any data by itself does not constitute or evidence a publication or public disclosure.

4. LIMITED MEDIA WARRANTY
DGC warrants the CLI Macros media, provided by DGC to the Customer under this Agreement, against physical defects for a period of ninety (90) days from the date of shipment by DGC. DGC will replace defective media at no charge to you, provided it is returned postage prepaid to DGC within the ninety (90) day warranty period. This shall be your exclusive remedy and DGC's sole obligation and liability for defective media. This limited media warranty does not apply if the media has been damaged by accident, abuse or misuse.

5. DISCLAIMER OF WARRANTY
EXCEPT FOR THE LIMITED MEDIA WARRANTY NOTED ABOVE, DGC MAKES NO WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY AND FITNESS FOR PARTICULAR PURPOSE ON ANY OF THE PUBLICATIONS, CLI MACROS OR MATERIALS SUPPLIED HEREUNDER.

6. LIMITATION OF LIABILITY
A. CUSTOMER AGREES THAT DGC'S LIABILITY, IF ANY, FOR DAMAGES, INCLUDING BUT NOT LIMITED TO LIABILITY ARISING OUT OF CONTRACT, NEGLIGENCE, STRICT LIABILITY IN TORT OR WARRANTY SHALL NOT EXCEED THE CHARGES PAID BY CUSTOMER FOR THE PARTICULAR PUBLICATION OR CLI MACRO INVOLVED. THIS LIMITATION OF LIABILITY SHALL NOT APPLY TO CLAIMS FOR PERSONAL INJURY CAUSED SOLELY BY DGC'S NEGLIGENCE. OTHER THAN THE CHARGES REFERENCED HEREIN, IN NO EVENT SHALL DGC BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING BUT NOT LIMITED TO LOST PROFITS AND DAMAGES RESULTING FROM LOSS OF USE, OR LOST DATA, OR DELIVERY DELAYS, EVEN IF DGC HAS BEEN ADVISED, KNEW OR SHOULD HAVE KNOWN OF THE POSSIBILITY THEREOF; OR FOR ANY CLAIM BY ANY THIRD PARTY.

B. ANY ACTION AGAINST DGC MUST BE COMMENCED WITHIN ONE (1) YEAR AFTER THE CAUSE OF ACTION ACCRUES.

7. GENERAL
A valid contract binding upon DGC will come into being only at the time of DGC's acceptance of the referenced Educational Services Order Form. Such contract is governed by the laws of the Commonwealth of Massachusetts, excluding its conflict of law rules. Such contract is not assignable. These terms and conditions constitute the entire agreement between the parties with respect to the subject matter hereof and supersedes all prior oral or written communications, agreements and understandings. These terms and conditions shall prevail notwithstanding any different, conflicting or additional terms and conditions which may appear on any order submitted by Customer. DGC hereby rejects all such different, conflicting, or additional terms.

8. IMPORTANT NOTICE REGARDING AOS/VS INTERNALS SERIES (ORDER #1865 & #1875)
Customer understands that information and material presented in the AOS/VS Internals Series documents may be specific to a particular revision of the product. Consequently user programs or systems based on this information and material may be revision-locked and may not function properly with prior or future revisions of the product. Therefore, Data General makes no representations as to the utility of this information and material beyond the current revision level which is the subject of the manual. Any use thereof by you or your company is at your own risk. Data General disclaims any liability arising from any such use and I and my company (Customer) hold Data General completely harmless therefrom.

Cut here and insert in binder spine pocket

Data General

Data General Corporation, Westboro, Massachusetts 01580