HEWLETT PACKARD

BSD IPC Reference Manual

for NS-ARPA/1000 and ARPA/1000

Software Services and Technology Division
11000 Wolfe Road
Cupertino, CA 95014-9804

Manual Part No. 91790-90060
E0495

Printed in U.S.A. April 1995
Fourth Edition

Printing History

The Printing History below identifies the edition of this manual and any updates that are included. Periodically, update packages are distributed which contain replacement pages to be merged into the manual, including an updated copy of this printing history page. Also, the update may contain write-in instructions.

Each reprinting of this manual will incorporate all past updates; however, no new information will be added. Thus, the reprinted copy will be identical in content to prior printings of the same edition with its user-inserted update information. New editions of this manual will contain new information, as well as all updates.

To determine what manual edition and update is compatible with your current software revision code, refer to the Manual Numbering File. (The Manual Numbering File is included with your software. It consists of an "M" followed by a five digit product number.)

First Edition	Aug 1991	Rev. 5.24/5240
Second Edition	Dec 1992	Rev. 6.0/6000
Third Edition	Nov 1993	Rev. 6.1/6100
Fourth Edition	Apr 1995	Rev. 6.2/6200

Preface

Hewlett-Packard NS-ARPA/1000 and ARPA/1000 network software products allow HP computer systems to communicate with other systems providing ARPA services and TCP/IP protocols. The BSD IPC functionality of NS-ARPA/1000 and ARPA/1000 provides the capability for remote process communication.

Audience

The *BSD IPC Reference Manual for NS-ARPA/1000 and ARPA/1000* is the primary reference source for programmers who will be writing or maintaining programs on NS-ARPA/1000 and ARPA/1000 that use Berkeley Software Distribution Interprocess Communication (BSD IPC). The *BSD IPC Reference Manual* should also be read by Network Managers so that they will have a clear understanding of the full implications of various NS-ARPA/1000 and ARPA/1000 functions and features.

Assumptions

This manual is intended for experienced programmers, familiar with the RTE-A operating system. For those operations that deal with HP 9000 systems, a working knowledge of the HP-UX operating system is also recommended. Network Managers, who have responsibility for generating and initializing nodes and configuring networks, should consult the *ARPA/1000 Node Manager's Manual*, part number 98170-90001, or both the *NS-ARPA/1000 Generation and Initialization Manual*, part number 91790-90030, and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*, part number 91790-90031.

Organization

Section 1	**Introduction**—presents an overview of NS-ARPA/1000 and ARPA/1000, introducing their User Services, which includes BSD IPC.
Section 2	**BSD IPC Concepts**—defines the terms and concepts commonly used in BSD IPC.
Section 3	**Using BSD IPC**—describes the steps to set up a BSD IPC connection, transfer data, and shut down the connection.
Section 4	**BSD IPC Calls**—provides reference information on each BSD IPC call, arranged in alphabetical order.
Section 5	**BSD IPC Utilities**—provides reference information on each BSD IPC utility available on the HP 1000, arranged in alphabetical order.

Section 6	**Socket Descriptor Utilities**—provides reference information on the socket descriptor utilities which allows operation on the socket descriptor bitmasks, arranged in alphabetical order.
Section 7	**Advanced Topics**—covers socket options and nonblocking I/O on the HP 1000.
Appendix A	**Example Programs**—provides example programs in C, Pascal, and FORTRAN using BSD IPC.
Appendix B	**Database and Header Files**—describes and lists the database and header files used in BSD IPC for programming in C, Pascal, and FORTRAN.
Appendix C	**Error Messages**—provides a list of error messages for BSD IPC.
Appendix D	**Definition of Terms**—defines NS-ARPA/1000 and ARPA/1000 BSD IPC terms.

Guide to NS-ARPA/1000 Manuals

The following are brief descriptions of the manuals included with the NS-ARPA/1000 product.

91790-90060 BSD IPC Reference Manual for NS-ARPA/1000 and ARPA/1000

Describes Berkeley Software Distribution Interprocess Communication (BSD IPC) on the HP 1000. BSD IPC on the HP 1000 offers a programmatic interface on the HP 1000 for multi-vendor connectivity to systems that offers BSD IPC 4.3.

91790-90020 NS-ARPA/1000 User/Programmer Reference Manual

Describes the user-level services provided by NS-ARPA/1000. The NS services are network file transfer (NFT), network interprocess communication (NetIPC), and remote program management (RPM). The ARPA services are TELNET and FTP. Because these are interactive and programmatic services, this manual is intended for interactive users as well as programmers. It should also be read by Network Managers before designing an NS-ARPA/1000network so that they will have a clear understanding of the full implications of various NS-ARPA/1000 functions and features.

91790-90050 NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual

Describes the user-level services provided by the DS/1000-IV backward compatible services. These services are Remote File Access (RFA), DEXEC, REMAT, RMOTE, program-to-program communication (PTOP), utility subroutines, remote I/O mapping, remote system download to memory-based DS/1000-IV nodes only, and remote virtual control panel.

91790-90030 NS-ARPA/1000 Generation and Initialization Manual

Describes the tasks required to install, generate, and initialize NS-ARPA/1000. This manual is intended for the Network Manager. Before reading this manual, the Network Manager should read the *NS-ARPA/1000 User/Programmer Reference Manual* to gain an understanding of the NS-ARPA/1000 user-level services. The Network Manager should also be familiar with the RTE-A operating system and system generation procedure.

91790-90031 NS-ARPA/1000 Maintenance and Principles of Operation Manual

Describes the NS-ARPA/1000 network maintenance utilities, troubleshooting techniques, and the internal operation of NS-ARPA/1000. The Network Manager should use this manual in conjunction with the *NS-ARPA/1000 Generation and Initialization Manual*. This manual may also be used by advanced users to troubleshoot their applications.

91790-90040 NS-ARPA/1000 Quick Reference Guide

Lists and briefly describes the interactive and programmatic services described in the *NS-ARPA/1000 User/Programmer Reference Manual* and the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*. The purpose of this guide is to provide a quick reference for users who are already familiar with the concepts and syntax presented in those two manuals.

The *NS-ARPA/1000 Quick Reference Guide* also contains abbreviated syntax for certain programs and utilities described in the *NS-ARPA/1000 Generation and Initialization Manual* and the *NS-ARPA/1000 Maintenance and Principles of Operation Manual*. For your convenience, the *NS-ARPA/1000 Quick Reference Guide* also contains a master index of NS-ARPA/1000 manuals. This is a combined index from the NS-ARPA/1000 manuals to help you find information that may be in more than one manual.

91790-90045 NS-ARPA/1000 Error Message and Recovery Manual

Lists and explains, in tabular form, all of the error codes and messages that can be generated by NS-ARPA/1000. This manual should be consulted by programmers and users who will be writing or maintaining programs for NS-ARPA/1000 systems. Because it contains error messages generated by the NS-ARPA/1000 initialization program NSINIT and other network management programs, it should be consulted by Network Managers.

91790-90054 File Server Reference Guide for NS-ARPA/1000 and ARPA/1000

Describes information on using and administering the HP 1000 file server, including runstring parameters, files needed for configuration, troubleshooting guidelines, and error messages.

5958-8523 NS Message Formats Reference Manual

Describes data communication messages and headers passed between computer systems communicating over Distributed System (DS) and Network Services (NS) links.

5958-8563 NS Cross-System NFT Reference Manual

Provides cross-system NFT information. It is a generic manual that is a secondary reference source for programmers and operators who will be using NFT on NS-ARPA/1000, NS3000/V, NS3000/XL, NS/9000, NS for the DEC VAX* computer, and PC (PC NFT on HP OfficeShare Network). Information provided in this manual includes file name and login syntax at all of the systems on which NS NFT is implemented, a brief description of the file systems used by each of these computers, and end-to-end mapping information for each supported source/target configuration.

*DEC and VAX are U.S. registered trademarks of Digital Equipment Corporation.

Guide to ARPA/1000 Manuals

The following are brief descriptions of the manuals included with the ARPA/1000 product.

91790-90060 BSD IPC Reference Manual for NS-ARPA/1000 and ARPA/1000

Describes Berkeley Software Distribution Interprocess Communication (BSD IPC) on the HP 1000. BSD IPC on the HP 1000 offers a programmatic interface on the HP 1000 for multi-vendor connectivity to systems that offers BSD IPC 4.3.

98170-90001 ARPA/1000 Node Manager's Manual

Provides the information required to configure, generate, initialize, and maintain HP 1000 nodes in an ARPA network. The Network Manager should use this manual in conjunction with the *ARPA/1000 User's Manual*, part number 98170-90002, which describes the ARPA Services supported on the HP 1000.

98170-90002 ARPA/1000 User's Manual

Describes the user-level services provided by ARPA/1000. The ARPA services are TELNET and FTP. Because these are interactive and programmatic services, this manual is intended for interactive users as well as programmers. This manual also contains the error messages you might encounter when using ARPA/1000.

91790-90054 File Server Reference Guide for NS-ARPA/1000 and ARPA/1000

Describes information on using and administering the HP 1000 file server, including runstring parameters, files needed for configuration, troubleshooting guidelines, and error messages.

Conventions Used in this Manual

NOTATION	DESCRIPTION
`nonitalics`	Words in syntax statements that are not in italics must be entered exactly as shown. Punctuation characters other than brackets, braces, and ellipses must also be entered exactly as shown. For example: `EXIT;`
`italics`	Words in syntax statements that are in italics denote a parameter that must be replaced by a user-supplied variable. For example: `CLOSE` *`filename`*
`[ ]`	An element inside brackets in a syntax statement is optional. Several elements stacked inside brackets means the user may select any one or none of these elements. For example: $\begin{bmatrix}\texttt{A}\\\texttt{B}\end{bmatrix}$ User *may* select A or B or neither.
`{ }`	When several elements are stacked within braces in a syntax statement, the user must select one of those elements. For example: $\begin{Bmatrix}\texttt{A}\\\texttt{B}\\\texttt{C}\end{Bmatrix}$ User *must* select A or B or C.
`. . .`	A horizontal ellipsis in a syntax statement indicates that a previous element may be repeated. For example: `[,`*`itemname`*`]...;` In addition, vertical and horizontal ellipses may be used in examples to indicate that portions of the example have been omitted.

`,`	A shaded delimiter preceding a parameter in a syntax statement indicates that the delimiter *must* be supplied whenever (a) that parameter is included or (b) that parameter is omitted and any *other* parameter that follows is included. For example:
	
	`itema``[,`*`itemb`*`][,`*`itemc`*`]`
	
	means that the following are allowed:
	
	`itema`
	`itema,itemb`
	`itema,itemb,itemc`
	`itema,,itemc`

Δ	When necessary for clarity, the symbol Δ may be used in a syntax statement to indicate a required blank or an exact number of blanks. For example:
	`SET[(`*`modifier`*`)]Δ(`*`variable`*`);`
$\underline{\text{underlining}}$	When necessary for clarity in an example, user input may be underlined. For example:
	`NEW NAME?` $\underline{\texttt{ALPHA}}$
	Brackets, braces or ellipses appearing in syntax or format statements that must be entered as shown will be underlined. For example:
	`LET` *`var`*`[`$\underline{\texttt{[}}$*`subscript`*$\underline{\texttt{]}}$`] =` *`value`*
	Output and input/output parameters are underlined. A notation in the description of each parameter distinguishes input/output from output parameters. For example:
	`CREATE (`*`parm1`*`,`$\underline{\textit{parm2}}$`,`$\underline{\textit{flags}}$`,`$\underline{\textit{error}}$`)`
$\boxed{}$	The symbol $\boxed{}$ may be used to indicate a key on the terminal's keyboard. For example, $\boxed{\texttt{RETURN}}$ indicates the carriage return key.
$\boxed{\texttt{CONTROL}}$*`char`*	Control characters are indicated by $\boxed{\texttt{CONTROL}}$ followed by the character. For example, $\boxed{\texttt{CONTROL}}$`Y` means the user presses the control key and the character Y simultaneously.

Table of Contents

List of Illustrations

Tables

1

Introduction

This manual describes the HP 1000 implementation of 4.3 Berkeley Software Distribution Interprocess Communication (BSD IPC). BSD IPC is a set of programming development tools for interprocess communication, originally developed by the University of California at Berkeley (UCB).

BSD IPC allows you to create distributed applications that pass data between programs (on the same computer or on separate computers on the network) by using a set of library calls. These library calls, when used in the correct sequence, allow you to create communication endpoints called `sockets` and transfer data between them.

The best examples of how BSD IPC can be used are the ARPA/Berkeley Services commonly used on UNIX* systems. Using BSD IPC, you can write your own distributed application programs to do a variety of tasks. For example, you can write distributed application programs to

- access a remote database
- access multiple computers at one time
- spread subtasks across several hosts

Note BSD IPC programs must be compiled with CDS on.

Multi-Vendor Connectivities

BSD IPC is offered as part of NS-ARPA/1000 and ARPA/1000, which provide the following industry standard ARPA/Berkeley services: BSD IPC, TELNET, and FTP.

BSD IPC provides a programmatic interface for multi-vendor connectivities between the HP 1000 and other machines that support 4.3 BSD IPC. TELNET and FTP provide virtual terminal connection and file transfer capabilities, respectively. TELNET and FTP are covered in the *NS-ARPA/1000 User/Programmer Reference Manual* and the *ARPA/1000 User's Manual.*

*UNIX® is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

2

BSD IPC Concepts

This section provides definitions of terms and concepts used in HP 1000 BSD IPC.

Sockets

BSD IPC processes communicate with each other via *sockets*. Sockets are local data structures with associated resources used for interprocess communication. Sockets are analogous to Class Numbers on the RTE-A system, allowing different processes to exchange data. These processes may reside on the same machine or on two different machines.

Sockets are communication endpoints. A pair of connected sockets sets up communication between two processes.

The `socket()` call is used to create a socket. The call returns a *socket descriptor*, which is used to identify the socket and is used in all subsequent socket-related BSD IPC calls. Socket descriptors are usually small integers.

Each socket has an assigned *socket structure* which is used to store data on the socket; such as the socket type, socket options, socket error conditions, and so on.

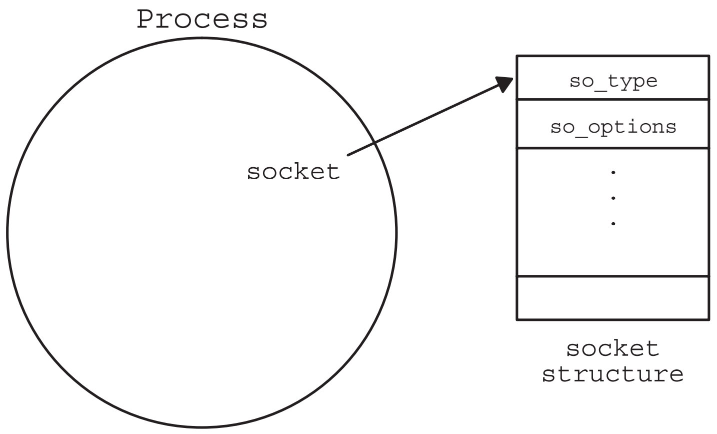


Figure 2-1. A Process with a Socket Created

Transport Layer Protocols

There are two Internet transport layer protocols that can be used with BSD IPC. They are TCP, which implements stream sockets, and UDP, which implements datagram sockets.

TCP

The Transmission Control Protocol (TCP) provides the underlying communication support for stream sockets. TCP is used to implement reliable, sequenced, flow-controlled, two-way communication based on byte streams.

With stream sockets, there are no end-of-message or end-of-data markers. This means that data received by an individual receiving call may not be equivalent to data sent by an individual sending call. In fact, the data received may contain part of the data or a multiple set of data sent by multiple sending calls. Although no attempt is made to preserve boundaries between data sent at different times, the data received will always be in the correct order (that is, in the order that it was sent).

UDP

The User Datagram Protocol (UDP) provides the underlying communication support for datagram sockets. UDP is an unreliable protocol. A process receiving messages on a datagram socket could find messages duplicated, out-of-sequence, or missing. Messages retain their record boundaries and are sent as individually addressed packets. UDP does not employ the concept of a connection between the communicating sockets.

Stream Sockets

Data transfer between an HP 1000 BSD IPC process and another BSD IPC process is in `stream mode`. In stream mode, data is transmitted in a stream of bytes; there are no end-of-message or end-of-data markers. This means that data received by an individual receiving call may not be equivalent to data sent by an individual sending call. In fact, the data received may contain part of the data or a multiple set of data sent by multiple sending calls. Although no attempt is made to preserve boundaries between data sent at different times, the data received will always be in the correct order (e.g., in the order that it was sent).

You may specify the maximum number of bytes that you are willing to send or receive through a parameter in the sending and receiving calls. For more information about manipulating the length of data sent and received, refer to the `send()` and `recv()` calls in Section 4, “BSD IPC Calls.”

Stream mode adheres to the Transport Layer’s Transmission Control Protocol (TCP).

Address Binding

Before a socket can be accessed across the network, it must be bound to an address. The `bind()` call is used to establish a socket's unique address. The address bound to a socket consists of the following fields:

- *Socket address family type*; e.g., the Internet (`AF_INET`) address family.

  The socket address family type defines the address format to be used for the socket. HP 1000 BSD IPC only supports the Internet (`AF_INET`) address family type which uses an address format of 16 bytes.

- *Port number* of the service used.

  Each service on your system has an assigned port number, stored in the */etc/services* file. The port number allows you to specify the service used by your process.

- *Internet (IP) address* of the host.

  The IP address distinguishes your node from other nodes on the network.

For more information about socket addresses and the fields in these addresses, refer to "Preparing Socket Addresses" in Section 3. The `/etc/services` file is covered in Appendix B.

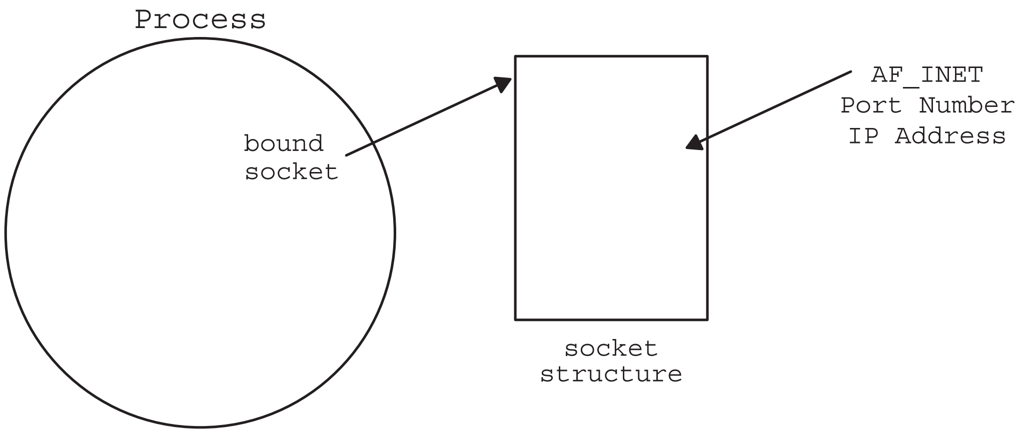


Figure 2-2. A Socket with a Bound Address

The Client-Server Model

Typical BSD IPC applications consist of two separate processes; one process (the *client*) requests a connection and the other process (the *server*) accepts it.

The server process creates a socket, binds an address to it, and sets up a mechanism, called a *listen queue*, for receiving connection requests.

The client process creates a socket and requests a connection to the server.

Once the server receives a connection request from the client, it creates another socket with all the same characteristics as the original server socket and establishes connection between the new server socket and the client socket. The original server socket continues to listen for more connection requests.

After a connection is established between the server and client, full-duplex (two-way) communication can occur between the two sockets, in a peer-to-peer manner. The distinction between client and server processes ceases to exist. Either process can send and receive data, and shut down the connection.

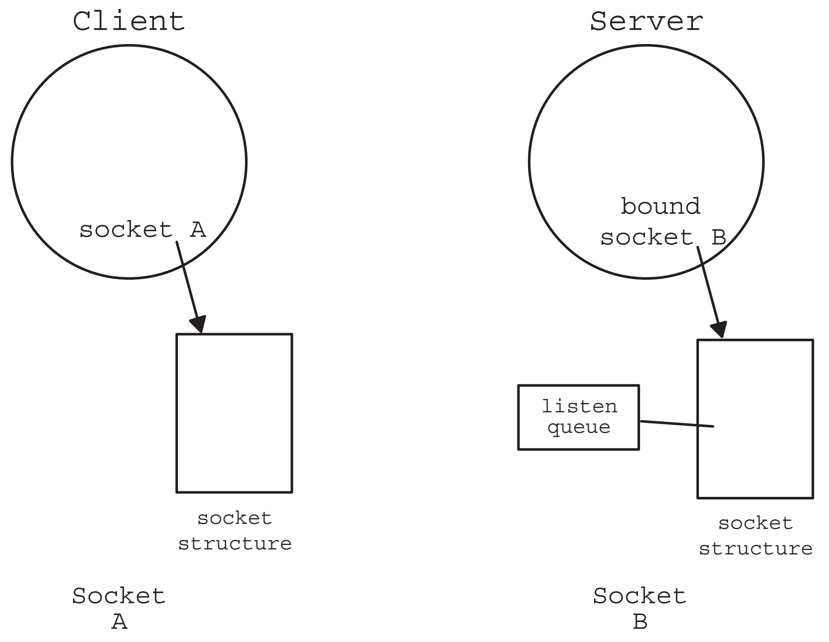
The next subsection shows the steps of the Client-Server Model in establishing connection.

Establishing a Connection

The following steps are used to establish connection between two BSD IPC processes.

1. The server creates a socket using the `socket()` call.
2. The server binds an address to the newly-created socket with the `bind()` call.
3. The server issues a `listen()` call to build a listen queue for the socket.
4. The server issues an `accept()` call and waits for a connection request from the client.
5. The client creates a socket using the `socket()` call.
6. The client requests connection to the specified server with a `connect()` call.
7. The server receives the connection request in the listen queue.
8. The server unblocks when it receives a connection request from the client process. The `accept()` call creates a new socket for the connection and returns the socket descriptor for the new socket.
9. The server establishes a connection to the client with the new socket that has all the characteristics of the original socket. Connection between the client and server has been established, and both processes can send and receive data as equal peers.
10. The original server socket continues to listen for more connection requests.

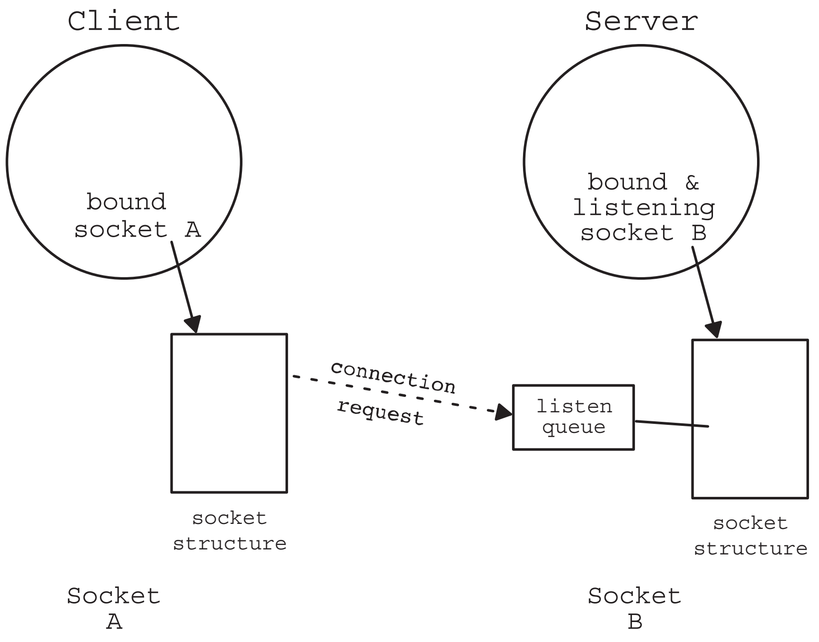
The following figures provide conceptual views of the client-server model at three different stages of establishing a connection. The steps that have been accomplished at each stage are listed below each figure.



5. Client creates a socket.

1. Server creates a socket.
2. Server binds an address to its socket.
3. Server sets up the listen queue.
4. Server waits for connection requests.

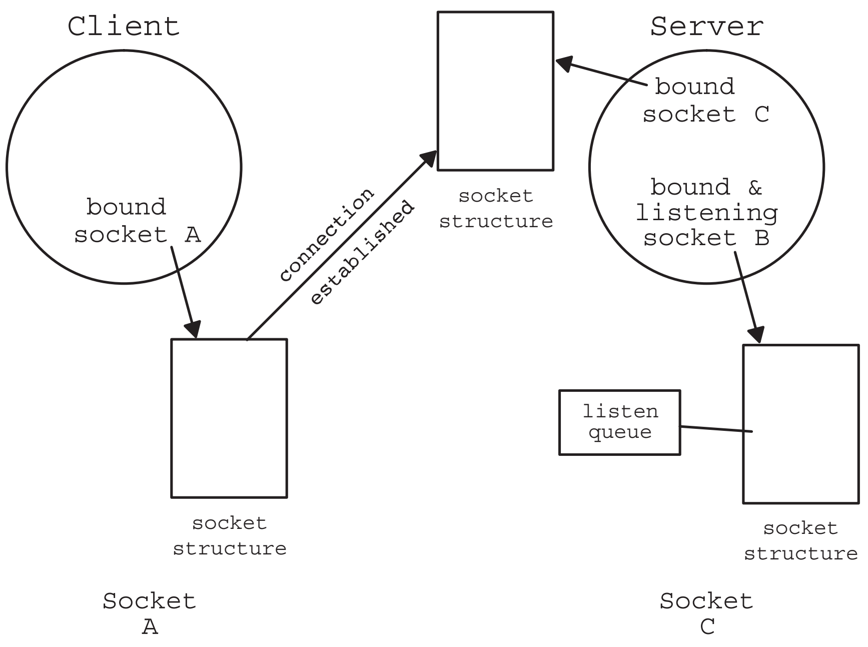
Figure 2-3. Client-Server in a Pre-Connection State



6. Client makes a connection request

7. Server receives the connection request in the listen queue.

Figure 2-4. Client-Server at Time of Connection Request



8. Server accepts the connection request.
9. Server establishes a connection to the client with a new server socket that has all the characteristics of the original socket.
10. Original server socket continues to listen for more connection requests.

Figure 2-5. Client-Server When Connection is Established

Sending and Receiving Data Over a Connection

Once connection is established between the client and server processes, both processes can send and receive data as equal peers. The `send()` call is used to send data, and the `recv()` call is used to receive data.

Depending on the need of your application, you can determine the pattern of data flow between the client and server processes.

For the following figure, note that Process A and Process B can be either the client or server process.

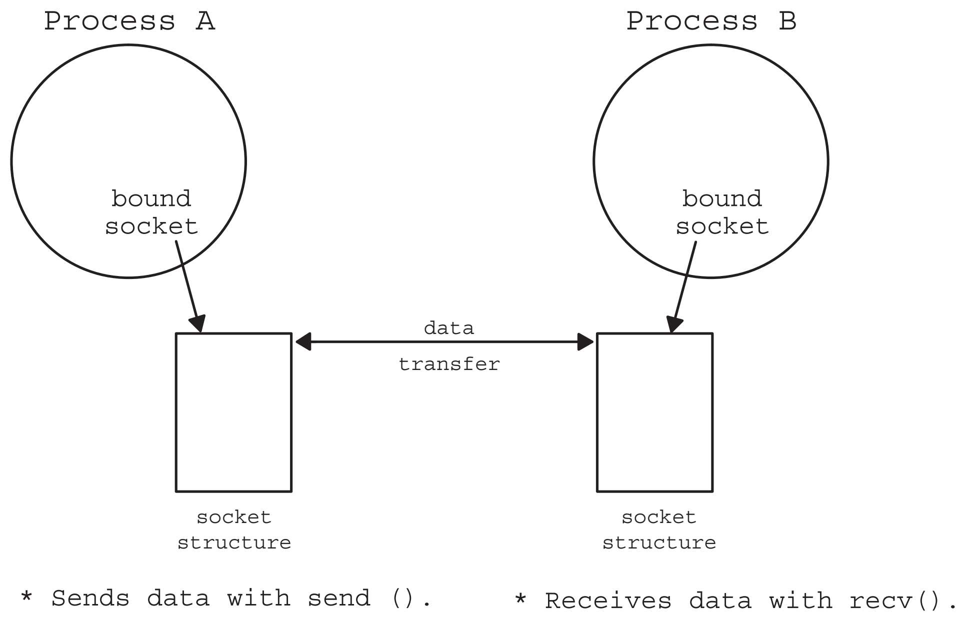


Figure 2-6. Data Transfer Between Two BSD IPC Processes

Terminating the Connection

Either the client or the server can terminate the connection.

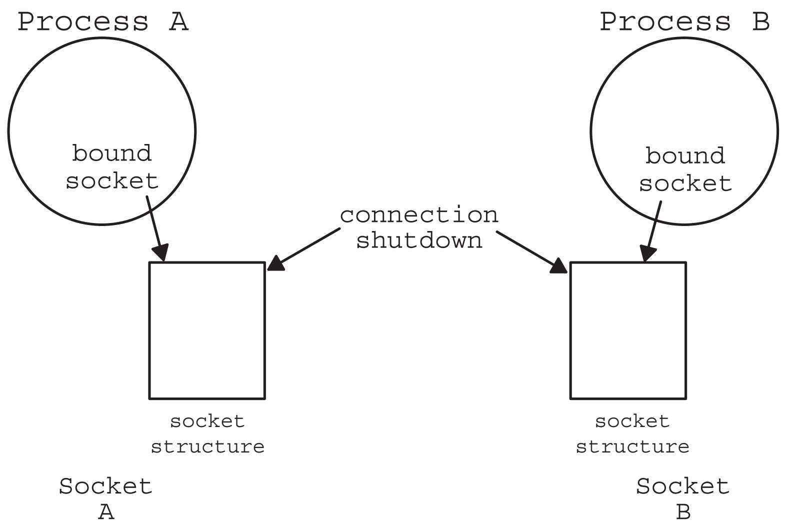
The `shutdown()` call closes a socket and shuts down the connection gracefully. This means that the application process need not worry about loss of data within the network. The transport protocol will deliver all the data to the receiver, even when the sender socket has already been shutdown.

The best way to end a session without losing data is summarized below:

1. With the `send()` call, the sending side sends an “I am finished” message. This message is defined by the application designer.
2. The receiving side reads this “I am finished” message with the `recv()` call. It finishes up any unfinished tasks (e.g., sends remaining data).
3. The receiving side sends a “I am finished” message and shuts down its socket with the `shutdown()` call.
4. The sending side receives the “I am finished” message.
5. The sending side frees its socket resources by issuing a `shutdown()` call.

When a process terminates, all its open sockets are closed.

The following figure illustrates the steps for terminating a connection. Note that Process A and Process B can be either the client or server.



```
1.  Sends “I am finished”.     ---->  2.  Receives “I am finished”.
                                          Finishes up.
                               <----  3.  Sends “I am finished”.
4.  Receives “I am finished”              Issues shutdown().
    notification.
5.  Issues shutdown().
```

Figure 2-7. Terminating a BSD IPC Connection

3

Using BSD IPC

This section describes the steps involved in using BSD IPC on the HP 1000. The following topics are covered in this section.

- Preparing Socket Addresses for the Server and Client
- Establishing Connection for the Server
- Establishing Connection for the Client
- Sending and Receiving Data
- Terminating the Connection
- Working with Pointers in Pascal and FORTRAN
- Data Types for Programming in C, Pascal, and FORTRAN
- BSD IPC Header Files
- Libraries for Linking BSD IPC Programs
- *Errno* and *errno* Returns
- Scheduling BSD IPC Programs

Table 3-1 lists the calls involved in creating, using, and terminating a BSD IPC connection with stream sockets, and Table 3-2 lists the calls used for datagram sockets.

Table 3-1. Building a BSD IPC Connection

Server Process	Client Process
1. `socket()` creates a socket	1. `socket()` creates a socket
2. `bind()` binds an address (See note below)	2. `bind()` binds an address (See note below)
3. `listen()` sets up a listen queue	
4. `accept()` waits & accepts a connection	
	5. `connect()` requests a connection
6. `send()` sends data `recv()` receives data `sendmsg()` sends vectored data `recvmsg()` receives vectored data	6. `send()` sends data `recv()` receives data `sendmsg()` sends vectored data `recvmsg()` receives vectored data
7. `shutdown()` shuts down a connection	7. `shutdown()` shuts down a connection
Note: `bind()` is necessary only if the application process wants to bind its socket to a specific port number. Most server processes would use `bind()` to bind a well-known port number to their socket so that client processes can access these server sockets.	

Usually, the server process is scheduled first. It creates a socket, binds an address to the socket, sets up a listen queue, and waits for requests from client processes.

The client process creates a socket and requests connection to the server. Once the server accepts the request, full-duplex connection is established between the two processes and the distinction between client and server can cease to exist. Both peer processes can send and receive data, as well as terminate the connection.

With datagram sockets there is no concept of a connection between the client and server processes. A client initiates a transaction by sending a datagram to the server. Both processes can send and receive datagrams to complete the transaction.

Table 3-2. Using Datagram Sockets

Server Process	Client Process
1. `socket()` creates a socket	1. `socket()` creates a socket
2. `bind()` binds an address (See note below)	2. `bind()` binds an address (See note below)
3. `recvfrom()` waits and receives datagram	
	4. `sendto()` sends datagram
5. `sendto()` sends datagram `recvfrom()` receives datagram	5. `sendto()` sends datagram `recvfrom()` receives datagram
6. `shutdown()` releases a socket	6. `shutdown()` releases a socket
Note: `bind()` is necessary only if the application process wants to bind its socket to a specific port number. Most server processes would use `bind()` to bind a well-known port number to their socket so that client processes can access these server sockets.	

Preparing Socket Addresses

Before you can begin to create a connection, you need to establish the correct socket addresses. For the server process, you need to set up a local socket address, which is then used in the `bind()` call to bind the address to the local socket. For the client process, you need to set up a socket address for the remote (server) process and use it in the `connect()` call, so as to specify which server process to connect to.

BSD IPC uses an address variable type `sockaddr_in` to store socket addresses. A socket address consists of three fields:

- socket address family type (`AF_INET`)
- port number of service used
- IP address of the node

Preparing a socket address consists of the following steps:

- declaring an address variable (`sockaddr_in`) suitable for storing a socket address
- getting the port number of the desired service

- assigning an IP address (server only)
- getting the remote host's IP address (client only)

These steps are covered here.

Declaring an Address Variable (Sockaddr_in)

Addressing information for both the client and server is contained in a variable of `sockaddr_in` type, which is an addressing variable used by the Internet family.

For C programming, `sockaddr_in` is a structure defined in the include file `<in.h>`. For Pascal and FORTRAN, `sockaddr_in` is a record type defined in `SOCKET.PASI` and `SOCKET.FTNI`, respectively.

`sockaddr_in` consists of the following fields which are used by BSD IPC:

```
sockaddr_in
    sin_family
    sin_port
    sin_addr
    sin_zero
```

`sin_family`	Specifies the address family. It must be set to `AF_INET`, for Internet address family. The address family specifies which type of addressing format is used in the socket addresses. `AF_INET` uses the `sockaddr_in` address format. Refer to "Address Family Type" in the Glossary for more information.
`sin_port`	Specifies the port number of the service to be used by the process. See the next subsection, "Getting the Port Number for the Desired Service," on how to obtain a port number for your BSD IPC process.
`sin_addr`	Specifies the Internet (IP) address. The IP address specifies the node on which the process resides. When you prepare a socket address for a server, you need to assign the IP address of the local host to this field. On the client process, you need to assign the IP address of the peer (server) process in this field to specify which server to connect to in the `connect()` call. Refer to "Getting the Remote Host's Internet Address" later in this section for more information. `Sin_addr` is actually a structure of type `in_addr`.
`sin_zero`	Padding field. Reserved for future use.

The `sockaddr_in` and `in_addr` structures are shown here for C, Pascal, and FORTRAN programming.

C Syntax

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

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;
};
```

Pascal Syntax

```
sockaddr_in   = RECORD
   CASE INTEGER OF
     1 : ( int1 : int );
     2 : ( sin_family    :  int;
           sin_port      :  int;
           sin_addr      :  in_addr;
           sin_zero      :  PACKED ARRAY [1..8] of char );
   END;

in_addr = PACKED RECORD
   CASE INTEGER OF
     1 : ( int1  : int );
     2 : (
              s_b1 : CHAR;
              s_b2 : CHAR;
              s_b3 : CHAR;
              s_b4 : CHAR;
     3 : ( s_w1 : int;
              s_w2 : int);
     4 : ( S_addr : long);
   END;
```

FORTRAN Syntax

```
INTEGER    SOCKADDR_IN(8)
INTEGER    SIN_FAMILY,SIN_PORT
INTEGER*4  SIN_ADDR
CHARACTER  SIN_ZERO*(8)
EQUIVALENCE  (SOCKADDR_IN(1),SIN_FAMILY)
EQUIVALENCE  (SOCKADDR_IN(2),SIN_PORT)
EQUIVALENCE  (SOCKADDR_IN(3),SIN_ADDR)
EQUIVALENCE  (SOCKADDR_IN(5),SIN_ZERO)

INTEGER    IN_ADDR(2)
INTEGER    S_W1,S_W2
INTEGER*4  S_ADDR
EQUIVALENCE  (IN_ADDR(1),S_W1,S_ADDR)
EQUIVALENCE  (IN_ADDR(2),S_W2)
```

The server uses the `sockaddr_in` address variable in its `bind()` call to bind an address to its socket and in the `accept()` call to determine the address of the client.

The client process uses the `sockaddr_in` address variable in the `connect()` call to specify which server to connect to. The client process need not explicitly bind an address to its local socket with the `bind()` call, because the `connect()` call binds a random address to the client socket. To find out the assigned address, use `getsockname()` after the `connect()` call.

For more information on the `bind()` and `connect()` calls, refer to Section 4, "BSD IPC Calls." For information on the `getsockname()` utility, refer to Section 5, "BSD IPC Utilities."

Getting the Port Number for the Desired Service

The port number specifies which service is used by the process. Both the client and server processes need to assign a port number as part of its socket address (see address variable `sockaddr_in` in the previous subsection).

The port number may be hard coded and directly assigned to the socket address in the program. (See Figure 3-1 below.) The other way is to get the port number of well-known services from the database file called `/etc/services` by using the `getservbyname()` function, then assign the returned port number to the socket address. (See Figure 3-2 below.)

```
#define SERVER_PORT 1000            /* server port address */
     :
struct sockaddr_in myaddr;          /* for local socket addr */
     :
myaddr.sin_port = SERVER_PORT;
```

Figure 3-1. Direct Assignment of Port Number (C Programming Example)

```
struct sockaddr_in myaddr;            /* for local socket addr */
    :
sp = getservbyname("example", tcp);
if (sp==NULL) {
        fprintf(stderr, "%S:host not found", argv[0]);
        exit(1);
}

myaddr.sin_port = sp->s_port;
```

Figure 3-2. Port Number Assignment with `getservbyname()` (C Programming Example)

The `getservbyname()` call and its parameters are summarized below.

```
service = getservbyname (name, proto)
```

name	Specify a valid service name.
proto	Specify the transport protocol to be used. Use "tcp" or 0 if TCP is the only protocol for the service.
service	Output variable which contains the port number for the specified service. The variable is defined as of the `servent` structure type, which has a field called `s_port` that contains the port number. Refer to the `getservbyname()` call in Section 5, "BSD IPC Utilities" for more information.

Note The `/etc/services` file contains a list of services and their corresponding port numbers that are available on the system. If the service is not already in `/etc/services`, you must add it. The `/etc/services` file should exist on both the client and server hosts. For more information on `/etc/services` refer to Appendix B, "Database and Header Files."

Assigning the IP Address for the Server

The third field in the socket address is the IP address of the node. The IP address is assigned to the `sin_addr` field in the socket address.

For the server process, you may assign a specific local IP address or assign `0` to this field. Assigning a specific local IP address means that this server process will listen on that IP connection. Assigning a `0` means that the server process will listen on all network connections coming through all its LAN cards.

Getting the Remote Host's Internet Address

The client process needs to set up a socket address for the peer (server) process. This socket address is used in the `connect()` call to specify which server to connect to.

This socket address, therefore, should contain addressing information on the server process. In order to get the server's IP address, use `gethostbyname()`, which returns the IP address of a given host name. You then assign this IP address to the IP address field in the socket address.

`Gethostbyname()` obtains the IP address of the given host from a database file called `/etc/hosts`. For more information on `/etc/hosts`, refer to Appendix B, "Database and Header Files."

`Gethostbyname()` and its parameters are summarized here.

```
    host = gethostbyname(name)
```

name	Pointer to a character string containing a valid host's name.
host	Output variable which contains the IP address of the specified host's name. The variable is of the `hostent` structure type, which has a field called `h_addr_list` that points to a list of IP addresses. Refer to `gethostbyname` in Section 5, "BSD IPC Utilities," for more information.

The `gethostbyname()` function is used in the client process to get the remote host's IP address, which is then assigned to the socket address structure used for the peer (server) process. The client process then uses this socket address in the `connect()` call to request connection to the specified server.

Setup for the Server

This subsection discusses in detail the actions taken by a server process prior to exchanging data. It discusses the calls the server executes and describes the parameters to these calls. Complete information on each call can be found in Section 4, "BSD IPC Calls," where the calls are discussed in alphabetical order. Example server programs in C, Pascal, and FORTRAN are provided in Appendix A, "Example Programs."

In the simplest case, there are five steps that the server process must complete before exchanging data with a client:

1. Set up the socket address for the server process. (Stream or datagram sockets.)
2. Create a socket with `socket()`. (Stream or datagram sockets.)
3. Bind the socket address set up in Step 1 to the new socket with `bind()`. (Stream or datagram sockets.)
4. Add a listen queue to the socket with `listen()`. (Stream sockets only.)
5. Wait for an incoming request with `accept()`. (Stream sockets only.)

These steps are described below.

Setting Up a Socket Address

You need to set up a socket address structure for the server process. Follow the steps here:

1. Declare an address variable of type `sockaddr_in`.
2. Get the port number for the service provided by the server process.
3. Assign an IP address to the server process.

These steps were described in the previous subsection, "Preparing Socket Addresses."

Creating a Socket

The server process must call `socket()` to create a BSD IPC socket. This must be done before any other BSD IPC call is executed.

The `socket()` call and its parameters are described here.

```
socket = socket(af, type, protocol)
```

af	Identifies the address family for the socket being created. It must be `AF_INET`.
type	Specifies the type of socket being created. It must be `SOCK_STREAM` or `SOCK_DGRAM`.
protocol	Specifies the underlying protocol to be used. `0` causes the default protocol to be used. The defaults are `IPROTO_TCP` for `SOCK_STREAM` type and `IPROTO_UDP` for `SOCK_DGRAM` type.
socket	If the call completes successfully, `socket` contains the socket descriptor for the newly-created socket. If the call encounters an error, `-1` is returned in *socket*, and *errno* contains the error code.

The socket descriptor returned by `socket()` references the newly-created socket. This descriptor is used by subsequent BSD IPC calls to establish a connection.

Refer to the `socket()` call in Section 4, "BSD IPC Calls," for more information.

Binding an Address to a Socket

After the server process has created a socket, the server must call `bind()` to associate a socket address to the socket. Until an address is bound to the server socket, other processes have no way to reference it.

The `bind()` call and its parameters are described here.

```
bind (socket, addr, addrlen)
```

socket	Specifies the socket descriptor of the local socket.

addr	Specifies the socket address to be bound to the socket.
addrlen	Specifies the length of the socket address. It is the size of variable `sockaddr_in`.

Refer to the `bind()` call in Section 4, “BSD IPC Calls,” for more information about binding a socket address.

Setting Up a Listen Queue (Stream Sockets Only)

After the server process has an address bound to it, it must call `listen()` to set up a queue that accepts incoming connection requests. The server cannot respond to a connection request until it has executed `listen()`.

The `listen()` call and its parameters are described here.

```
    listen (socket, backlog)
```

socket	Specifies the socket descriptor of local socket.
backlog	Specifies the maximum number of connection requests allowed in the queue at any time. Further incoming connection requests are rejected. Valid range: 1 to 5.

Refer to the `listen()` call in Section 4, “BSD IPC Calls,” for more information.

Accepting a Connection (Stream Sockets Only)

The server can accept any connection requests that enters its queue after it executes `listen()`. The server issues the `accept()` call and waits for a connection request. `Accept()` blocks until there is a connection request from a client process in the queue. `Accept()` then creates a new socket for the connection and returns the socket descriptor for the new socket. The new socket

- is created with the same properties as the old socket
- has the same local port number as the old socket
- is connected to the client process’ socket.

The `accept()` call and its parameters are described here.

```
    newsocket = accept (socket, addr, addrlen)
```

socket	Specifies the socket descriptor used in a previous `listen()` call.
addr	Specifies the socket address structure to contain the socket address of the client process.
addrlen	Specifies the length of *addr*.

newsocket	New socket created by `accept()`. If the call encounters an error, `-1` is returned in *newsocket*, and *errno* contains the error code.

There is no way for the server process to indicate which requests it can accept. It must accept all requests or none. Your server process can keep track of which process a connection is from by examining the address returned by `accept()`. Once you have this address, you can use `gethostbyaddr()` to get the host name. You can close down the connection if you do not want the server process to communicate with that particular host or port.

There is an option for the server process to return immediately after it issues the `accept()` call if there are no connection requests pending. Refer to “Nonblocking I/O” in Section 7, “Advanced Topics.”

Setup for the Client

This section discusses in detail the actions taken by a client process prior to exchanging data with a server process. It discusses the calls the client executes and describes the parameters to these calls. Complete information about each call can be found in Section 4, “BSD IPC Calls,” where the calls are arranged in alphabetical order.

The three steps that the client process must take are

1. Set up the socket address structure for the peer (server) process.
2. Create a socket with the `socket()` call.
3. Bind socket to an address. (Datagram sockets only.)
4. Make a connection request with the `connect()` call, using the address structure set up in Step 1 to specify which server process to connect to. (Stream sockets only.)

These steps are described below.

Setting Up the Socket Address

You need to set up a socket address structure to specify the server process that you want to connect to. To do so, follow the steps here:

1. Declare an address variable of type `sockaddr_in`.
2. Specify the port number of the service with `getservbyname()`.
3. Get the server process’ host IP address with `gethostbyname()`.

These steps are described in detail in the previous subsection, “Preparing Socket Addresses.”

Creating a Socket

Similar to the server process, the client process must also call `socket()` to create a BSD IPC socket. The socket must be created before the client can request a connection to the server process.

For a client process, the `socket()` call and its parameters are identical to those used by the server process when it creates a socket. Refer to “Creating a Socket” for the server process earlier in this section.

The socket descriptor for the newly-created socket should be used in the `connect()` call to establish connection to the server and in subsequent data transmission.

Binding an Address to a Socket (Datagram Sockets Only)

After the server process has created a socket, the server must call `bind()` to associate a socket address to the socket. Until an address is bound to the server socket, other processes have no way to reference it.

The `bind()` call and its parameters are described here.

```
    bind (socket, addr, addrlen)
```

socket	Specifies the socket descriptor of the local socket.
addr	Specifies the socket address to be bound to the socket.
addrlen	Specifies the length of the socket address. It is the size of variable `sockaddr_in`.

Refer to the `bind()` call in Section 4, “BSD IPC Calls,” for more information about binding a socket address.

Requesting a Connection (Stream Sockets Only)

The client process requests a connection to the server with the `connect()` call. The server must be prepared to accept the connection—in other words, the server must have

- created a socket
- bound an address to the socket
- set up a listen queue
- issued an `accept()` call, waiting for a connection request.

The `connect()` call and its parameters are described below.

```
    connect (socket, addr, addrlen)
```

socket	Specifies the socket descriptor of local socket requesting a connection.

addr	Specifies the socket address of the server socket to which the client wants to connect.
addrlen	Size of address structure *addr*.

`Connect()` initiates a connection and blocks if the connection is not ready, unless you are using nonblocking I/O. (For information on nonblocking I/O, refer to Section 7, "Advanced Topics.") When the connection is ready, the client process completes its `connect()` call.

As soon as the `connect()` call returns, the client process can send data. Local client IP and port number are bound when `connect()` is executed if you have not already bound them explicitly. These address values are chosen by the local host. To get the assigned address, use `getsockname()`.

Sending and Receiving Data – Stream Sockets

After the `connect()` and `accept()` calls are successfully executed, the connection is established and data can be sent and received between the two socket endpoints.

Sending Data – Stream Sockets

The `send()` call is used to send data. Both the client and server socket can send data. The `send()` call and its parameters are described here.

```
    count = send (socket, buffer, len, flags)
```

count	Number of bytes sent.
socket	Socket descriptor of socket on which data will be sent.
buffer	Buffer of data to be sent.
len	Size of *buffer*, in bytes.
flags	Optional flags. Currently, there are no flags supported on the HP 1000.

`Send()` blocks if there is no space available to hold the data to be sent, unless you are using nonblocking I/O. (For information on nonblocking I/O, refer to Section 7, "Advanced Topics.")

Refer to the `send()` call in Section 4, "BSD IPC Calls," for more information about sending data.

Receiving Data – Stream Sockets

The `recv()` call is used to receive data. Both client and server socket can receive data. The `recv()` call and its parameters are described here.

```
    count = recv (socket, buffer, len, flags)
```

count	Number of bytes received.
socket	Socket descriptor of socket from which data will be received.
buffer	Buffer that is to receive data.
len	Size of data buffer, in bytes.
flags	Optional flags.
	`MSG_PEEK` copies data into buffer, but does not discard data afterwards.

`Recv()` blocks until there is at least one byte of data to be received, unless you are using nonblocking I/O. (For information on nonblocking I/O, refer to Section 7, "Advanced Topics.") The host does not wait for *len* bytes of data to be available; if less than *len* bytes are available, that number of bytes is received.

No more than *len* bytes of data are received. If there are more than *len* bytes of data on the socket, the remaining bytes are received on the next `recv()` call.

Use the `MSG_PEEK` option to preview incoming data. If this option is set on a `recv()` call, any data returned remains in the socket buffer as though it had not been read yet. The next `recv()` call returns the *same data*.

Refer to the `recv()` call in Section 4, "BSD IPC Calls," for more information on receiving data.

Sending and Receiving Vectored Data – Stream Sockets

BSD IPC provides the `sendmsg()` call to send vectored data and the `recvmsg()` call to receive vectored data. For more information on these two calls and on vectored data, refer to `recvmsg()` and `sendmsg()` in Section 4, "BSD IPC Calls."

Sending and Receiving Data – Datagram Sockets

After the `bind()` call is successfully executed, data can be sent and received between the two socket endpoints.

Sending Data – Datagram Sockets

The `sendto()` call is used to send datagrams. Both the client and server socket can send data. The `sendto()` call and its parameters are described here.

```
    count = sendto (socket, buffer, len, flags, addr, addrlen)
```

count	Number of bytes sent.
socket	Socket descriptor of socket on which data will be sent.
buffer	Buffer of data to be sent.
len	Size of buffer, in bytes.
flags	Optional flags. Currently, there are no flags supported on the HP 1000.
addr	Specifies the destination socket address for the data.
addrlen	Size of address structure *addr*.

Refer to the `sendto()` call in Section 4, "BSD IPC Calls," for more information about sending data.

Receiving Data – Datagram Sockets

The `recvfrom()` call is used to receive datagrams. Both client and server sockets can receive data. The `recvfrom()` call and its parameters are described here.

```
    count = recvfrom (socket, buffer, len, flags, addr, addrlen)
```

count	Number of bytes received.
socket	Socket descriptor of socket from which data will be received.
buffer	Buffer that is to receive data.
len	Size of data buffer, in bytes.
flags	Optional flag. `MSG_PEEK` copies data into buffer, but does not discard data afterwards.
addr	Returns address of socket that sent datagram.
addrlen	Size of address structure *addr*.

`Recvfrom()` blocks I/O until there is a datagram to be received, unless you are using nonblocking I/O. (For information on nonblocking I/O, refer to Section 7, "Advanced Topics".)

No more than *`len`* bytes of data are received. If the datagram is longer than *`len`* bytes, the remaining bytes are discarded.

Use the `MSG_PEEK` option to preview incoming data. If this option is set on a `recvfrom()` call, any data returned remains in the socket buffer as though it had not been read yet. The next `recvfrom()` call returns the same data.

Refer to the `recvfrom()` call in Section 4, "BSD IPC Calls," for more information on receiving data.

Closing the Connection

To terminate the connection between stream sockets, either the client or server process can use the `shutdown()` call. The `shutdown()` call closes a socket and shuts down the connection. For datagram sockets, the `shutdown()` call is used only to close the socket, not shut down the connection.

The `shutdown()` call and its parameters are described here.

```
    shutdown (socket, how)
```

socket	The socket descriptor of the socket to be shut down.
how	The type of shutdown to take place. `0` disallows any more receives on the socket. `1` disallows any more sends from the socket. `2` disallows any more sends and receives.

Working with Pointers in Pascal and FORTRAN

BSD IPC uses C programming language semantics. Many of the parameters in the BSD IPC calls use pointers to access data. Consequently, we provide two routines for Pascal and FORTRAN users to handle pointers: `ByteAdrOf()` and `AddressOf()`.

ByteAdrOf() Function

The `ByteAdrOf()` function can be used to obtain the byte address of data objects that are accessed by character pointers in C (`char *`*`variable`*).

The `ByteAdrOf()` routine has the following format:

```
ByteAdrOf(firstobjword, offset)
```

firstobjword	Name of the first (16-bit) word of the data object.
offset	Offset from the beginning of the data object. May be positive or negative. (The first byte of a data object resides at offset zero.)

AddressOf() Function

The RTE-A routine `AddressOf()` can be used to obtain the addresses of data objects that are accessed by pointers other than character pointers in C. The `AddressOf()` routine has the following format:

```
AddressOf(firstobjword)
```

firstobjword	Name of the first (16-bit) word of the data object.

Note The `ByteAdrOf()` and `AddressOf()` functions provide Pascal and FORTRAN users the capability to pass an address where a pointer would be used in the C-language.

Example:

C format:

```
getsockopt (socket, level, optname, optval, optlen)

int  socket, level, optname;
char *optval;
int  *optlen;
```

Pascal and FORTRAN usage:

```
    getsockopt(socket, level, optname, ByteAdrOf(optval,offset)
                  AddressOf(optlen))
```

Data Types in C, Pascal, FORTRAN

The syntax shown for BSD IPC calls uses C programming language syntax. The following table provides the equivalent data types used for Pascal and FORTRAN. The data structures and records are shown in full (with their fields) in the respective C, Pascal, and FORTRAN header files provided in Appendix B, “Database and Header Files.”

Table 3-3. Data Types in C, Pascal, and FORTRAN

C	Pascal	FORTRAN
int	−32768..32767	INTEGER
char	CHAR	CHARACTER
long	INTEGER	INTEGER*4
short	−32768..32767	INTEGER*2
u_long	INTEGER	INTEGER*4
u_short	0...65535	INTEGER*2
fd_set	fd_setType = RECORD	INTEGER*4 FD_SETTYPE
struct sockaddr_in	sockaddr_in = RECORD	INTEGER SOCKADDR_IN(8)
struct in_addr	in_addr = PACKED RECORD	INTEGER IN_ADDR(2)
struct msghdr	msghdr = RECORD	INTEGER MSGHDR(6)
struct iovec	iovec = RECORD	INTEGER MSG_IOVEC
struct hostent	hostent = RECORD	INTEGER HOSTENT(5)
struct netent	netent = RECORD	INTEGER NETENT(5)
struct protoent	protoent = RECORD	INTEGER PROTOENT(3)
struct servent	servent = RECORD	INTEGER SERVENT(4)
struct timeval	timeval = RECORD	INTEGER*4 TIMEVAL(2)
struct fd_set	fd_setType = RECORD	INTEGER*4 FD_SETTYPE

BSD IPC Header Files

Header files provide standard definition of macros and variables used in programming. BSD IPC provides header files for C, Pascal, and FORTRAN. Information on header files is provided in Appendix B, "Database and Header Files."

Table 3-4 lists the header files that need to be included in the BSD IPC programs.

Table 3-4. BSD IPC Header Files

C	Pascal	FORTRAN

`<types.h>`
`<socket.h>`
`<in.h>`
`<netdb.h>`
`<fcntl.h>`
	`SOCKET.PASI`	`SOCKET.FTNI`

Libraries for Linking BSD IPC Programs

HP 1000 NetIPC programs should be compiled and linked as CDS programs. Refer to the *RTE-A Programmer's Reference Manual*, part number 92077-90007, and *RTE-A Link User's Manual*, part number 92077-90035, for more information on CDS programs. After the program is linked, an RTE executable file (type 6) is ready to be scheduled.

There are two sets of libraries to consider when linking BSD IPC programs:

- *HPC.LIB*. The HP 92078A Virtual Code+ (VC+) product provides a C support library called `HPC.LIB`. During installation, it is copied to `/LIBRARIES/HPC.LIB`. Those users who do not have a standard C library may use this library to link BSD IPC programs. Refer to the subsection, "ERRNO and ERRNO Returns," later in this section for information on linking BSD IPC programs to resolve the *errno* variable.

- *BIGNS_CDS.LIB*. The `/LIBRARIES/BIGNS_CDS.LIB` library is created by merging other libraries. This is usually done through the command file, `INSTALL_NS1000.CMD` or `INSTALL_ARPA.CMD`.

  For NS-ARPA/1000 only, note the following:

  Because of an entry point conflict between the BSD IPC library and the RTE-MPE backward compatible services library, only one of these libraries is merged to create `BIGNS_CDS.LIB`. If your `BIGNS_CDS.LIB` is created with no RTE-MPE backward compatible services access, then the BSD IPC library (`BSD_CDS.LIB`) is part of `BIGNS_CDS.LIB`. Refer to the "Building NS-ARPA Libraries" subsection in the *NS-ARPA/1000 Generation and Initialization Manual*, part number 91790-90030, for further explanation.

  If your `BIGNS_CDS.LIB` is created for RTE-MPE backward compatible services, you will have to search `BSD_CDS.LIB` first to resolve external references for BSD IPC. Then search `BIGNS_CDS.LIB` (which is the default and done automatically).

Errno and Errno Returns

`Errno` is a standard error variable used in UNIX programming. For portability, the C library (`HPC.LIB`) also returns error values in a global variable called `errno`. In order to resolve references to this `errno` variable, programmers on the HP 1000 must perform the following steps:

For C Programmers

1. Put the following include file statement in the `include` section of the program.

   ```
   #include <errno.h>
   ```

2. Search the C library (either your own C library or `HPC.LIB`) during the linking phase.

For Pascal Programmers

1. Include the following compiler directive in the `IMPORT` section of the program or module.

   ```
   SEARCH 'errnodec.rel' $ERRNODEC
   ```

   The relocatable `errnodec.rel` is provided with the product in the `/NS1000/REL` directory. It is recommended that the network manager copy this file into a directory that is in the standard search path of the `$SEARCH` directive. (Look at `$SEARCH` in the Pascal manual for more information.)

2. In the linking phase to resolve the external reference for `errno`, do one of the following:

   a. Relocate `errnodec.rel` along with the main program.

   b. Search the C library (either your own C library or `HPC.LIB`).

For FORTRAN Programmers

1. In the linking phase to resolve the external reference for `errno`, do one of the following:

   a. Relocate `errnodec.rel` along with the main program. `Errnodec.rel` is a relocatable file provided with the product in the `/NS1000/REL` directory.

   b. Search the C library (either your own C library or `HPC.LIB`).

Scheduling BSD IPC Programs

BSD IPC itself does not include a call to schedule a peer process. The method used to schedule a remote BSD IPC process depends on the types of systems involved. These methods are discussed here.

Remote HP 1000 BSD IPC Process

There are at least six different ways (listed below) to schedule a remote HP 1000 BSD IPC process from another HP 1000 node. A remote HP 1000 BSD IPC process must be ready to execute by being an RTE type 6 file.

- *Remote Process Management (RPM)*. NS-ARPA/1000 only. The `RPMCreate` call programmatically schedules a program. RPM is an NS Common Service and is described in the *NS-ARPA/1000 User/Programmer Reference Manual*, part number 91790-90020.

- *Program-to-Program communication (PTOP)*. NS-ARPA/1000 only. The `POPEN` call programmatically schedules a program. PTOP is a DS/1000-IV Compatible Service and is described in the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*, part number 91790-90050.

- *Distributed EXEC (DEXEC)*. NS-ARPA/1000 only. One of the DEXEC scheduling calls, such as DEXEC 9, 10, 12, 23, 24, programmatically schedules a program. DEXEC is a DS/1000-IV Compatible Service and is described in the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*.

- *REMAT*. NS-ARPA/1000 only. The REMAT `QU` (queue schedule a program without wait) command interactively schedules a program. REMAT is a DS/1000-IV Compatible Service and is described in the *NS-ARPA/1000 DS/1000-IV Compatible Services Reference Manual*.

- *TELNET virtual terminal*. Logon remotely with TELNET and use the RTE `XQ` (schedule a program without wait) command to interactively schedule a program. TELNET is an ARPA Service and is described in the *NS-ARPA/1000 User/Programmer Reference Manual* and in the *ARPA/1000 User's Manual.*

- *RTE WELCOME file*. The WELCOME file can have RTE run commands to schedule programs after system boot up. Refer to the *RTE-A System Generation and Installation Manual*, part number 92077-90034, for information about booting up the RTE system and about the WELCOME file.

You cannot use any of the above NS-ARPA and DS/1000-IV compatible services to schedule a remote HP 1000 process from a non-HP 1000 node. These services are not provided with cross-system support.

Remote HP 1000 processes that are to work with non-HP 1000 processes can be manually started or can be programs that are started at system start up.

- To manually start up a BSD IPC program, simply logon to the HP 1000 system and run the BSD IPC program with the RTE `XQ` (run program without wait) command.

- To have the BSD IPC program execute at system start up, put the RTE `XQ` command in the WELCOME file.

The `XQ` command is explained in the *RTE-A User's Manual*, part number 92077-90002.

Remote HP-UX and UNIX BSD IPC Process

Remote HP 9000 processes can be manually started or can be scheduled by daemons that are started at system start up. In HP-UX a daemon is a process that runs continuously and usually performs system administrative tasks. Although a daemon runs continuously, it performs actions upon an event happening or at designated times.

To manually start up a BSD IPC program, simply logon to the HP 9000 system and run the BSD IPC program. HP recommends that you write a BSD IPC daemon to schedule your BSD IPC programs. You can start the daemon at system start up by placing it in your `/etc/netlinkrc` file. Refer to the HP 9000 LAN software installation documentation for more information about this file and system start up.

Remote PC BSD IPC Process

To manually start up a PC BSD IPC program, enter the BSD IPC program name at the MS-DOS* prompt.

To execute from within MS-Windows, copy the BSD IPC program files to your Windows directory and double click with the mouse on the executable file.

*MS-DOS is a U.S. registered trademark of Microsoft Corporation.

4

BSD IPC Calls

This section provides reference information on the BSD IPC calls. The calls are arranged in alphabetical order for easy referencing. Table 4-1 lists the calls covered in this section. Figure 4-1 on the following page shows a summary flowchart of BSD IPC calls used by a server process and client process.

Table 4-1. Building a BSD IPC Connection

BSD IPC Call	Description
`accept()`	Accepts a connection on a socket.
`bind()`	Binds an address to a socket.
`connect()`	Initiates a connection on a socket.
`fcntl()`	Provides socket option control.
`getsockopt()`	Returns current socket options.
`listen()`	Listens for a connection on a socket.
`recv()`	Receives a message from a stream socket.
`recvfrom()`	Receives a message from a datagram socket.
`recvmsg()`	Receives vectored data from a stream socket.
`select()`	Provides synchronous socket I/O multiplexing.
`send()`	Sends message to a stream socket.
`sendmsg()`	Sends vectored data to a stream socket.
`sendto()`	Sends message to a datagram socket.
`setsockopt()`	Sets socket options.
`shutdown()`	Shuts down a socket.
`socket()`	Creates a socket, an endpoint for communication.

Caution The command syntax shown in this section uses C programming syntax. Table 3-3 in the previous section provides equivalent data types for Pascal and FORTRAN programming.

BSD IPC Calls—Summary Flowchart

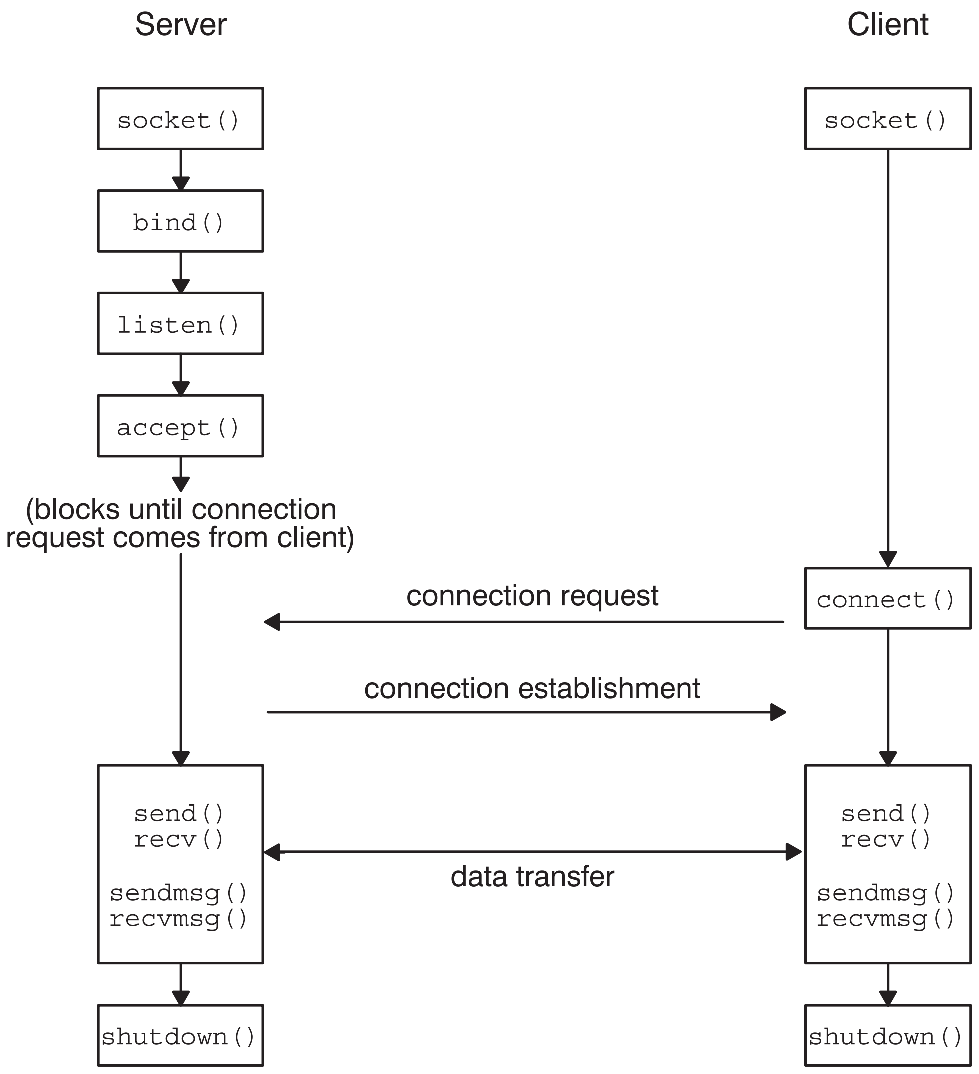


Figure 4-1. BSD IPC Calls—Summary Flowchart

Accepts a connection on a socket and creates a new socket. The call returns the new socket descriptor. The `accept()` call is used by the server process to wait for and accept a connection request from the client process. This call is used for stream sockets only.

Syntax

```
newsocket = accept(socket, addr, addrlen)

int                    newsocket, socket, *addrlen;
struct sockaddr_in     *addr;
```

Parameters

newsocket	New socket descriptor created by `accept()`. If the call is successful, the value returned is an integer equal to or greater than `0`. If the call fails, it returns `-1`.
socket	Original socket descriptor, created by a previous `socket()` call.
addr	Pointer to address structure. The address structure should be of `sockaddr_in` type, which is described in “Preparing Socket Addresses” in Section 3. On return, this structure contains the socket address of the client process that is connected to the server's new socket.
addrlen	Pointer to an integer variable that contains the length, in bytes, of the address structure specified by *addr* (e.g., length of structure `sockaddr_in`, which is 16 bytes). On return, *addrlen* contains the length, in bytes, of the actual client socket address returned in *addr*.

Discussion

`Accept()` is used by the server to accept connection requests from client processes. A client process requests a connection to the server process with the `connect()` call. These connection requests are entered into the server's listen queue. The server process can accept any connection requests that enter its queue after it executes `listen()`. `Accept()` extracts the first connection on the queue of pending connections, creates a new socket for the connection, and returns the socket descriptor for the new socket. The new socket

- is created with the same properties as the old socket
- has the same bound port number as the old socket
- is connected to the client process' socket

`Accept()` blocks until there is a connection request from a client process in the queue, unless you are using nonblocking I/O.

If you are using nonblocking I/O and no pending connections are present on the queue, `accept()` returns `-1` in *`newsocket`* and *`errno`* contains an `EAGAIN` error. The original socket, *`socket`*, remains open. It is possible to determine if a listening socket has pending connection requests ready for an `accept()` call by using `select()` for reading. Refer to `select()` for more information.

Nonblocking I/O is specified with the `O_NONBLOCK` flag setting in the `fcntl()` call. Refer to `fcntl()` for more information. Nonblocking I/O is covered in Section 7 "Advanced Topics."

There is no way for the server process to indicate which requests it can accept. It must accept all requests or none. Your server process can keep track of which process a connection request is from by examining the address returned by `accept()`. Once you have this address (e.g., the *`addr`* parameter), you can use `gethostbyaddr()` to get the host name. You can close down the connection if you do not want the server process to communicate with that particular client host or port.

If the size of the client's socket address is greater than the value of *`addrlen`*, then only the first *`addrlen`* bytes of the client's address will be returned in the socket address structure pointed by *`addr`*.

HP 1000 Specific Information

BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

- The `ByteAdrOf()` function can be used to obtain the byte address of data objects that are accessed by character pointers in C (that is, `char *`*`variable`*).
- The RTE-A routine `AddressOf()` is used to obtain the address of data objects that are accessed by pointers other than character pointers (e.g., `int *`*`variable`*).

Use the `AddressOf()` function to obtain the *`addr`* and *`addrlen`* pointers.

Refer to "Working With Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Error Returns

If `accept()` returns `-1` in *`newsocket`*, the call has failed. The global variable *`errno`* provides information on the cause of the call's failure. The following table lists possible error returns from the `accept()` call.

Error Mnemonic	Meaning
`[EINTERR]`	This error requires HP notification.
`[EAGAIN]`	Nonblocking I/O is enabled and no connection is present to be accepted.
`[EINVAL]`	One of the following occurred: The value of *`addrlen`* is less than zero. The socket was not created by the `socket()` call; thus, it is not of the Berkeley Socket type. The socket has already been shutdown. The socket is not ready to accept connections yet. A `listen()` call must be done before an `accept()` call.
`[EMFILE]`	The maximum number of socket descriptors for this process are already currently open.
`[ENOTSOCK]`	The socket descriptor, *`socket`*, is not a valid socket descriptor.
`[EOPNOTSUPP]`	The socket descriptor, *`socket`*, is not a valid listen socket.
`[ENOBUFS]`	No buffer space is available. The `accept()` call cannot be completed. The queued socket's `connect()` request is aborted.
`[EHOSTDOWN]`	The network software on the local host is not running.

bind()

Binds the specified socket address to the socket.

Syntax

```
result = bind(socket, addr, addrlen)

int                 result, socket, addrlen;
struct sockaddr_in  *addr;
```

Parameters

result	`0` if `bind()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
addr	Pointer to socket address structure that is to be bound to *socket*. The socket address should use a structure of `sockaddr_in` type, which is described in “Preparing Socket Addresses” in Section 3.
addrlen	Length (in bytes) of the socket address structure (e.g., size of structure `sockaddr_in`, which is 16 bytes). *Addr* should be at least 16 bytes.

Discussion

`Bind()` assigns the address specified in *addr* to the specified socket, *socket*.

Set up the address structure with a local address before you make the `bind()` call. The address structure should be of type `sockaddr_in`. For more information on setting up a socket address, refer to “Preparing Socket Addresses” in Section 3.

The socket address contains three addressing fields:

- The address family type.
- The server’s port number.
- The server’s IP address. (The IP address field is currently ignored by the `bind` call.)

The address family type should be `AF_INET`.

If the port number field (`sin_port`) in the socket address structure is defaulted to `0`, the system will assign a unique port number for the socket. Port numbers from 1 to 1023 are reserved for superusers.

Caution For stream sockets, the client process need not explicitly bind an address to its local socket with the `bind()` call, because the `connect()` call binds a random address to the client socket. To find out the assigned address, use `getsock-name()` after the `connect()` call.

HP 1000 Specific Information

BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

- The `ByteAdrOf()` function can be used to obtain the byte address of data objects which are accessed by character pointers in C (that is, `char *`*`variable`*).
- The RTE-A routine `AddressOf()` is used to obtain the address of data objects which are accessed by pointers other than character pointers.

Use the `AddressOf()` function to obtain the *`addr`* pointer.

Refer to “Working With Pointers in Pascal and FORTRAN” in Section 3 for more information on pointers.

Error Returns

If `bind()` returns `-1`, the call has failed. The global variable *`errno`* provides information on the cause of the call’s failure. The following table lists possible error returns from the `bind()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EINVAL]	The socket is already bound to an address, the socket has been shut down, *addrlen* is an invalid value, or socket is not a Berkeley socket.
[ENOTSOCK]	The socket descriptor, *socket*, is not a valid socket descriptor.
[EOPNOTSUPP]	The socket is not of a type that supports address binding.
[EAFNOSUPPORT]	Requested address does not match the address family of this socket.
[EADDRINUSE]	The specified address is already in use. Use the SO_REUSEADDR option in `setsockopt()` to force binding of the socket to the specified address.
[EADDRNOTAVAIL]	The specified address is invalid or not available from the local machine, or it is a reserved port available only to superusers.
[EHOSTDOWN]	The network software on the system is not running.

connect()

Initiates a connection request on a socket. This call is issued by the client process to connect to a specified server process. The `connect()` call is used for stream sockets only.

Syntax

```
result = connect(socket, addr, addrlen)

int                  result, socket, addrlen;
struct sockaddr_in   *addr;
```

Parameters

result	`0` if `connect()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket requesting a connection.
addr	Pointer to a structure containing the socket address of the remote (server) socket to which the connection is to be established. The socket address should be of `sockaddr_in` type, which is described in "Preparing Socket Addresses" in Section 3.
addrlen	Length, in bytes, of the address structure specified by *addr* (e.g., length of structure `sockaddr_in`, which is 16 bytes). *Addrlen* should be at least 16 bytes.

Discussion

The `connect()` call normally blocks until the connection completes, unless nonblocking mode is enabled.

In nonblocking mode if the connection cannot be completed immediately, `connect()` returns an `EINPROGRESS` error. In this case, the `select()` call can be used on the socket to determine if the connection has completed by selecting it for write. Refer to the `select()` call for more information.

Nonblocking mode is enabled by setting the `O_NONBLOCK` flag in the `fcntl()` call. For more information on nonblocking I/O, refer to Section 7, "Advanced Topics."

If the client socket does not already have a socket address bound to it, the `connect()` call will bind a random address to the client socket. To find the assigned address, use `getsockname()` after the `connect()` call.

Each connection between a client and server process is uniquely identified by the following 5-tuples: <domain, client TCP port address, client IP address, server TCP port address, server IP address>. If there is already a connection on the client system that is a duplicate of the one just requested, then the `connect()` call will return an `EADDRINUSE` error. This is because BSD IPC does not allow duplicate connections between two pairs of client-server sockets with the same socket addresses. There will be no way to differentiate these two associations.

HP 1000 Specific Information

Although the user has already been allocated a socket, via the `socket()` call, an `EMFILE` error indicating that there are no sockets in the system for the process could be returned, because an extra socket is needed for internal use during connection establishment.

BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

- The `ByteAdrOf()` function can be used to obtain the byte address of data objects which are accessed by character pointers in C (for example, `char *`*`variable`*).
- The RTE-A routine `AddressOf()` is used to obtain the address of data objects which are accessed by pointers other than character pointers.

Use the `AddressOf()` function to obtain the *`addr`* pointer.

Refer to "Working With Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Error Returns

If `connect()` returns `-1`, the call has failed. The global variable *`errno`* provides information on the cause of the call's failure. The following table lists possible error returns from the `connect()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EINVAL]	The socket has already been shut down, has a `listen()` active on it, or is not a BSD IPC socket. Or, *`addrlen`* is a bad value.
[EMFILE]	The system limit of socket descriptors has been exceeded. This could happen since sockets need to be created for internal use.
[ENOTSOCK]	*`socket`* is not a valid socket descriptor.
[EOPNOTSUPP]	A `connect()` attempt was made on a socket type which does not support this call.
[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
[EADDRINUSE]	The address is already in use.
[EADDRNOTAVAIL]	The specified address is not a valid server port address.
[ENOBUFS]	No buffer space is available. The `connect()` call has failed.

Error Mnemonic	Meaning
`[EISCONN]`	The socket is already connected.
`[ETIMEDOUT]`	Connection establishment timed out without establishing a connection. *`Backlog`* on the server may be full.
`[ECONNREFUSED]`	The attempt to connect was rejected by the server.
`[EINPROGRESS]`	Nonblocking I/O is enabled using `O_NONBLOCK` and the connection has been initiated. This is not a failure. Use `select()` to find out when the connection is complete.

fcntl()

Provides socket I/O control. Can be used to set nonblocking I/O mode for the specified socket.

Syntax

```
result = fcntl(socket, cmd, status)

int   socket, cmd;
long  result, status;
```

Parameters

result	`0` if `fcntl()` is successful. `-1` if a failure occurs.	
socket	Socket descriptor of a local socket.	
cmd	Command to get or set socket status. The possible values for *cmd* are:	
	`F_GETFL`	Get the socket status. The status value is returned in *status*.
	`F_SETFL`	Set the status to value as specified in *status*. The only status setting currently supported is `O_NONBLOCK`.
status	Specify the socket status. It is a 32-bit data type, each bit of which represents a characteristic of the socket. Setting and unsetting the bit in this parameter sets or unsets the socket characteristic, respectively. For `F_SETFL`, `fcntl()` sets the current status to the value specified in *status*. The currently supported value for *status* is:	
	`O_NONBLOCK`	This option designates the socket as nonblocking. A request on a nonblocking socket that cannot complete immediately returns to the caller and sets *errno* to `EAGAIN`. This option affects the following calls: `accept()`, `connect()`, `recv()`, `recvfrom()`, and `send()`. In a nonblocking `connect()` call, the *errno* value returned is set to `EINPROGRESS` instead of `EAGAIN`.

Discussion

`Fcntl()` is a function that returns the value in a 32-bit integer.

`Fcntl()` with `F_GETFL` is used to get the current status of the socket. `Fcntl()` with `F_SETFL` is used to set the status of the socket. The only *`status`* option currently supported on the HP 1000 BSD IPC is nonblocking mode. Nonblocking mode specifies that the socket does not block (wait) for an I/O event but returns immediately, with an error condition if the event cannot complete. For more information on nonblocking mode, refer to Section 7, "Advanced Topics".

Sockets are created in blocking mode by default.

Error Returns

If the `fcntl()` call is successful, it returns the settings of the requested flags in a 32-bit integer, *`status`*. If it failed, it returns a `-1`. The global variable *`errno`* provides information on the cause of the call's failure. The following table lists possible error returns from the `fcntl()` call.

Error Mnemonic	Meaning
`[EINVAL]`	The specified socket is not a BSD IPC socket.
`[ENOTSOCK]`	*`socket`* is not a valid socket descriptor.
`[EOPNOTSUPP]`	An invalid *`cmd`* or *`status`* value was used.
`[EHOSTDOWN]`	The network software is not running on the local host.

Returns status of current socket options.

Syntax

```
result = getsockopt(socket, level, optname, optval, optlen)

int   result, socket, level, optname;
char *optval;
int  *optlen;
```

Parameters

result	`0` if `getsockopt()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
level	The protocol level at which the socket option resides. To specify "socket" level, *`level`* should be `SOL_SOCKET`. To specify "TCP" level, *`level`* should be `IPPROTO_TCP`.
optname	Socket option name. The following options are supported for "socket" level (`SOL_SOCKET`) options:

`SO_KEEPALIVE`	(Toggle option) Sets a timer for 90 minutes for connected sockets. After 90 minutes expire, and if the connection has been idle during this period, `SO_KEEPALIVE` forces a transmission every 60 seconds for up to 7 minutes, after which the idle connection is shut down. In summary, `SO_KEEPALIVE` allows an idle period of 97 minutes before connection shutdown. If this option is toggled off, an indefinite idle time is allowed. This option is set by default.
`SO_REUSEADDR`	(Toggle option) Allows local socket address reuse. This allows multiple sockets to be bound to the same local port address. This option modifies the rules used by `bind()` to validate local addresses. `SO_REUSEADDR` allows more than one socket to be bound to the same port number at the same time; however, it only allows one single socket to be actively listening for connection requests on the port number. The host will still check at connection time to be sure any other socket with the same local

		address and local port does not have the same remote address and remote port. `Connect()` fails if the uniqueness requirement is violated.
	`SO_RCVBUF`	Returns the buffer size of a socket's receive socket buffer. The default buffer size is 4096 bytes. A stream socket's buffer size can be increased or decreased only prior to establishing a connection.
	`SO_SNDBUF`	Returns the buffer size of a socket's send socket buffer. The default buffer size is 4096 bytes. A stream socket's buffer size can be increased or decreased only prior to establishing a connection.

The following options are supported for "TCP" level (`IPPROTO_TCP`) options:

`TCP_MAXSEG`	Returns the maximum segment size in use for the socket. The value for this option can only be examined, it cannot be set. If the socket is not yet connected, TCP returns a default size of 512 bytes.
`TCP_NODELAY`	(Toggle option) Instructs TCP to send data as soon as it receives it and to bypass the buffering algorithm that tries to avoid numerous small packets from being sent to the network.

`optval`	Byte pointer to a variable into which an option value is returned. *`optval`* returns a NULL if the option information is not of interest and not to be passed to the calling process. Although *`optval`* is typed as (`char *`), the value that it points to is not terminated by `\0`.
`optlen`	Pointer to a variable containing the maximum number of bytes to be returned by *`optval`*. On return, it contains the actual number of bytes returned by *`optval`*.

Discussion

To get the status of current socket options, use `getsockopt()`. To set socket options, use `setsockopt()`.

There are two kinds of socket options: boolean (toggle) options and non-boolean options.

Boolean options are options that can be set on or off. To determine whether or not a boolean option is set, use `getsockopt()` with the desired option specified in *`optname`*. If the option is set, `getsockopt()` returns without an error. If the boolean option is not set, `getsockopt()` returns `-1` and *`errno`* is set to `ENOPROTOOPT`. The currently supported boolean options are: `SO_KEEPALIVE`, `SO_REUSEADDR`, and `TCP_NODELAY`.

Non-boolean options contain specific values. Non-boolean options use *optval* and *optlen* to pass information. On return, the character array pointed to by *optval* contains the value of the specified option or NULL if the option information is not of interest. *Optlen* points to an integer that contains the actual number of bytes of option information in the character array pointed to by *optval*.

HP 1000 Specific Information

The SO_LINGER option (available on HP-UX) is not provided on the HP 1000.

BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

- The ByteAdrOf() function can be used to obtain the byte address of data objects that are accessed by character pointers in C (that is, char **variable*).
- The RTE-A routine AddressOf() is used to obtain the address of data objects that are accessed by pointers other than character pointers.

Use the ByteAdrOf() function to obtain the *optval* pointer. Use the AddressOf() function to obtain the *optlen* pointer.

Refer to "Working With Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Error Returns

If the getsockopt() call fails, it returns a -1. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the getsockopt() call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EFAULT]	The *optval* or optlen parameter is not valid.
[EINVAL]	The specified option is unknown at the socket level or the socket has been shut down, or the specified socket is not a BSD IPC socket.
[ENOTSOCK]	*socket* is not a valid socket.
[ENOPROTOOPT]	The requested socket option is currently not set.
[EHOSTDOWN]	The network software on the local host is not running.

listen()

Sets up a listen queue for the specified socket on the server process and listens for connection requests. The `listen()` call is used for stream sockets only.

Syntax

```
result = listen(socket, backlog)

int result, socket, backlog;
```

Parameters

result	`0` if `listen()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
backlog	Defines the maximum allowable length of the queue for pending connections. The current valid range for *backlog* is 1 to 5. If any other value is specified, the system automatically assigns the closest value within range. If the queue is greater than the backlog, additional incoming requests will be rejected.

Discussion

To accept connections, a socket is first created with the `socket()` call, a queue for incoming connections is set up with the `listen()` call, and then connection is accepted with the `accept()` call.

If a socket has not been bound to a local port before the `listen()` call is invoked, the system automatically binds a local port for the socket to listen on. You can find out the assigned port number with the `getsockname()` utility. In this case, you must provide a way to notify this port number to client processes so they can specify it in their connection request calls to the server.

Error Returns

If the `listen()` call is successful, it returns a `0`. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `listen()` call.

Error Mnemonic	Meaning
`[EINTERR]`	This error requires HP notification.
`[EINVAL]`	The socket has been shut down or is already connected, or it is not a BSD IPC socket.
`[EMFILE]`	Currently, there are no resources available.
`[ENOTSOCK]`	*`socket`* is not a valid socket descriptor.
`[EOPNOTSUPP]`	The socket is not of a type that supports the `listen()` call. It is not a BSD IPC socket.
`[EADDRINUSE]`	There is already another socket that is listening on the same port address.
`[EHOSTDOWN]`	The network software is not running on the local host.

recv()

Receives data from a socket. The `recv()` call may be used by both the server and client processes. This call is used for stream sockets only.

Syntax

```
count= recv(socket, buffer, len, flags)

int  count, socket, len;
char *buffer;
long flags;
```

Parameters

count	Returns the number of bytes actually received. Returns `0` if the remote process has gracefully shut down and there is no more data in the receive buffer. Returns `-1` if the call encounters an error.
socket	Socket descriptor of the local socket receiving data.
buffer	Byte pointer to the data buffer.
len	Maximum number of bytes that will be returned into the buffer referenced by *buffer*. No more than *len* bytes of data are received. If there are more than *len* bytes of data on the socket, the remaining bytes are received on the next `recv()`.
flags	Optional flag options. The currently supported values for *flags* are:

`0`	No option.
`MSG_PEEK`	Option to preview incoming data. If this option is set on the `recv()` call, any data returned remains in the socket buffer as though it had not been read yet. The next `recv()` call returns the *same data*.

Discussion

The `recv()` calls may only be used after connection has been established between two processes.

There is no concept of message boundaries for HP 1000 BSD IPC sockets. Data is returned to the user on the `recv()` call as soon as it becomes available. If no data is available to be received, `recv()` waits for data to arrive, unless nonblocking I/O is enabled. `Recv()` does not wait for *len* bytes to be available; if less than *len* bytes are available, that number of bytes is received.

If the connection has been gracefully released by the remote side, and all the data has been received by the user, then `recv()` will return `0`.

If nonblocking I/O is enabled, the `recv()` request will complete in one of three ways:

- If there is enough data available to satisfy the entire request, `recv()` will complete successfully, having read *`len`* bytes of data in the buffer.
- If there is not enough data available to satisfy the entire request, `recv()` will complete successfully, having read as much data as possible, and returns the number of bytes it was able to read.
- If there is no data available, `recv()` will return `-1` with *`errno`* set to `EAGAIN`.

Nonblocking I/O is enabled by setting flag option `O_NONBLOCK` using `fcntl()`.

By selecting the socket for read indication, the `select()` call may be used to determine when a socket has data available to be read by a `recv()` call. See the `select()` call for more information.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in EMA (Extended Memory Area) cannot be accessed directly.

HP 1000 Specific Information

- `Recv()` requires the pointer to the data structure *`buffer`* to be a byte pointer. Pascal and FORTRAN users will have to use the routine `ByteAdrOf()` in order to get the byte address of the start of the data. For more information, refer to “Working With Pointers in Pascal and FORTRAN” in Section 3.
- `Recv()` on the HP 1000 supports the `MSG_PEEK` option. `Recv()` on the HP-UX supports the `MSG_PEEK` and `MSG_OOB` options.
- The HP 1000 currently does not support signals; hence, if a connection is terminated, the `recv()` call returns `-1`.

Error Returns

If the `recv()` call is successful, it returns the number of bytes received. If it failed, it returns a `-1`. The global variable *`errno`* provides information on the cause of the call’s failure. The following table lists possible error returns from the `recv()` call.

Error Mnemonic	Meaning
`[EINTERR]`	This error requires HP notification.
`[EINVAL]`	Invalid *`len`* value `<0`. Or, the socket is not a BSD IPC socket.
`[ENOTSOCK]`	The *`socket`* parameter is not a valid socket descriptor.
`[ECONNRESET]`	Connection aborted by the remote process.
`[ENOTCONN]`	*`socket`* has not yet been connected.
`[ESHUTDOWN]`	The socket has already been shutdown for receiving data.
`[EREMOTERELEASE]`	The remote side has done a send shutdown; hence, there will be no more data to receive.
`[EHOSTDOWN]`	The network software is not running on the local host.
`[EINPROGRESS]`	Connection has not yet been established.

recvfrom()

Receives datagrams from a socket. The `recvfrom()` call may be used by both the server and client processes.

Syntax

```
count= recvfrom(socket, buffer, len, flags, addr, addrlen)

int                 count, socket, len, *addrlen;
char                *buffer
long                flags;
struct sockaddr_in  *addr;
```

Parameters

count	Returns the number of bytes actually received. Returns `-1` if the call encounters an error.
socket	Socket descriptor of the local socket receiving data.
buffer	Byte pointer to the data buffer.
len	Maximum number of bytes that will be returned into the buffer referenced by buffer. No more than *len* bytes of data are received. If the next datagram is larger than *len* bytes, the remaining bytes are discarded.
flags	Optional flag options. The currently supported values for *flags* are:

`0`	No option.
`MSG_PEEK`	Option to preview incoming data. If this option is set on the `recvfrom()` call, any data returned remains in the socket buffer as though it had not been read yet. The next `recvfrom()` call returns the same data.

addr	Pointer to a structure containing the socket address of the remote socket which sent the data. The socket address will be of `sockaddr_in` type, which is described in “Preparing Socket Addresses” in Section 3.
addrlen	Length, in bytes, of the returned address structure.

Discussion

The `recvfrom()` call may only be used after the socket has been bound to an address by `bind()`.

If no data is available to be received, `recvfrom()` waits for data to arrive, unless nonblocking I/O is enabled.

If nonblocking I/O is enabled, the `recvfrom()` request will complete in one of two ways:

1. If there is a datagram available to satisfy the request, `recvfrom()` will complete successfully.
2. If there is no data available, `recvfrom()` will return `-1` with *errno* set to EAGAIN.

Nonblocking I/O is enabled by setting flag option O_NONBLOCK using `fcntl()`.

By selecting the socket for read indication, the `select()` call may be used to determine when a socket has data available to be read by a `recvfrom()` call. See the `select()` call for more information.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in EMA (Extended Memory Area) cannot be accessed directly.

HP 1000 Specific Information

- `Recvfrom()` requires the pointer to the data structure buffer to be a byte pointer. Pascal and FORTRAN users will have to use the routine `ByteAdrOf()` in order to get the byte address of the start of the data. For more information, refer to "Working with Pointers in Pascal and FORTRAN" in Section 3.
- `Recvfrom()` on the HP 1000 supports the `MSG_PEEK` option.

Error Returns

If the `recvfrom()` call is successful, it returns the number of bytes received. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `recvfrom()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EINVAL]	Invalid *len* value <0. Or, the socket is not a BSD IPC datagram-type socket.
[ENOTSOCK]	The *socket* parameter is not a valid socket descriptor.
[EHOSTDOWN]	The network software is not running on the local host.

Receives vectored data on a socket. This call may be used by both the server and client processes. The `recvmsg()` call is used for stream sockets only.

Syntax

```
count= recvmsg(socket, msg, flags)

int            count, socket;
struct msghdr *msg;
long           flags;
```

Parameters

count	Returns the number of bytes received. Returns `0` if the remote process has gracefully shut down and there is no more data in the receive buffer. Returns `-1` if the call encounters an error.
socket	Socket descriptor of a local socket that is receiving the data.
msg	A pointer to the data structure, `msghdr`, which has two fields called `msg_iov` and `msg_iovlen`. `Msg_iov` is a pointer to an array of data elements, and `msg_iovlen` contains the number of data elements in the array. See “Discussion” below for more information.
flags	The currently supported values for *flags* are:

`0`	No option.
`MSG_PEEK`	Option to preview incoming data. If this option is set on the `recv()` call, any data returned remains in the socket buffer as though it had not been read yet. The next `recv()` call returns the *same data*.

Discussion

`Recvmsg()` facilitates the receiving of vectored data. Unlike a data buffer, which is a structure containing actual data, a data vector is a structure that can *describe* several *data objects*. The description of each object consists of a byte address and a length. The byte address describes where the object is located and the length indicates how much data the object contains. Any kind of data object (arrays, portions of arrays, records, simple variables, etc.) can be described by a data vector.

When a data vector is used to identify data to be sent, it describes where the data is located. This is referred to as a *gathered write*. When a data vector is used to identify data to be received, it describes where the data is to be placed. This is referred to as a *scattered read*.

Using data vectors may be more efficient than using data buffers in certain circumstances. For example, a process that receives data from several different buffers must call `recv()` several times, or copy the data into a packing buffer prior to receiving it. However, if you use `recvmsg()` you may describe all of the buffers in one `recvmsg()` call.

Figure 4-2 is an example of a data vector and the data objects that it represents. The data vector describes the characters “HERE IS THE DATA.”

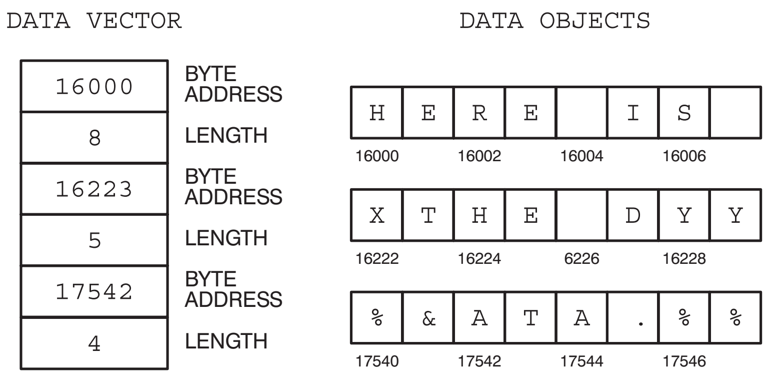


Figure 4-2. Vectored Data

Each data object is described by a byte address and a length. Each byte address/length pair of a data vector is stored in a structure of `iovec` type.

The parameter in the `recvmsg()` call relevant to vectored data is *msg*, which is a pointer to the `msghdr` structure. Within this `msghdr` structure are two fields: `msg_iov` and `msg_iovlen`, which are used for vectored data. `Msg_iov` is a pointer to an array of `iovec` records. `Msg_iovlen` contains the number of `iovec` elements in the array. Each `iovec` element contains the starting byte address of data to be received in `iov_base` and the number of bytes to be received in this data vector in `iov_len`.

The `msghdr` and `iovec` structures are shown below for C, Pascal, and FORTRAN.

C

```
struct msghdr {
    caddr_t    msg_name;            /* optional address */
    int        msg_namelen;         /* size of address */
    struct     iovec *msg_iov;      /* scatter/gather array */
    int        msg_iovlen;          /* # elements in msg_iov */
    caddr_t    msg_accrights;       /* access rights sent/rec'd */
    int        msg_accrightslen;
};

struct iovec {
    char *iov_base; /* starting byte address of buffer */
    int  iov_len;   /* size of buffer in bytes */
   };
```

Pascal

```
msghdr = RECORD
   CASE INTEGER OF
     1 : ( int1 : int );
     2 : ( msg_name          : int;     { Byte pointer to caddr_t }
           msg_namelen       : int;
           msg_iov           : int;     { Word pointer to iovec }
           msg_iovlen        : int;
           msg_accrights     : int;     { Byte pointer to caddr_t}
           msg_accrightslen : int);
   END;

iovec = RECORD
   iov_base : int;    { Byte pointer }
   iov_len  : int;
 END;
```

FORTRAN

```
INTEGER    MSGHDR(6)
INTEGER    MSG_NAME,MSG_NAMELEN,MSG_IOV,MSG_IOVLEN
INTEGER    MSG_ACCRIGHTS,MSG_ACCRIGHTSLEN
EQUIVALENCE  (MSGHDR(1),MSG_NAME)
EQUIVALENCE  (MSGHDR(2),MSG_NAMELEN)
EQUIVALENCE  (MSGHDR(3),MSG_IOV)
EQUIVALENCE  (MSGHDR(4),MSG_IOVLEN)
EQUIVALENCE  (MSGHDR(5),MSG_ACCRIGHTS)
EQUIVALENCE  (MSGHDR(6),MSG_ACCRIGHTSLEN)

INTEGER      IOVEC(2)
INTEGER      IOV_BASE,IOV_LEN
EQUIVALENCE  (IOV_BASE,IOVEC(1)),(IOV_LEN,IOVEC(2))
```

In order to use a data vector of 5 vectors, you should declare the following array of `iovec` records (shown below in C programming format):

```
struct iovec data_buffer[5];
```

In our example, since `msg_iov` is a pointer to the beginning of the `iovec` array:

```
msg_iov = &data_buffer[0];
```

`Msg_iovlen` is the number of relevant elements of the `iovec` array. So, if we wanted to use only the first 3 array elements of the `iovec` array, set `msg_iovlen = 3`.

If a connection has been gracefully released by the remote process and all the data has been received by the user, then `recvmsg()` returns a `0`.

HP 1000 Specific Information

- `Recvmsg()` on the HP 1000 supports the `MSG_PEEK` flag, while HP-UX supports `MSG_PEEK` and `MSG_OOB` flag options.
- HP 1000 currently does not support signals.
- BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

  The `ByteAdrOf()` function can be used to obtain the byte address of data objects which are accessed by character pointers in C (for example, `char *`*`variable`*).

  The RTE-A routine `AddressOf()` is used to obtain the address of data objects which are accessed by pointers other than character pointers.

  Use the `AddressOf()` function to obtain the *msg* pointer. Use the `ByteAdrOf()` function to obtain the data address to be put in the `iovec` element.

  Refer to "Working with Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in Extended Memory Area (EMA) cannot be accessed directly.

Error Returns

If the `recvmsg()` call is successful, it returns the number of bytes sent. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `recvmsg()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EINVAL]	Invalid `msg_iovlen` value. The maximum `iovec` elements allowed is 16.
[ENOTSOCK]	The *`socket`* parameter is not a valid socket descriptor.
[ECONNRESET]	Connection aborted by the remote process.
[ENOTCONN]	*`socket`* has not yet been connected.
[ESHUTDOWN]	The socket has already been shutdown for receives.
[EHOSTDOWN]	The network software is not running on the local host.
[EINPROGRESS]	The connection has not been established.
[EREMOTERELEASE]	The remote side has done a shutdown.

select()

Provides synchronous socket I/O multiplexing.

Syntax

```
result = select(count, reads, writes, exceptions, timeout)

int             result, count;
fd_set          *reads, *writes, *exceptions;
struct timeval  *timeout;
```

Parameters

result	Returns the number of socket descriptors contained in the `select()` call bitmasks. `-1` means an error has occurred. `0` means the time limit has expired and all the bitmasks are cleared.
count	Specifies the number of sockets for `select()` to examine. `Select()` examines socket descriptors from `0` to `(count-1)`. Currently, users are allowed a maximum of 31 socket descriptors, so the valid range for count is 1 to 31. Since socket descriptors are numbered starting with `0`, callers should specify *count* as their highest socket descriptor `+ 1`. (Note: *count* specifies the *number* of socket descriptors for selection. Hence, a count of 5 means that the `select()` call will examine socket descriptors from 0 through 4.)
reads	Pointer to a bitmask to specify which socket descriptors (from 0 to *count*`-1`) to select for reading. Set the bitmask to 0 with `FD_ZERO` if no descriptors need to be selected for reads. On return, it contains a pointer to the bitmask specifying which socket descriptors (from 0 to *count*`-1`) are ready for reading. Use the `FD_SET` macro and `fd_set` variable type to set the socket descriptors for reads before you issue the `select()` call. After issuing `select()`, use `FD_ISSET` to test for the bits in the bitmask. Refer to Section 6, “Socket Descriptor Utilities,” for more information on clearing, setting, and testing the bits in the bitmasks.
writes	Pointer to a bitmask to specify which socket descriptors (from 0 to *count*`-1`) to select for writing. Set the bitmask to 0 with `FD_ZERO` if no descriptors need to be selected for writes. On return, it contains a pointer to the bitmask specifying which socket descriptors (from 0 to *count*`-1`) are ready for writing.

Use the `FD_SET` macro and `fd_set` variable type to set the socket descriptors for writes before you issue the `select()` call. After issuing `select()`, use `FD_ISSET` to test for the bits in the bitmask. Refer to Section 6, "Socket Descriptor Utilities," for more information on clearing, setting, and testing the bits in the bitmasks.

exceptions Pointer to a bitmask to specify which socket descriptor (from 0 to *count*`-1`) to select for exceptional conditions. Set the bitmask to 0 with `FD_ZERO` if no descriptors need to be selected for exceptions.

On return, it contains a pointer to the bitmask specifying which socket descriptors (from 0 to *count*`-1`) have an exceptional condition pending.

Use the `FD_SET` macro and `fd_set` variable type to set the socket descriptors for exceptions before you issue the `select()` call. After issuing `select()`, use `FD_ISSET` to test for the bits in the bitmask. Refer to Section 6, "Socket Descriptor Utilities," for more information on clearing, setting, and testing the bits in the bitmasks.

The currently supported condition is when connections get terminated.

timeout Pointer to the `timeval` structure which specifies the interval in which to examine the socket descriptors. The `timeval` structure contains two fields: `tv_sec` and `tv_usec`.

The *timeout* parameter works as follows:

If both `tv_sec` and `tv_usec` are 0, `select()` returns immediately after checking the descriptors.

If either `tv_sec` or `tv_usec` is non-zero, then `select()` returns when one of the specified descriptors is ready for I/O, but `select()` does not wait beyond the specified amount of time (in number of seconds and microseconds).

If the `timeval` pointer itself is 0, then `select()` waits indefinitely and returns only when one of the selected descriptors is ready for I/O.

Discussion

The `select()` call can be used to avoid a situation in which a program blocks while waiting for an event to occur on a socket and holds up processing of the program.

The `select()` call is used to wait for any one of multiple events to occur and to notify the process when any or all of the events occur. For example, the `select()` call can be used to check if data is available to be received for a socket before issuing the `recv()` call; hence, avoiding a block on `recv()`.

To use `select()`, follow the procedure below:

1. Define a variable of type `fd_set` for a bitmask to be used in the `select()` call.
2. Clear all bits in the specified bitmask with `FD_ZERO`.
3. Use `FD_SET` to set the bit corresponding to a specific socket descriptor in the bitmask. Set the bits for the sockets that you want to select (for example, to select for readiness for reading).
4. Use the bitmask in the `select()` call.
5. Check *`result`* to see if the `select()` call returned successfully or an error.
6. Once `select()` returns successfully, use `FD_ISSET` to test if a specified bit in a bitmask is set (for example, if the socket corresponding to the bit is ready for reads).
7. The process can then proceed to handle the specified event.

Socket descriptor, *`s`*, is represented in the bitmask by bit `(1 << (s MOD 32))`.

When `select()` completes successfully, it returns the three bit masks modified as follows: for each socket descriptor less than or equal to (*`count`*`-1`), the corresponding bit in each mask is set if the bit was set upon entry and the socket descriptor is ready for reading, writing, or has an exception condition pending.

Any or all of *`reads`*, *`writes`*, and *`exceptions`* may be given as `0` if no socket descriptors are of interest. If all the masks are given as `0` and *`timeout`* is not zero, `select()` blocks for the time specified. If all the masks are given as `0`, and *`timeout`* is also zero, `select()` returns an error.

Bitmask Routines

It is recommended that the following routines be used to set, clear, or examine the bitmasks.

`FD_ZERO`	Procedure to clear all bits in the specified bitmask. *It is recommended that the bits be cleared in all bitmasks before using them; otherwise, you may get unexpected results.*
`FD_SET`	Procedure to turn on the specified socket's bit in the bitmask.
`FD_CLR`	Procedure to turn off the specified socket's bit in the bitmask.
`FD_ISSET`	Function to test the specified socket's bit, *`socket`*, in the bitmask, *`bitmask`*.

Refer to Section 6, "Socket Descriptor Utilities," for detailed information about these bitmask routines.

HP 1000 Specific Information

1. The HP 1000 does not support signals. Hence, it is recommended that programs always set the *`exceptions`* select bit in order to be informed about unexpected events.
2. In UNIX, if all the bitmasks are set to `0` and *`timeout`* is a zero pointer, then `select()` blocks until interrupted by a signal. Since RTE-A signals are currently not supported in the HP 1000 network, this setting is disallowed, and `select()` will return an error.
3. Both C and Pascal programmers must be careful about setting the appropriate bit in the bitmask. `Select()` expects the bitmasks to be set with the highest socket descriptor at the MSB of the bitmask word. Use of the bitmask utilities described above is strongly recommended.
4. Pascal and FORTRAN users must use the `AddressOf()` function to get the pointers for *`reads`*, *`writes`*, *`exceptions`*, and *`timeout`*. For more information on pointers, refer to "Working with Pointers in Pascal and FORTRAN," in Section 3.

Error Returns

If the `select()` call is successful, it returns the number of descriptors contained in the bitmasks. If the time limit expired, then `select()` returns `0` and all the bitmasks are cleared.

If `select()` failed, it returns a `-1`. The global variable *`errno`* provides information on the cause of the call's failure. The following table lists possible error returns from the `select()` call.

Error Mnemonic	Meaning
`[EINTERR]`	This error requires HP notification.
`[EINVAL]`	Invalid `timeval` variable used for *`timeout`*; or, the value of *`count`* is not valid; or, one or more of the socket descriptors in the bitmasks were invalid; or, the socket is not in a state that permits the type of select desired for that socket.
`[EHOSTDOWN]`	The network software on the local host is not running.
`[EREMOTERELEASE]`	The remote side has shut down.

Sends data on a socket. This call may be used by both the server and client processes. The `send()` call is used for stream sockets only.

Syntax

```
count = send(socket, buffer, len, flags)

int  count, socket, len;
char *buffer;
long flags;
```

Parameters

count	Returns the number of bytes actually sent. Returns `-1` if the call encounters an error.
socket	Socket descriptor of a local socket that is sending the data.
buffer	Byte pointer to a buffer which contains the data to be sent.
len	Number of bytes that need to be sent from the data buffer. In blocking mode, there is no restriction on the size of data to be sent except for that imposed by the system, which currently is 32767 bytes. In nonblocking mode, if the data is too long to pass atomically through the underlying protocol, the message is not transmitted, `-1` is returned, and *errno* is set to `EMSGSIZE`.
flags	Currently there are no supported options.

Discussion

The `send()` calls may only be used after connection has been established between two processes.

`Send()` blocks until the specified number of bytes have been queued to be sent, unless nonblocking I/O is enabled.

If nonblocking I/O is enabled, the `send()` call will complete in one of three ways:

- If there is enough space available in the system to buffer all the data, `send()` will complete successfully, having written out all of the data, and return the number of bytes written.
- If there is not enough space in the buffer to write out the entire request, `send()` will complete successfully, having written as much data as possible, and return the number of bytes it was able to write.
- If there is no space in the system to buffer any of the data, `send()` will return `-1`, having written no data, with *errno* set to `EAGAIN`.

Nonblocking I/O is enabled by setting flag option `O_NONBLOCK` using `fcntl()`.

By selecting the socket for write indication, the `select()` call may be used to determine when a socket has data available to be sent by a subsequent `send()` call. See the `select()` call for more information.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in Extended Memory Area (EMA) cannot be accessed directly.

HP 1000 Specific Information

- `Send()` requires the pointer to the data structure *`buffer`* to be a byte pointer. Pascal and FORTRAN users will have to use the `ByteAdrOf` routine to get the byte address of the start of the data. For more information on pointers, refer to "Working with Pointers in Pascal and FORTRAN" in Section 3.
- `Send()` on the HP 1000 does not support any flags.
- HP 1000 currently does not support signals. If a `send()` is attempted on a socket which has lost its connection to its peer, `send()` returns a `-1` with *errno* set to `EPIPE`.

Error Returns

If the `send()` call is successful, it returns the number of bytes sent. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `send()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EAGAIN]	In nonblocking mode, the socket does not have space to accept any data at all.
[EINVAL]	The *len* parameter contains a bad value. The input value for *len* must be equal to or greater than zero. Or, the socket is not a BSD IPC socket.
[EPIPE]	An attempt was made to send on a socket whose connection has been shutdown by the remote peer.
[EMSGSIZE]	In nonblocking mode, the socket requires that messages be sent atomically, and the message size exceeded the outbound buffer size.
[ENOTSOCK]	The *socket* parameter is not a valid socket descriptor.
[EOPNOTSUPP]	An invalid flag was specified. No flags are supported currently.

Error Mnemonic	Meaning
[ENOTCONN]	A send() on a socket that is not connected, or a send() on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
[ESHUTDOWN]	The socket has already been shutdown for send.
[EHOSTDOWN]	The network software on the local host is not running.
[EINPROGRESS]	Connection has not been fully established yet.

sendmsg()

Sends vectored data on a socket. This call may be used by both the server and client processes. The `sendmsg()` call is used for stream sockets only.

Syntax

```
count = sendmsg(socket, msg, flags)

int            count, socket;
struct msghdr  msg;
long           flags;
```

Parameters

count	Returns the number of bytes actually sent. Returns `-1` if the call encounters an error.
socket	Socket descriptor of a local socket that is sending the data.
msg	Pointer to a `msghdr` structure, which has two fields called `msg_iov` and `msg_iovlen`. `Msg_iov` is a pointer to an array of data elements, and `msg_iovlen` contains the number of data elements in the array. See "Discussion" below for more information.
flags	Currently there are no supported flags options.

Discussion

`Sendmsg()` facilitates the sending of vectored data. Unlike a data buffer, which is a structure containing actual data, a data vector is a structure that can *describe* several *data objects*. The description of each object consists of a byte address and a length. The byte address describes where the object is located and the length indicates how much data the object contains. Any kind of data object (arrays, portions of arrays, records, simple variables, etc.) can be described by a data vector.

When a data vector is used to identify data to be sent, it describes where the data is located. This is referred to as a *gathered write*. When a data vector is used to identify data to be received, it describes where the data is to be placed. This is referred to as a *scattered read*.

Using data vectors may be more efficient than using data buffers in certain circumstances. For example, a process that sends data from several different buffers must call `send()` several times, or copy the data into a packing buffer prior to sending it. However, if you use `sendmsg()` you may describe all of the buffers in one `sendmsg()` call.

Figure 4-3 is an example of a data vector and the data objects that it represents. The data vector describes the characters "HERE IS THE DATA."

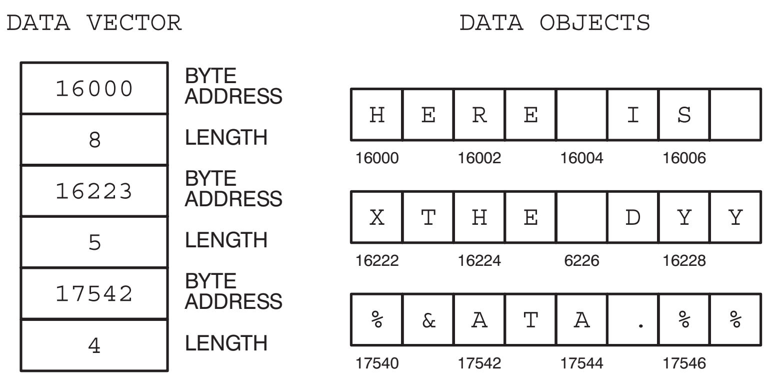


Figure 4-3. Vectored Data

Each data object is described by a byte address and a length. Each byte address/length pair of a data vector for `sendmsg()` is stored in a structure of `iovec` type.

The parameter in a `sendmsg()` call relevant to vectored data is *msg*, which is a pointer to the `msghdr` structure. Within this `msghdr` structure are two fields: `msg_iov` and `msg_iovlen`, which are used for vectored data. `Msg_iov` is a pointer to an array of `iovec` records. `Msg_iovlen` contains the number of `iovec` elements in the array. Each `iovec` element contains the starting byte address of data to be sent in `iov_base` and the number of bytes to be sent in this data vector in `iov_len`.

The `msghdr` and `iovec` structures are shown here in C, Pascal, and FORTRAN.

C

```
struct msghdr {
    caddr_t    msg_name;            /* optional address */
    int        msg_namelen;         /* size of address */
    struct     iovec *msg_iov;      /* scatter/gather array */
    int        msg_iovlen;          /* # elements in msg_iov */
    caddr_t    msg_accrights;       /* access rights sent/rec'd */
    int        msg_accrightslen;
};

struct iovec {
    char *iov_base; /* starting byte address of buffer */
    int iov_len;    /* size of buffer in bytes */
   };
```

Pascal

```
msghdr = RECORD
   CASE INTEGER OF
     1 : ( int1 : int );
     2 : ( msg_name          : int;     { Byte pointer to caddr_t }
           msg_namelen       : int;
           msg_iov           : int;     { Word pointer to iovec }
           msg_iovlen        : int;
           msg_accrights     : int;     { Byte pointer to caddr_t}
           msg_accrightslen  : int);
      END;

iovec = RECORD
   iov_base    : int;  { Byte pointer }
   iov_len     : int;
 END;
```

FORTRAN

```
INTEGER    MSGHDR(6)
INTEGER    MSG_NAME,MSG_NAMELEN,MSG_IOV,MSG_IOVLEN
INTEGER    MSG_ACCRIGHTS,MSG_ACCRIGHTSLEN
EQUIVALENCE  (MSGHDR(1),MSG_NAME)
EQUIVALENCE  (MSGHDR(2),MSG_NAMELEN)
EQUIVALENCE  (MSGHDR(3),MSG_IOV)
EQUIVALENCE  (MSGHDR(4),MSG_IOVLEN)
EQUIVALENCE  (MSGHDR(5),MSG_ACCRIGHTS)
EQUIVALENCE  (MSGHDR(6),MSG_ACCRIGHTSLEN)

INTEGER      IOVEC(2)
INTEGER      IOV_BASE,IOV_LEN
EQUIVALENCE  (IOV_BASE,IOVEC(1)),(IOV_LEN,IOVEC(2))
```

In order to use a data vector of 5 vectors, you should declare the following array of `iovec` records (shown below in C programming format):

```
struct iovec data_buffer[5];
```

In our example, since `msg_iov` is a pointer to the beginning of the `iovec` array:

```
msg_iov = &data_buffer[0];
```

`Msg_iovlen` is the number of relevant elements of the `iovec` array. So, if we wanted to use only the first 3 array elements of the `iovec` array, set `msg_iovlen` to `3`.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in Extended Memory Area (EMA) cannot be accessed directly.

HP 1000 Specific Information

- `Sendmsg()` currently does not support any flag options, while HP-UX supports `MSG_OOB`.
- HP 1000 currently does not support signals. If `sendmsg()` attempts a send on a socket that has lost its connection to its peer, `sendmsg()` returns `-1` with *errno* set to `EPIPE`.
- BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

  The `ByteAdrOf()` function can be used to obtain the byte address of data objects which are accessed by character pointers in C (that is, `char *variable`).

  The RTE-A routine `AddressOf()` is used to obtain the address of data objects which are accessed by pointers other than character pointers.

  Use the `AddressOf()` function to obtain the *msg* pointer. Use the `ByteAdrOf()` function to obtain the data address to be put in the `iovec` element.

  Refer to "Working with Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Error Returns

If the `sendmsg()` call is successful, it returns the number of bytes sent. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `sendmsg()` call.

Error Mnemonic	Meaning
[EINTERR]	This error requires HP notification.
[EINVAL]	The `msg_iovlen` is invalid. The maximum number of iovec elements is 16.
[EPIPE]	An attempt was made to send on a socket that was connected, but the connection has been shutdown either by the remote peer or by this side of the connection.
[EMSGSIZE]	The socket requires that messages be sent atomically, and the message size exceeded the outbound buffer size.
[ENOTSOCK]	The *socket* parameter is not a valid socket descriptor.
[EOPNOTSUPP]	An invalid flag was specified. No flags are supported currently.
[ENOTCONN]	A `send()` on a socket that is not connected, or a `send()` on a socket that has not completed the connect sequence with its peer, or is no longer connected to its peer.
[EHOSTDOWN]	The network software on the local host is not running.

sendto()

Sends data on a socket. The `sendto()` call may be used by both the server and client processes.

Syntax

```
count = sendto(socket, buffer, len, flags, addr, addrlen)

int                 count, socket, len, addrlen;
char                *buffer;
long                flags;
struct sockaddr_in  *addr;
```

Parameters

count	Returns the number of bytes actually sent. Returns `-1` if the call encounters an error.
socket	Socket descriptor of a local socket that is sending the data.
buffer	Byte pointer to a buffer which contains the data to be sent.
len	Number of bytes that need to be sent from the data buffer. The size of data that can be sent is limited to 32767 bytes. However, the HP 1000 cannot receive UDP datagrams larger than 9216 bytes.
flags	Currently there are no supported options.
addr	Pointer to a structure containing the address of the remote socket to which the data will be sent. The socket address should be of *sockaddr_in* type, which is described in “Preparing Socket Addresses” in Section 3.
addrlen	Length, in bytes, of the address structure specified by *addr* (for example, length of structure *sockaddr_in*, which is 16 bytes). *Addrlen* should be at least 16 bytes.

Discussion

The `sendto()` call may only be used after the socket has been bound to an address by `bind()`.

Caution Because BSD IPC uses 16-bit addressing, BSD IPC cannot access data with 32-bit addressing. Therefore, data in Extended Memory Area (EMA) cannot be accessed directly.

HP 1000 Specific Information

`Sendto()` requires the pointer to the data structure buffer to be a byte pointer. Pascal and FORTRAN users will have to use the `ByteAdrOf` routine to get the byte address of the start of the data. For more information on pointers, refer to "Working with Pointers in Pascal and FORTRAN" in Section 3.

`Sendto()` on the HP 1000 does not support any `flags`.

Error Returns

If the `sendto()` call is successful, it returns the number of bytes sent. If it failed, it returns a `-1`. The global variable `errno` provides information on the cause of the call's failure. The following table lists possible error returns from the `sendto()` call.

Error Mnemonic	Meaning
`[EINTERR]`	This error requires HP notification.
`[EINVAL]`	Invalid *`len`* value `<0`. Or, the socket is not a BSD IPC datagram-type socket.
`[ENOTSOCK]`	The *`socket`* parameter is not a valid socket descriptor.
`[EOPNOTSUPP]`	An invalid flag was specified. No flags are currently supported.
`[EHOSTDOWN]`	The network software on the local host is not running.
`[EHOSTUNREACH]`	There is no route to the host.

setsockopt()

Sets socket options.

Syntax

```
result = setsockopt(socket, level, optname, optval, optlen)

int  result, socket, level, optname, optlen;
char *optval;
```

Parameters

result	`0` if `setsockopt()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
level	The protocol level at which the socket option resides. To specify "socket" level, *level* should be `SOL_SOCKET`. To specify "TCP" level, *level* should be `IPPROTO_TCP`.
optname	Socket option name. The following options are supported for "socket" level (`SOL_SOCKET`) options:

`SO_KEEPALIVE`	(Toggle option) Sets a timer for 90 minutes for connected sockets. After 90 minutes expire, and if the connection has been idle during this period, `SO_KEEPALIVE` forces a transmission every 60 seconds for up to 7 minutes, after which the idle connection is shut down. In summary, `SO_KEEPALIVE` allows an idle period of 97 minutes before connection shutdown. If this option is toggled off, an indefinite idle time is allowed. This option is set by default.
`SO_REUSEADDR`	(Toggle option) Allows local socket address reuse. This allows multiple sockets to be bound to the same local port address.

This option modifies the rules used by `bind()` to validate local addresses. `SO_REUSEADDR` allows more than one socket to be bound to the same port number at the same time; however, it only allows one single socket to be actively listening for connection requests on the port number. The host will still check at connection time to be sure any other socket with the same local address and local port does not have the same remote address and remote port. `Connect()` fails if the uniqueness requirement is violated.

`SO_RCVBUF`	Changes the buffer size of a socket's receive socket buffer. The default buffer size is 4096 bytes. The maximum buffer size is 32766 bytes. A stream socket's buffer size can be increased or decreased only prior to establishing a connection.
`SO_SNDBUF`	Changes the buffer size of a socket's send socket buffer. The default buffer size is 4096 bytes. The maximum buffer size is 32766 bytes. A stream socket's buffer size can be increased or decreased only prior to establishing a connection.

The following options are supported for "TCP" level (`IPPROTO_TCP`) options:

`TCP_MAXSEG`	Returns the maximum segment size in use for the socket. The value for this option can only be examined, it cannot be set. If the socket is not yet connected, TCP returns a default size of 512 bytes.
`TCP_NODELAY`	(Toggle option) Instructs TCP to send data as soon as it receives it and to bypass the buffering algorithm that tries to avoid numerous small packets from being sent over the network.

`optval` Byte pointer to a value or boolean flag for the specified option.

(Since *`optval`* is a byte pointer, Pascal and FORTRAN users should use the `ByteAdrOf()` function to get the byte address of the option value.) Although *`optval`* is a byte pointer, the value itself is not terminated by a `\0`.

`optlen` Size, in bytes, of *`optval`*.

Discussion

To get the status of current socket options, use `getsockopt()`. To set socket options, use `setsockopt()`.

There are two kinds of socket options: boolean (toggle) options and non-boolean options.

Boolean options are options that can be set on or off. To set a boolean option, set the value of *optval* to a non-zero value. To turn off an option, set the value of *optval* to 0. Currently the supported boolean options are: SO_KEEPALIVE, SO_REUSEADDR, and TCP_NODELAY.

Non-boolean options contain specific values. Non-boolean options use *optval* and *optlen* to pass information. The parameter *optval* is a pointer to a character array that specifies the value for the option. The parameter *optlen* specifies the size, in bytes, of *optval*.

HP 1000 Specific Information

1. HP 1000 does not allow socket buffer size to be changed after a connection is established. On the HP-UX system, the buffer size can be increased after the connection is established.

2. The SO_LINGER option (available in HP-UX) is not provided on the HP 1000.

3. BSD IPC uses C programming semantics. Many of the parameters in the BSD IPC calls use pointers to variables. Consequently, two routines are provided for Pascal and FORTRAN users to handle pointers.

   The ByteAdrOf() function can be used to obtain the byte address of data objects which are accessed by character pointers in C (that is, char **variable*).

   The RTE-A routine AddressOf() is used to obtain the address of data objects which are accessed by pointers other than character pointers.

   Use the ByteAdrOf() function to obtain the byte pointer for *optval*.

   Refer to "Working with Pointers in Pascal and FORTRAN" in Section 3 for more information on pointers.

Error Returns

If the setsockopt() call is successful, it returns a 0. If it failed, it returns a -1. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the setsockopt() call.

Error Mnemonic	Meaning
[EFAULT]	The *optval* or optlen parameter is not valid.
[EINVAL]	The specified option level is unknown. Or the socket is not a BSD IPC socket.
[ENOTSOCK]	*socket* is not a valid socket.
[EOPNOTSUPP]	Unknown option was specified.
[EISCONN]	A connection has already been established for the socket. The send and receive buffer sizes for the socket can only be changed prior to establishing a connection.
[EHOSTDOWN]	The network software on the local host is not running.

Shuts down a socket. This call may be used by either the server or client process.

Syntax

```
result = shutdown(socket, how)

int result, socket, how;
```

Parameters

result	`0` if `shutdown()` is successful. `-1` if a failure occurs.
socket	Socket descriptor of local socket to be shut down.
how	Method of shutdown, as follows:

`0`	Disallows further receives. Once the socket has been shut down for receives, all further `recv()` calls return `-1`, with *errno* set to `ESHUTDOWN`.
`1`	Disallows further sends. Once the socket has been shut down for sends, all further `send()` calls return `-1`, with *errno* set to `ESHUTDOWN`.
`2`	Disallows further sends and receives.

Discussion

Multiple shutdowns on a connected socket or shutdown on a socket that is not connected will return errors.

For `SOCK_STREAM` sockets, a shutdown results in the connection being closed gracefully. This means that although the `shutdown()` call returns immediately, the Transport layer will make a best effort to get any buffered data across to the remote side in the right sequence.

If a `shutdown()` is performed on a socket that has a `listen()` pending on it, that socket becomes fully shutdown when *how* equals `1`.

The best way to shut down and end a session without losing data is summarized below:

1. With the `send()` call, the sending side sends an “I am finished” message. This message is defined by the application designer.

2. The receiving side reads this “I am finished” message with the `recv()` call. It finishes up any unfinished tasks (e.g., sends remaining data).
3. The receiving side sends “I am finished” message with `send()` and shuts down its socket with the `shutdown()` call.
4. The sending side receives the “I am finished” message.
5. The sending side frees its socket resources by issuing a `shutdown()` call.

For datagram sockets, `shutdown()` only releases the socket. There is no connection to shut down. The parameter *how* should be set to `2`.

HP 1000 Specific Information

1. HP 1000 will release the socket descriptor when both the send and receive sides of the connection are closed. This can be done via separate shutdown calls that close the send and receive sides of the connection or by setting *how* to `2` in a single call. UNIX does not release the socket descriptor when the socket is completely shutdown through this call. A call to `close()` must be invoked to clear the UNIX socket descriptor.
2. Once the receive side is shut down, all further `recv()` calls on the HP 1000 socket will return an error. UNIX returns an `EOF` condition.
3. Once the send side is shut down, all further `send()` calls on the HP 1000 socket will return an `ESHUTDOWN` error. No signal is sent to the process. UNIX, however, sends the `SIGPIPE` signal.

Error Returns

If the `shutdown()` call is successful, it returns `0`. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call’s failure. The following table lists possible error returns from the `shutdown()` call.

Error Mnemonic	Meaning
`[EINVAL]`	The specified socket is not a BSD IPC socket.
`[ENOTSOCK]`	The *socket* parameter is not a valid socket descriptor.
`[ESHUTDOWN]`	The network software on the local host is not running.

Creates a socket, an endpoint for communication, and returns a socket descriptor for the socket. This must be the first BSD IPC call used in the process. Both server and client processes need to create a socket with the `socket()` call.

Syntax

```
socket = socket(af, type, protocol)

int socket, af, type protocol;
```

Parameters

socket	Socket descriptor for the newly-created socket. It is an integer with a valid range of `0` to `30`.
	This socket descriptor is used in subsequent BSD IPC calls to reference this socket.
	If the call fails, a `-1` is returned in *socket* and the global variable *errno* contains the error code.
af	Address family for the socket being created. It must be set to `AF_INET`, for Internet address family.
	The address family defines the address format used in socket operations. The `AF_INET` address family uses an address structure (`sockaddr_in`) of 16 bytes. Refer to “Address Family” in the Glossary for more information.
type	Type of socket being created. It must be set to `SOCK_STREAM` or `SOCK_DGRAM`.
	The socket type specifies the semantics of communication for the socket. A `SOCK_STREAM` type provides sequenced, reliable, two-way, connection-based bytes streams. Refer to “Socket Type” in Appendix D for more information.
protocol	Underlying protocol to be used. `0` causes the system to choose a protocol type to use.

Discussion

Sockets of `SOCK_STREAM` type are full-duplex byte streams. A stream socket must be in a connected state before any data can be sent or received on it. A connection to another socket is created with the `connect()` call on the client side and a corresponding `accept()` call on the server side. Once connected, data can be transferred using the `send()` and `recv()` calls. When a session has been completed, a `shutdown()` call can be performed.

Transmission Control Protocol (TCP), the communication protocol used to implement `SOCK_STREAM` for `AF_INET` sockets, ensures that data is not lost or duplicated. If a peer has buffer space for data and the data cannot be transmitted within a reasonable length of time, the connection is considered broken and the next `recv()` call indicates an error with *errno* set to `ETIMEDOUT`.

If the socket level option `SO_KEEPALIVE` is set, the TCP protocol keeps inactive socket connections, which have been idle for a period of 90 minutes, active by forcing transmission every 60 seconds for up to 7 minutes. These transmissions are not visible to users, and cannot be read by a `recv()` call. The `SO_KEEPALIVE` option can be set for a socket via the `setsockopt()` call, described earlier in this section.

An error is indicated if no response can be elicited on an otherwise idle connection for an extended period (e.g., 6 minutes). An `EPIPE` error is returned in *errno* if a process sends on a broken stream. Zero bytes read is returned if a process tries to receive on a broken stream.

Error Returns

If the `socket()` call is successful, it returns an integer between `0` and `30` in *socket*, specifying the socket descriptor. If it failed, it returns a `-1`. The global variable *errno* provides information on the cause of the call's failure. The following table lists possible error returns from the `socket()` call.

Error Mnemonic	Meaning
[ENFILE]	Currently there are no resources available.
[EMFILE]	The per-process socket descriptor table is full.
[EPROTONOSUPPORT]	The specified protocol is not supported.
[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.
[EAFNOSUPPORT]	The specified address family is not supported on this version of the system.
[ENOBUFS]	No buffer space is available. The socket cannot be created.
[EHOSTDOWN]	The network software is not running on the local host.

5

HP 1000 BSD IPC Utilities

This section provides reference information on Berkeley Socket utilities supported on the HP 1000. These Berkeley Socket utilities are used to

- Manipulate and return information on the following database files: `/etc/hosts`, `/etc/networks`, `/etc/protocols`, and `/etc/services`.
- Obtain the socket address of the local and peer sockets.
- Manipulate Internet (IP) addresses and ASCII strings that represent IP addresses in Internet "dot" notation.
- Convert bytes from network order to host order and vice versa. (HP 1000, HP 9000, and TCP/IP protocols all use network order. These functions are provided for portability.)

Special Considerations

In order to successfully use the Berkeley Socket (BSD IPC) utilities, you must be aware of the following:

- The `/ETC` directory must be created before you run the BSD IPC utilities.
- Most of the utilities return pointers to structures that are dynamically allocated. If any of these functions that allocate dynamic memory are called repeatedly without freeing the allocated memory, they will eventually fail and return a null pointer. See the following discussion on "Releasing Dynamically Allocated Memory" for more information.

Releasing Dynamically Allocated Memory

The BSD IPC utility functions that allocate dynamic memory include:

```
gethostbyaddr   getnetbyaddr   getprotobyname     getservbyname
gethostbyname   getnetbyname   getprotobynumber   getservbyport
gethostent      getnetent      getprotent         getservent
                                                  inet_ntoa
```

In order to release space dynamically allocated by the above utility functions, use `free()` from the standard C library (generally found in `HPC.LIB`).

FORTRAN program fragments showing how to free the memory used by the specified BSD IPC utility functions above are given in the following subsection. FORTRAN users must be careful to always pass a byte address to `free()`. When `free()` is called from FORTRAN, there is no automatic type casting on the pointer parameter as done in C programming.

Examples

The FORTRAN program fragment below shows how to release the memory dynamically allocated for the `hostent` structure. Functions that return a pointer to a `hostent` structure are `gethostbyaddr`, `gethostbyname`, and `gethostent`.

```
                  .
                  .
                  .
   $alias /MEM/ = 0
          include socket.ftni
          common /MEM/ MEM(0:1)
          integer*2 ptr, hostptr, netptr, protoptr, servptr, char_ptr
                  .
                  .
          hostptr = gethostent()        ! get hostent struct
                  .                     ! free hostent struct
          ptr = MEM(hostptr+1)          ! h_aliases array
          do while (MEM(ptr) .ne. 0)
             call free(MEM(ptr))
             ptr = ptr + 1
          end do
                  .
                  .
          ptr = MEM(hostptr+4)          ! h_addr_list array
          do while (MEM(ptr) .ne. 0)
             call free(MEM(ptr))
             ptr = ptr + 1
          end do
                  .
                  .
          call free(MEM(hostptr))       ! h_name
          call free(MEM(hostptr+1)*2)   ! h_aliases
          call free(MEM(hostptr+4)*2)   ! h_addr_list
          call free(hostptr*2)          ! hostent

                  .
                  .
```

The FORTRAN example below shows how to release the memory dynamically allocated for the `netent` structure. Functions that return a pointer to a `netent` structure are `getnetbyaddr`, `getnetbyname`, and `getnetent`.

```
                 .
                 .
                 .
   $alias /MEM/ = 0
          include socket.ftni
          common /MEM/ MEM(0:1)
          integer*2 ptr, hostptr, netptr, protoptr, servptr, char_ptr
                 .
                 .
          netptr = getnetent()         ! get netent struct
                 .                     ! free netent struct
          ptr = MEM(netptr+1)          ! n_aliases array
          do while (MEM(ptr) .ne. 0)
             call free(MEM(ptr))
             ptr = ptr + 1
          end do
                 .
                 .
          call free(MEM(netptr))       ! n_name
          call free(MEM(netptr+1)*2)   ! n_aliases
          call free(netptr*2)          ! netent

                 .
                 .
```

The FORTRAN example below shows how to release the memory dynamically allocated for the `protoent` structure. Functions that return a pointer to a `protoent` structure are `getprotobyname`, `getprotobynumber`, and `getprotoent`.

```
                .
                .
$alias /MEM/ = 0
      include socket.ftni
      common /MEM/ MEM(0:1)
      integer*2 ptr, hostptr, netptr, protoptr, servptr, char_ptr
             .
             .

      protoptr = getprotoent()      ! get protoent struct
             .                      ! free protoent struct
      ptr = MEM(protoptr+1)         ! p_aliases array
      do while (MEM(ptr) .ne. 0)
         call free(MEM(ptr))
         ptr = ptr + 1
      end do
             .
             .

      call free(MEM(protoptr))      ! p_name
      call free(MEM(protoptr+1)*2)  ! p_aliases
      call free(protoptr*2)         ! protoent

             .
             .
```

The FORTRAN example below shows how to release the memory dynamically allocated for the `servent` structure. Functions that return a pointer to a `servent` structure are `getservbyname`, `getservbyport`, and `getservent`.

```
                 .
                 .
                 .
   $alias /MEM/ = 0
         include socket.ftni
         common /MEM/ MEM(0:1)
         integer*2 ptr, hostptr, netptr, protoptr, servptr, char_ptr
                 .
                 .
         servptr = getservent()       ! get servent struct
                 .                    ! free servent struct
         ptr = MEM(servptr+1)         ! s_aliases array
         do while (MEM(ptr) .ne. 0)
            call free(MEM(ptr))
            ptr = ptr + 1
         end do
                 .
                 .
         call free(MEM(servptr))       ! s_name
         call free(MEM(servptr+1)*2)   ! s_aliases
         call free(MEM(servptr+3))     ! s_proto
         call free(servptr*2)          ! servent

                 .
                 .
                 .
```

The FORTRAN example below shows how to release the memory dynamically allocated by `inet_ntoa`.

```
                 .
                 .
                 .
   $alias /MEM/ = 0
         include socket.ftni
         common /MEM/ MEM(0:1)
         integer*2 ptr, hostptr, netptr, protoptr, servptr, char_ptr
                 .
                 .
         char_ptr = inet_ntoa(ipaddr)
                 .
                 .
         call free(char_ptr)
                 .
                 .
                 .
```

The utilities described in this section are in alphabetical order for easy referencing. Table 5-1 lists the utilities covered in this section.

Table 5-1. Berkeley Socket Utilities

BSD IPC Utilities	Description
`endhostent()`	Closes the `/etc/hosts` file.
`endnetent()`	Closes the `/etc/networks` file.
`endprotoent()`	Closes the `/etc/protocols` file.
`endservent()`	Closes the `/etc/services` file.
`gethostbyaddr()`	Returns host information from the specified IP address.
`gethostbyname()`	Returns host information from the specified host.
`gethostent()`	Reads the next line of the `/etc/hosts` file and returns host information.
`getlocalname()`	Returns the name of the host/local system.
`getnetbyaddr()`	Returns network information of the specified network address.
`getnetbyname()`	Returns network information of the specified network.
`getnetent()`	Reads the next line of the `/etc/networks` file and returns network information.
`getpeername()`	Returns the address of the peer socket that is connected to the specified local socket.
`getprotobyname()`	Returns protocol information of the specified protocol name.
`getprotobynumber()`	Returns protocol information of the specified protocol number.
`getprotoent()`	Reads the next line of the `/etc/protocols` and returns protocol information.
`getservbyname()`	Returns service information of the specified service name.
`getserbyport()`	Returns service information of the specified port number.
`getservent()`	Reads the next line of the `/etc/services` file and returns service information.
`getsockname()`	Returns the socket address of the specified local socket.
`htonl()`	Converts a 32-bit quantity from host order to network order.
`htons()`	Converts a 16-bit quantity from host order to network order.
`inet_addr()`	Interprets character strings representing numbers in the Internet standard "dot" notation, and returns numbers suitable for use as Internet (IP) addresses.
`inet_lnaof()`	Breaks apart Internet (IP) addresses, and returns the local node address portion.
`inet_makeaddr()`	Constructs an Internet (IP) address from an Internet network number and a local node address.
`inet_netof()`	Breaks apart the Internet (IP) address, and returns the network number.
`inet_network()`	Interprets character strings representing numbers in the Internet standard "dot" notation, and returns numbers suitable for use as Internet network numbers.

BSD IPC Utilities	Description
`inet_ntoa()`	Takes an Internet address and returns an ASCII string representing the address in “dot” notation.
`ntohl()`	Converts a 32-bit quantity from network order to host order.
`ntohs()`	Converts a 16-bit quantity from network order to host order.
`sethostent()`	Opens and rewinds the `/etc/hosts` file.
`setnetend()`	Opens and rewinds the `/etc/networks` file.
`setprotoent()`	Opens and rewinds the `/etc/protocols` file.
`setservent()`	Opens and rewinds the `/etc/services` file.

endhostent()

Closes the `/etc/hosts` file.

Syntax

```
    result= endhostent()

    int   result
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.

Discussion

`Endhostent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/hosts` file. The following utilities manipulate the `/etc/hosts` file.

- `endhostent()`—closes the `/etc/hosts` file.
- `gethostbyaddr()`—returns host information from the specified IP address.
- `gethostbyname`—returns host information from the specified host name.
- `gethostent()`—reads the next line of the `/etc/hosts` file and returns host information on that host.
- `sethostent()`—opens and rewinds the `/etc/hosts` file.

Closes the `/etc/networks` file.

Syntax

```
result= endnetent()

int  result
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.

Discussion

`Endnetent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/networks` file. The following utilities also manipulate the `/etc/networks` file.

- `getnetbyaddr()`—sequentially searches from the beginning of the `/etc/networks` file until a network number matches the specified parameter or until EOF is encountered.
- `getnetbyname()`—sequentially searches from the beginning of the `/etc/networks` file until a network name or alias matches the specified parameter or until EOF is encountered.
- `getnetent()`—returns the next line of the entry from the `/etc/networks` file, opening the file if necessary.
- `setnetent()`—opens and rewinds the `/etc/networks` file.

endprotoent()

Closes the `/etc/protocols` file.

Syntax

```
result= endprotoent()

int  result
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.

Discussion

`Endprotoent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/protocols` file. The following utilities also manipulate the `/etc/protocols` file.

- `getprotobyname()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol name or alias matches the specified parameter or until EOF is encountered.
- `getprotobynumber()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol number matches the specified parameter or until EOF is encountered.
- `getprotoent()`—returns the next line of the entry from the `/etc/protocols` file, opening the file if necessary.
- `setprotoent()`—opens and rewinds the `/etc/protocols` file.

Closes the `/etc/services` file.

Syntax

```
    result= endservent()

    int   result
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.

Discussion

`Endservent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/services` file. The following utilities also manipulate the `/etc/services` file.

- `getservbyname()`—sequentially searches from the beginning of the `/etc/services` file until a service name or alias matches the specified parameter or until EOF is encountered.
- `getservbyport()`—sequentially searches from the beginning of the `/etc/services` file until a port number matches the specified parameter or until EOF is encountered.
- `getservent()`—returns the next line of the entry from the `/etc/services` file, opening the file if necessary.
- `setservent()`—opens and rewinds the `/etc/services` file.

gethostbyaddr()

Returns host information on the host with the specified IP address.

Syntax

```
host = gethostbyaddr(addr, len, type)

struct hostent *host;
char           *addr;
int            len, type;
```

Parameters

host	Pointer to `hostent` structure that contains the host information.
	The `hostent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct hostent {
  char *h_name;        /* official name of host */
  char **h_aliases;    /* alias list */
  int  h_addrtype;     /* host address type = AF_INET */
  int  h_length;       /* length of address = 4 bytes */
  char **h_addr_list;  /* list of addresses */
                       /* NULL terminates the list */
};
```

addr	Character pointer to a variable that contains the IP address of the host. The IP address must be in network order (that is, bytes ordered from left to right).
len	Number of bytes of an IP address.
type	The type of socket address family used. Must be set to `AF_INET`.

Discussion

`Gethostbyname()` and `gethostbyaddr()` both return a pointer to a structure of type `hostent`, which contains the broken-out fields of a line in the `/etc/hosts` file. `Gethostbyaddr()` sequentially searches the `/etc/hosts` file for an IP address matching the one specified in *addr* or until EOF is encountered.

`Gethostbyname()` sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matches the specified parameter or until EOF. Names are matched in a case-insensitive manner.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to the 32-bit IP address, *`addr`*.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/hosts` file.

Error Returns

If successful, `gethostbyaddr()` returns a pointer to the requested `hostent` structure. It returns NULL if the *`addr`* parameter cannot be found in the `/etc/hosts` file, or if *`addr`* or *`len`* is invalid.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

gethostbyname()

Returns host information on the host with the specified host name.

Syntax

```
host = gethostbyname(name)

struct hostent   *host;
char             *name;
```

Parameters

host Pointer to the *hostent* structure that contains host information.

The hostent structure is defined in the include files <netdb.h>, SOCKET.PASI, and SOCKET.FTNI, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct hostent {
  char *h_name;        /* official name of host */
  char **h_aliases;    /* alias list */
  int  h_addrtype;     /* host address type = AF_INET */
  int  h_length;       /* length of address = 4 bytes */
  char **h_addr_list;  /* list of addresses */
                       /* NULL terminates the list */
};
```

name Pointer to string that contains the name of the host about whom you need to obtain information. Terminate the string with the \0 character.

Discussion

Gethostbyname() and gethostbyaddr() both return a pointer to a structure of type hostent, which contains the broken-out fields of the /etc/hosts file. Names are matched in a case-insensitive manner.

Gethostbyname() sequentially searches from the beginning of the file until a host name (among either the official names or the aliases) matches the specified parameter in or until EOF. gethostbyaddr() sequentially searches the /etc/hosts file for an IP address matching the one specified in *addr*.

Pascal and FORTRAN programs need to use the ByteAdrOf() function to get the character pointer for *name*.

Refer to Appendix B, "Database and Header Files," for detailed information on the /etc/hosts file.

Error Returns

If successful, `gethostbyname()` returns a pointer to the requested `hostent` structure. It returns NULL if the host name cannot be found in the `/etc/hosts` file.

Releasing Memory

Refer to "Releasing Dynamically Allocated Memory" earlier in this section for information on releasing memory allocated by this call.

gethostent()

Reads the next line of the `/etc/hosts` file and returns the host information.

Syntax

```
host = gethostent()

struct hostent *host;
```

Parameters

host Pointer to a `hostent` structure containing host information.

The `hostent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct hostent {
  char *h_name;        /* official name of host */
  char **h_aliases;    /* alias list */
  int  h_addrtype;     /* host address type = AF_INET */
  int  h_length;       /* length of address = 4 bytes */
  char **h_addr_list; /* list of addresses */
                       /* NULL terminates the list */
};
```

`Gethostent()`, `gethostbyaddr()`, and `gethostbyname` each return a pointer to a structure of type `hostent`, which contains the broken-out fields of a line in the network database file, `/etc/hosts`.

`Gethostent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/hosts` file. The following utilities manipulate the `/etc/hosts` file.

- `endhostent()`—closes the `/etc/hosts` file.
- `gethostbyaddr()`—returns host information from the specified IP address.
- `gethostbyname`—returns host information from the specified host name.
- `gethostent()`—reads the next line of the `/etc/hosts` file and returns host information on that host.
- `sethostent()`—opens and rewinds the `/etc/hosts` file.

Error Returns

`Gethostent()` returns a null pointer (0) on EOF or when it is unable to open the `/etc/hosts` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getlocalname()

Returns the name of the host/local system.

Syntax

```
error = getlocalname(hostname)

character*(*)  hostname;
integer*2      error;
```

Parameters

hostname	Returns a FORTRAN character string for the local node name.
error	Is an integer that returns 0 for success or non-zero if the network is down.

Discussion

The *hostname* string returned by `getlocalname()` is limited to 50 characters.

Note that the syntax is defined for FORTRAN. Character strings need special attention when they are passed between C, FORTRAN, and Pascal programs. To understand the differences between C, FORTRAN, and Pascal character strings, refer to the following manuals:

HP C/1000 Reference Manual, part number 92571-90001;

FORTRAN 77 Reference Manual, part number 92836-90001;

Pascal/1000 Reference Manual, part number 92833-90001.

Discussion on character strings is also available in the *RTE-A Programmer's Reference Manual*, part number 92077-90007.

Returns network information on the specified network number.

Syntax

```
network = getnetbyaddr(net, type)

struct netent *network;
long           net;
int           type;
```

Parameters

network	Pointer to a `netent` structure that contains network information returned by `getnetbyaddr()`. The `netent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct    netent {
    char         *n_name;       /* official name of net */
    char        **n_aliases ;   /* alias list */
    int           n_addrtype;   /* net address type */
    unsigned long n_net;        /* network # */
};
```

net	Network number from which to get network information.
type	The socket address family type. It must be set to `AF_INET`.

Discussion

`Getnetbyaddr()`, `getnetbyname()`, and `getnetent()` each return a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database file, `/etc/networks`.

`Getnetbyaddr()` sequentially searches from the beginning of the `/etc/networks` file until a network number matching its parameter *net* is found, or until EOF is encountered. The parameter *net* for the network number must be in network order (that is, bytes ordered from left to right).

`Getnetbyaddr()` is one of the Berkeley Socket utilities used to manipulate the `/etc/networks` file. The following utilities also manipulate the `/etc/networks` file.

- `endnetent()`—closes the `/etc/networks` file.
- `getnetbyname()`—sequentially searches from the beginning of the `/etc/networks` file until a network name or alias matches the specified parameter or until EOF is encountered.

- `getnetent()`—returns the next line of the entry from the `/etc/networks` file, opening the file if necessary.
- `setnetent()`—opens and rewinds the `/etc/networks` file.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/networks` file.

Error Returns

`Getnetbyaddr()` returns a null pointer (`0`) on EOF or when it is unable to open the `/etc/networks` file. `Getnetbyaddr()` also returns a null pointer if the parameter *`type`* is invalid (that is, it is not set to “`AF_INET`”).

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

Returns network information on the specified network name.

Syntax

```
network = getnetbyname(name)

struct netent   *network;
char            *name;
```

Parameters

network	Pointer to a `netent` structure that contains network information returned by `getnetbyname()`.

The `netent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct    netent {
    char         *n_name;       /* official name of net */
    char        **n_aliases ;   /* alias list */
    int           n_addrtype;   /* net address type */
    unsigned long n_net;        /* network # */
};
```

name	Pointer to string that contains the network name from which to get network information. Terminate the name string with the character `\0`.

Discussion

`Getnetbyname()`, `getnetbyaddr()`, and `getnetent()` each return a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database file, `/etc/networks`.

`Getnetbyname` sequentially searches from the beginning of the `/etc/networks` file until a network name (among either the official names or the aliases) matches the specified parameter *name*, or until EOF is encountered.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to *name*.

`Getnetbyname()` is one of the Berkeley Socket utilities used to manipulate the `/etc/networks` file. The following utilities also manipulate the `/etc/networks` file.

- `endnetent()`—closes the `/etc/networks` file.
- `getnetbyaddr()`—sequentially searches from the beginning of the `/etc/networks` file until a network number matches the specified parameter or until EOF is encountered.

- `getnetent()`—returns the next line of the entry from the `/etc/networks` file, opening the file if necessary.
- `setnetent()`—opens and rewinds the `/etc/networks` file.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/networks` file.

Error Returns

`Getnetbyname()` returns a null pointer (0) on EOF or when it is unable to open the `/etc/networks` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

Reads the next line of the `/etc/networks` file and returns the network information.

Syntax

```
network = getnetent()

struct netent *network;
```

Parameters

network Pointer to a `netent` structure that contains network information returned by `getnetent()`.

The `netent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct    netent {
    char         *n_name;       /* official name of net */
    char        **n_aliases ;   /* alias list */
    int           n_addrtype;   /* net address type */
    unsigned long n_net;        /* network # */
};
```

Discussion

`Getnetent()`, `getnetbyaddr()`, and `getnetbyname()` each return a pointer to a structure of type `netent`, which contains the broken-out fields of a line in the network database file, `/etc/networks`.

`Getnetent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/networks` file. The following utilities also manipulate the `/etc/networks` file.

- `endnetent()`—closes the `/etc/networks` file.
- `getnetbyaddr()`—sequentially searches from the beginning of the `/etc/networks` file until a network number matches the specified parameter or until EOF is encountered.
- `getnetbyname()`—sequentially searches from the beginning of the `/etc/networks` file until a network name or alias matches the specified parameter or until EOF is encountered.
- `setnetent()`—opens and rewinds the `/etc/networks` file.

Refer to Appendix B, "Database and Header Files," for detailed information on the `/etc/networks` file.

Error Returns

`Getnetent()` returns a null pointer (`0`) on EOF or when it is unable to open the `/etc/networks` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getpeername()

Returns the socket address of the peer socket connected to the specified local socket.

Syntax

```
    result = getpeername(socket, addr, addrlen)

    int                result, socket, *addrlen;
    struct sockaddr_in *addr;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
addr	Pointer to an address structure that contains the socket address of the peer socket that is connected to *socket*. The address structure should be of `sockaddr_in` type, which is described in "Preparing Socket Address Variables" in Section 3.
addrlen	Pointer to an integer variable that contains the length, in bytes, of the address structure specified by *addr* (for example, length of structure `sockaddr_in`, which is 16 bytes). On return, pointer to an integer variable that contains the actual length of the peer socket address. If *addr* does not point to enough space to contain the whole socket address of the peer socket, only the first *addrlen* bytes of the address are filled in the structure pointed to by *addr*.

Discussion

Pascal and FORTRAN programs need to use the `AddressOf()` function to get the pointer to *addr* and *addrlen*.

Error Returns

If `getpeername()` is successful, `0` is returned. If the call fails, `-1` is returned and *errno* contains the cause of the failure. The following table lists possible error returns from the `getpeername()` call.

Error Mnemonic	Meaning
[ENOTSOCK]	`Socket` is not a valid socket descriptor.
[ENOTCONN]	`Socket` is not connected to a peer socket.
[EHOSTDOWN]	The network software on the local host is not running.

getprotobyname()

Returns protocol information on the specified protocol name.

Syntax

```
protocol = getprotobyname(name)

struct protoent   *protocol;
char              *name;
```

Parameters

protocol	Pointer to a `protoent` structure that contains the protocol information returned by `getprotobyname()`.

The `protoent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct      protoent  {
    char       *p_name;       /* official protocol name */
    char      **p_aliases;    /* alias list */
    int         p_proto;      /* protocol # */
};
```

name	Pointer to string that contains the protocol name from which to get protocol information. It can be either an official protocol name or an alias. Terminate the string with the character `\0`.

Discussion

`Getprotoent()`, `getprotobynumber()`, and `getprotobyname()` each return a pointer to a structure of type `protoent`, which contains the broken-out fields of a line in the network protocol database file, `/etc/protocols.`

`Getprotobyname()` sequentially searches from the beginning of the `/etc/protocols` file until a matching protocol name or alias is found, or until EOF is encountered.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to *name*.

`Getprotobyname()` is one of the Berkeley Socket utilities used to manipulate the `/etc/protocols` file. The following utilities also manipulate the `/etc/protocols` file.

- `endprotoent()`—closes the `/etc/protocols` file.
- `getprotobynumber()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol number matches the specified parameter or until EOF is encountered.

- `getprotoent()`—returns the next line of the entry from the `/etc/protocols` file, opening the file if necessary.
- `setprotoent()`—opens and rewinds the `/etc/protocols` file.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/protocols` file.

Error Returns

`Getprotobyname()` returns a null pointer (0) on EOF or when it is unable to open `/etc/protocols` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getprotobynumber()

Returns protocol information on the specified protocol number.

Syntax

```
protocol = getprotobynumber(protonumb)

struct protoent   *protocol;
int                protonumb;
```

Parameters

protocol Pointer to a `protoent` structure that contains protocol information returned by `getprotobynumber()`.

The `protoent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct      protoent  {
    char       *p_name;       /* official protocol name */
    char      **p_aliases;    /* alias list */
    int         p_proto;      /* protocol # */
};
```

protonumb Protocol number from which to get protocol information.

Discussion

`Getprotoent()`, `getprotobynumber()`, and `getprotobyname()` each return a pointer to a structure of type `protoent`, which contains the broken-out fields of a line in the network protocol database file, `/etc/protocols`.

`Getprotobynumber()` sequentially searches from the beginning of file `/etc/protocols` until a matching protocol number is found, or until EOF is encountered.

`Getprotobynumber()` is one of the Berkeley Socket utilities used to manipulate the `/etc/protocols` file. The following utilities also manipulate the `/etc/protocols` file.

- `endprotoent()`—closes the `/etc/protocols` file.
- `getprotobyname()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol name or alias matches the specified parameter or until EOF is encountered.
- `getprotoent()`—returns the next line of the entry from the `/etc/protocols` file, opening the file if necessary.
- `setprotoent()`—opens and rewinds the `/etc/protocols` file.

Refer to Appendix B, "Database and Header Files," for detailed information on the `/etc/protocols` file.

Error Returns

`Getprotobynumber()` returns a null pointer (0) on EOF or when it is unable to open `/etc/protocols` file.

Releasing Memory

Refer to "Releasing Dynamically Allocated Memory" earlier in this section for information on releasing memory allocated by this call.

getprotoent()

Reads the next line of the `/etc/protocols` file and returns the protocol information.

Syntax

```
protocol = getprotoent()

struct protoent *protocol;
```

Parameters

protocol	Pointer to a `protoent` structure that contains protocol information returned by `getprotoent()`. The `protoent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct      protoent  {
    char       *p_name;        /* official protocol name */
    char      **p_aliases;     /* alias list */
    int          p_proto;      /* protocol # */
};
```

Discussion

`Getprotoent()`, `getprotobynumber()`, and `getprotobyname()` each return a pointer to a structure of type `protoent`, which contains the broken-out fields of a line in the network protocol database file, `/etc/protocols`.

`Getprotoent()` reads the next line of the `/etc/protocols` file, opening the file if necessary.

`Getprotoent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/protocols` file. The following utilities also manipulate the `/etc/protocols` file.

- `endprotoent()`—closes the `/etc/protocols` file.
- `getprotobyname()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol name or alias matches the specified parameter or until EOF is encountered.
- `getprotobynumber()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol number matches the specified parameter or until EOF is encountered.
- `setprotoent()`—opens and rewinds the `/etc/protocols` file.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/protocols` file.

Error Returns

`Getprotoent()` returns a null pointer (0) on EOF or when it is unable to open the `/etc/protocols` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getservbyname()

Returns service information on the specified service name.

Syntax

```
service = getservbyname(name, proto)

struct servent   *service;
char             *name, *proto;
```

Parameters

service	Pointer to a `servent` structure that contains service information returned by `getservbyname()`. The `servent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct servent {
   char   *s_name;     /* official service name */
   char  **s_aliases;  /* alias list */
   int     s_port;     /* port #, network byte order */
   char   *s_proto;    /* protocol to use */
}
```

name	Pointer to string that contains the service name from which to get information on the service. It can be either an official service name or an alias. Terminate the string with the character `\0`.
proto	Pointer to string that contains the name of the transport protocol to use when contacting the service. Use “`tcp`” or `0` if TCP is the only protocol for the service. (Remember to terminate the string with the character `\0`.)

Discussion

`Getservent()`, `getservbyname()`, and `getservbyport()` each return a pointer to a structure of type `servent`, which contains the broken-out fields of a line in the network services database file, `/etc/services`.

`Getservbyname()` sequentially searches from the beginning of the `/etc/services` file until a matching service name or alias is found, or until EOF is encountered. If a non-NULL protocol name is also supplied in *proto* (for example, “tcp”), the search must also match the specified protocol name.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointers to *name* and *proto*.

`Getservbyname()` is one of the Berkeley Socket utilities used to manipulate the `/etc/services` file. The following utilities also manipulate the `/etc/services` file.

- `endservent()`—closes the `/etc/services` file.
- `getservbyport()`—sequentially searches from the beginning of the `/etc/services` file until a port number matches the specified parameter or until EOF is encountered.
- `getservent()`—returns the next line of the entry from the `/etc/services` file, opening the file if necessary.
- `setservent()`—opens and rewinds the `/etc/services` file.

Refer to Appendix B, "Database and Header Files," for detailed information on the `/etc/services` file.

Error Returns

`Getservbyname()` returns a null pointer (0) on EOF or when it is unable to open `/etc/services` file.

Releasing Memory

Refer to "Releasing Dynamically Allocated Memory" earlier in this section for information on releasing memory allocated by this call.

getservbyport()

Returns service information on the specified port number.

Syntax

```
service = getservbyport(port, proto)

struct servent   *service;
int               port;
char             *proto;
```

Parameters

service	Pointer to a `servent` structure that contains service information returned by `getservbyname()`.

The `servent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct servent {
   char   *s_name;     /* official service name */
   char  **s_aliases;  /* alias list */
   int     s_port;     /* port #, network byte order */
   char   *s_proto;    /* protocol to use */
}
```

port	Port number from which to get information on the service.
proto	Pointer to string that contains the name of the transport protocol to use when contacting the service. Use "`tcp`" or `0` if TCP is the only protocol for the service. Terminate string with character `\0`.

Set this value to NULL if you do not want to specify any specific protocol.

Discussion

`Getservent()`, `getservbyname()`, and `getservbyport()` each return a pointer to a structure of type `servent`, which contains the broken-out fields of a line in the network services database file, `/etc/services`.

`Getservbyport()` sequentially searches from the beginning of the `/etc/services` file until a matching port number is found, or until EOF is encountered. If a non-NULL protocol name is also supplied in *`proto`* (for example, "tcp"), the search must also match the specified protocol name.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to *`proto`*.

`Getservbyport()` is one of the Berkeley Socket utilities used to manipulate the `/etc/services` file. The following utilities also manipulate the `/etc/services` file.

- `endservent()`—closes the `/etc/services` file.
- `getservbyname()`—sequentially searches from the beginning of the `/etc/services` file until a service name or alias matches the specified parameter or until EOF is encountered.
- `getservent()`—returns the next line of the entry from the `/etc/services` file, opening the file if necessary.
- `setservent()`—opens and rewinds the `/etc/services` file.

Refer to Appendix B, “Database and Header Files,” for detailed information on the `/etc/services` file.

Error Returns

`Getservbyport()` returns a null pointer (0) on EOF or when it is unable to open `/etc/services` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getservent()

Reads the next line of the `/etc/services` file and returns information on the service.

Syntax

```
service = getservent()

struct servent  *service;
```

Parameters

service	Pointer to a `servent` structure that contains service information returned by `getservent()`.

The `servent` structure is defined in the include files `<netdb.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. It is shown below in C programming format.

```
struct servent {
   char   *s_name;     /* official service name */
   char  **s_aliases;  /* alias list */
   int     s_port;     /* port #, network byte order */
   char   *s_proto;    /* protocol to use */
}
```

Discussion

`Getservent()`, `getservbyname()`, and `getservbyport()` each return a pointer to a structure of type `servent`, which contains the broken-out fields of a line in the network services database file, `/etc/services`.

`Getservent()` reads the next line of the `/etc/services` file, opening the file if necessary.

`Getservent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/services` file. The following utilities also manipulate the `/etc/services` file.

- `endservent()`—closes the `/etc/services` file.
- `getservbyname()`—sequentially searches from the beginning of the `/etc/services` file until a service name or alias matches the specified parameter or until EOF is encountered.
- `getservbyport()`—sequentially searches from the beginning of the `/etc/services` file until a port number matches the specified parameter or until EOF is encountered.
- `setservent()`—opens and rewinds the `/etc/services` file.

Refer to Appendix B, "Database and Header Files," for detailed information on the `/etc/services` file.

Error Returns

`Getservent()` returns a null pointer (0) on EOF or when it is unable to open `/etc/services` file.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

getsockname()

Returns the socket address of the specified local socket.

Syntax

```
result = getsockname (socket, addr, addrlen)

int                  result, socket, *addrlen;
struct sockaddr_in   *addr;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
socket	Socket descriptor of a local socket.
addr	Pointer to a socket address variable to contain the address of the specified socket. The socket address should be of `sockaddr_in` type, which is described in "Preparing Socket Address Variables" in Section 3. On return, the socket address structure will contain the local socket address information.
addrlen	Pointer to an integer variable that contains the length, in bytes, of the address structure specified by *addr* (for example, length of structure `sockaddr_in`, which is 16 bytes). On return, it is the pointer to an integer that contains the actual length of the socket address returned in *addr*. If *addr* does not point to enough space to contain the whole address of the socket, only the first *addrlen* bytes of the address are returned.

`Getsockname()` is used to find the socket address of a local socket. To obtain the socket address of the peer socket, use `getpeername()`.

Sometimes in a client process, the local socket's address is assigned randomly by the `connect()` call, instead of bound explicitly by `bind()`. In this case, `getsockname()` can be used to find the socket address of the local socket.

Pascal and FORTRAN programs need to use the `AddressOf()` function to get the pointers to *addr* and *addrlen*.

Error Returns

If the call is successful, `0` is returned. If the call failed, `-1` is returned and the error code is stored in *errno*. The following table lists possible error returns from `getsockname()`.

Error Mnemonic	Meaning
[ENOTSOCK]	*Socket* is not a valid socket descriptor.
[EHOSTDOWN]	The network software on the local host is not running.

Converts a 32-bit quantity from host order to network order.

Syntax

```
netlong = htonl(hostlong)

u_long netlong, hostlong;
```

Parameters

netlong	32-bit integer in network order, returned by `htonl()`.
hostlong	32-bit integer in host order.

Discussion

There are four routines to convert an integer from network order to host order and vice versa.

- `htonl()`—converts a 32-bit integer from host order to network order.
- `htons()`—converts a 16-bit integer from host order to network order.
- `ntohl()`—converts a 32-bit integer from network order to host order.
- `ntohs()`—converts a 16-bit integer from network order to host order.

htons()

Converts a 16-bit quantity from host order to network order.

Syntax

```
netshort = htons(hostshort)

u_short netshort, hostshort;
```

Parameters

netshort	16-bit integer in network order, returned by `htons()`.
hostshort	16-bit integer in host order.

Discussion

There are four routines to convert an integer from network order to host order and vice versa.

- `htonl()`—converts a 32-bit integer from host order to network order.
- `htons()`—converts a 16-bit integer from host order to network order.
- `ntohl()`—converts a 32-bit integer from network order to host order.
- `ntohs()`—converts a 16-bit integer from network order to host order.

Interprets character strings representing numbers in the Internet standard “dot” notation, and returns numbers suitable for use as Internet (IP) addresses.

Syntax

```
IPaddr = inet_addr(string)

struct in_addr IPaddr;
char           *string;
```

Parameters

IPaddr	Internet (IP) address returned by `inet_addr()`.
string	Pointer to a character string representing numbers expressed in the Internet standard “dot” notation, such as: “192.41.233.2”. Terminate the string with the character `\0`.

Discussion

The routine `inet_addr()` converts character strings to 32-bit Internet (IP) addresses. The routine `inet_ntoa()` does the reverse conversion; it converts a 32-bit IP address to an ASCII string in “dot” notation.

The return value of `inet_addr()`, *IPaddr*, may be assigned to an address structure (or record) of type `in_addr`, which is an address variable used to store IP addresses for the Internet family. Refer to “Preparing Socket Addresses” in Section 3 for more information on the `in_addr` address structure.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to *string*.

Refer to “IP Address” in Appendix D for more information on the IP address and its formats.

Note All IP addresses are returned in network order (bytes ordered from left to right). All network numbers and local node address portions are returned as machine format integer values. Bytes in HP-UX systems are ordered from left to right.

Error Returns

The value `-1` is returned by `inet_addr()` for malformed requests.

inet_lnaof()

Breaks apart an IP address and returns the node address portion of the IP address.

Syntax

```
node = inet_lnaof(IPaddr)

u_long          node;
struct in_addr  IPaddr;
```

Parameters

node	Node address portion of the IP address returned by `inet_lnaof()`.
IPaddr	IP address of a host. IP addresses for the Internet family are stored in an address variable of type `in_addr`. Refer to “Preparing Socket Addresses” in Section 3 for more information about `in_addr`.

Discussion

An IP address consists of two parts: the network address and the node address of a host. The routine `inet_lnaof()` breaks apart an IP address and returns the *node address* portion of the IP address. The routine `inet_netof()` breaks apart an IP address and returns the *network address* portion of the IP address.

The network portion of the IP address returned is based on the Class A, B, C categorization of IP addresses and does not include subnet masks.

Refer to “IP Address” in Appendix D for more information on the IP address and its components.

Note All IP addresses are returned in network order (bytes ordered from left to right). All network numbers and local node address portions are returned as machine format integer values. Bytes in HP-UX systems are ordered from left to right.

Constructs an Internet (IP) address from an Internet network address and a local node address.

Syntax

```
IPaddr = inet_makeaddr(net, node)

struct in_addr   IPaddr;
u_long           net, node;
```

Parameters

IPaddr	Internet (IP) address constructed from the specified network address and node address. IP addresses for the Internet family are stored in an address variable of type `in_addr`. Refer to “Preparing Socket Addresses” in Section 3 for more information on `in_addr`.
net	Internet network number that defines the network on which a node resides. The network number makes up a portion of an IP address.
node	Internet node address that defines the address of a node within a network. The node address makes up a portion of an IP address.

Discussion

An IP address consists of two parts: the network address and the node address of a host. The routine `inet_makeaddr()` takes a network address and a node address and constructs an IP address.

Refer to “IP Address” in Appendix D for more information on the IP address and its components.

Note All IP addresses are returned in network order (bytes ordered from left to right). All network numbers and local node address portions are returned as machine format integer values. Bytes in HP-UX systems are ordered from left to right.

inet_netof()

Breaks apart an IP address and returns the network address portion of the IP address.

Syntax

```
network = inet_netof(IPaddr)

u_long          network;
struct in_addr  IPaddr;
```

Parameters

network	Network address portion of the IP address returned by `inet_netof()`.
IPaddr	IP address of the local host. IP addresses for the Internet family are stored in an address variable of type `in_addr`. Refer to "Preparing Socket Address Variables" in Section 3 for more information about `in_addr`.

Discussion

An IP address consists of two parts: the network address and the node address of a host. The routine `inet_netof()` breaks apart an IP address and returns the *network address* portion of the IP address. The routine `inet_lnaof()` breaks apart an IP address and returns the *node address* portion of the IP address.

Refer to "IP Address" in Appendix D for more information on the IP address and its components.

Note All IP addresses are returned in network order (bytes ordered from left to right). All network numbers and local node address portions are returned as machine format integer values. Bytes in HP-UX systems are ordered from left to right.

inet_network()

Interprets character strings representing numbers in the Internet standard “dot” notation, and returns numbers suitable for use as Internet network numbers.

Syntax

```
network = inet_network(string)

struct in_addr  network;
char            *string;
```

Parameters

network	Internet network number returned by `inet_network()`.
string	Pointer to character string representing numbers expressed in the Internet standard “dot” notation, such as: “192.41.233.2”. Terminate the string with the character `\0`. See “Internet Dot Notation” in Appendix D for more information on dot notation.

Discussion

The routine `inet_network()` converts character strings to Internet network numbers, suitable for use as part of an Internet (IP) address. The routine `inet_addr()` converts character strings to 32-bit whole IP addresses.

Pascal and FORTRAN programs need to use the `ByteAdrOf()` function to get the character pointer to *string*.

Refer to “IP Address” in Appendix D for more information on the IP address and its formats.

Note All IP addresses are returned in network order (bytes ordered from left to right). All network numbers and local node address portions are returned as machine format integer values. Bytes in HP-UX systems are ordered from left to right.

Error Returns

The value `-1` is returned by `inet_addr()` for malformed requests.

inet_ntoa()

Takes an Internet (IP) address and returns an ASCII string representing the address in “dot” notation.

Syntax

```
string = inet_ntoa(IPaddr)

char             *string;
struct in_addr   IPaddr;
```

Parameters

string	Returns pointer to character string (byte address) representing numbers expressed in the Internet standard “dot” notation, such as: “192.41.233.2”. The string is terminated with character `\0`. See “Internet Dot Notation” in Appendix D for more information on dot notation.
IPaddr	Internet (IP) address. IP addresses for the Internet family are stored in an address variable of type `in_addr`. Refer to “Preparing Socket Addresses” in Section 3 for more information about `in_addr`.

Discussion

The routine `inet_ntoa()` converts IP addresses to ASCII strings representing IP addresses in “dot” notation. The routine `inet_addr()` does the reverse conversion; it converts an ASCII string representing IP address in “dot” notation to an IP address.

Pascal and FORTRAN programs cannot directly access the string built by `inet_ntoa()`.

Refer to “IP Address” in Appendix D for more information on IP addresses, and “Internet Dot Notation” for more information on Internet dot notation.

Error Returns

The value `-1` is returned by `inet_addr()` for malformed requests.

Releasing Memory

Refer to “Releasing Dynamically Allocated Memory” earlier in this section for information on releasing memory allocated by this call.

Converts a 32-bit quantity from network order to host order.

Syntax

```
hostlong = ntohl(netlong)

u_long hostlong, netlong;
```

Parameters

hostlong	32-bit integer in host order, returned by `ntohl()`.
netlong	32-bit integer in network order.

Discussion

There are four routines to convert an integer from network order to host order and vice versa.

- `htonl()`—converts a 32-bit integer from host order to network order.
- `htons()`—converts a 16-bit integer from host order to network order.
- `ntohl()`—converts a 32-bit integer from network order to host order.
- `ntohs()`—converts a 16-bit integer from network order to host order.

ntohs()

Converts a 16-bit quantity from network order to host order.

Syntax

```
hostshort = ntohs(netshort)

u_short hostshort, netshort;
```

Parameters

hostshort	16-bit integer in host order, returned by `ntohs()`.
netshort	16-bit integer in network order.

Discussion

There are four routines to convert an integer from network order to host order and vice versa.

- `htonl()`—converts a 32-bit integer from host order to network order.
- `htons()`—converts a 16-bit integer from host order to network order.
- `ntohl()`—converts a 32-bit integer from network order to host order.
- `ntohs()`—converts a 16-bit integer from network order to host order.

sethostent()

Opens and rewinds the `/etc/hosts` file.

Syntax

```
    result = sethostent (stayopen)

    int  result, stayopen;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
stayopen	A zero value closes the `/etc/hosts` file after each call to the file by one of the following calls: `gethostbyaddr()`, `gethostbyname`, and `gethostent()`. A non-zero value leaves the `/etc/services` file open after a `gethostbyaddr()`, `gethostbyname()`, or `gethostent()` call. This allows the next `gethostent()` to read from the next line of the `/etc/hosts` file rather than from the beginning of the file.

Discussion

`Sethostent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/hosts` file. The following utilities manipulate the `/etc/hosts` file. The `/etc/hosts` file is used by the following BSD IPC utilities:

- `endhostent()`—closes the `/etc/hosts` file.
- `gethostbyaddr()`—returns host information from the specified IP address.
- `gethostbyname`—returns host information from the specified host name.
- `gethostent()`—reads the next line of the `/etc/hosts` file and returns host information on that host.
- `sethostent()`—opens and rewinds the `/etc/hosts` file.

Error Returns

`Sethostent()` returns `-1` when it is unable to open the `/etc/hosts` file.

setnetent()

Opens and rewinds the `/etc/networks` file.

Syntax

```
    result = setnetent (stayopen)

    int   result, stayopen;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
stayopen	A zero value closes the `/etc/networks` file after each call to the file by one of the following calls: `getnetbyaddr()`, `getnetbyname()`, and `getnetent()`. A non-zero value leaves the `/etc/services` file open after a `getnetbyaddr()`, `getnetbyname()`, or `getnetent()` call. This allows the next `getnetent()` to read from the next line of the `/etc/networks` file rather than from the beginning of the file.

Discussion

`Setnetent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/networks` file. The following utilities also manipulate the `/etc/networks` file.

- `endnetent()`—closes the `/etc/networks` file.
- `getnetbyaddr()`—sequentially searches from the beginning of the `/etc/networks` file until a network number matches the specified parameter or until EOF is encountered.
- `getnetbyname()`—sequentially searches from the beginning of the `/etc/networks` file until a network name or alias matches the specified parameter or until EOF is encountered.
- `getnetent()`—returns the next line of the entry from the `/etc/networks` file, opening the file if necessary.

Error Returns

`Setnetent()` returns a `-1` when it is unable to open the `/etc/networks` file.

Opens and rewinds the `/etc/protocols` file.

Syntax

```
    result = setprotoent (stayopen)

    int  result, stayopen;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
stayopen	A zero value closes the `/etc/protocols` file after each call to the file by one of the following calls: `getprotobyname()`, `getprotobynumber()`, and `getprotoent()`. A non-zero value leaves the `/etc/protocols` file open after a `getprotobyname()`, `getprotobynumber()`, or `getprotoent()` call. This allows the next `getprotoent()` to read from the next line of the `/etc/protocols` file rather than from the beginning of the file.

Discussion

`Setprotoent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/protocols` file. The following utilities also manipulate the `/etc/protocols` file.

- `endprotoent()`—closes the `/etc/protocols` file.
- `getprotobyname()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol name or alias matches the specified parameter or until EOF is encountered.
- `getprotobynumber()`—sequentially searches from the beginning of the `/etc/protocols` file until a protocol number matches the specified parameter or until EOF is encountered.
- `getprotoent()`—returns the next line of the entry from the `/etc/protocols` file, opening the file if necessary.

Error Returns

`Setprotoent()` returns a `-1` when it is unable to open the `/etc/protocols` file.

setservent()

Opens and rewinds the `/etc/services` file.

Syntax

```
result = setservent (stayopen)

int  result, stayopen;
```

Parameters

result	`0` if the call is successful. `-1` if a failure occurs.
stayopen	A zero value closes the `/etc/services` file after each call to the file by one of the following calls: `getservbyname()`, `getservbyport()`, and `getservent()`. A non-zero value leaves the `/etc/services` file open after a `getservbyname()`, `getservbyport()`, or `getservent()` call. This allows the next `getservent()` to read from the next line of the `/etc/services` file rather than from the beginning of the file.

Discussion

`Setservent()` is one of the Berkeley Socket utilities used to manipulate the `/etc/services` file. The following utilities also manipulate the `/etc/services` file.

- `endservent()`—closes the `/etc/services` file.
- `getservbyname()`—sequentially searches from the beginning of the `/etc/services` file until a service name or alias matches the specified parameter or until EOF is encountered.
- `getservbyport()`—sequentially searches from the beginning of the `/etc/services` file until a port number matches the specified parameter or until EOF is encountered.
- `getservent()`—returns the next line of the entry from the `/etc/services` file, opening the file if necessary.

Error Returns

`Setservent()` returns a `-1` when it is unable to open the `/etc/services` file.

6

HP 1000 Socket Descriptor Utilities

This section provides reference information on socket descriptor utilities, which operate on socket descriptor bitmasks. Socket descriptor bitmasks are used by the `select()` call to specify which sockets are ready for reading, writing, or have exceptional conditions pending.

The socket descriptor utilities are provided as procedures or functions for Pascal and FORTRAN users and as macros in the include files for C users.

Table 6-1 lists the utilities covered in this section.

Table 6-1. Berkeley Socket Descriptor Utilities

BSD IPC Utilities	Description
`FD_CLR()`	Clears the socket descriptor bit in the bitmask.
`FD_ISSET()`	Test whether the socket descriptor bit is set in the bitmask.
`FD_SET()`	Sets the socket descriptor bit in the bitmask.
`FD_ZERO()`	Clears the entire bitmask.

Refer to the `select()` call in Section 4 for more information on how the socket descriptor bitmasks and the socket descriptor utilities are used for synchronous I/O multiplexing.

Note The bitmasks are stored in a special data type defined as `fd_set`. Refer to the header files for C (`types.h`), Pascal (`SOCKET.PASI`), and FORTRAN (`SOCKET.FTNI`) in Appendix B, “Database and Header Files,” to see the structure of data type `fd_set`.

FD_CLR()

Clears the specified socket descriptor's bit in the bitmask.

Syntax

```
FD_CLR  (socket, bitmask)

int      socket;
fd_set   *bitmask;
```

Parameters

socket	Socket descriptor of a local socket.
bitmask	Pointer to a variable of `fd_set` type which contains the bitmask of the socket descriptors.

Discussion

`FD_CLR()` clears only the bit corresponding to the socket descriptor, *socket*, in the specified *bitmask*.

HP 1000 Specific Information

- The `FD_CLR()` call is provided as a preprocessor macro for C users and as a procedure call for Pascal and FORTRAN users.
- Since there is no return value, users will have to be careful about passing a valid socket descriptor to this routine. Any socket descriptor, *socket*, outside the range of 0 to 30 will be silently rounded to the closest value in the range.
- Pascal and FORTRAN users need to use the `AddressOf()` function to get the pointer to the bitmask.

Tests whether the specified socket descriptor's bit is set in the specified bitmask.

Syntax

```
result = FD_ISSET (socket, bitmask)

int      result, socket;
fd_set   *bitmask;
```

Parameters

result	Result of `FD_ISSET()` call. *result* = `1` means the bit is set for the specified socket descriptor, *socket*. Otherwise, *result* = `0`.
socket	Socket descriptor of a local socket.
bitmask	Pointer to a `fd_set` variable type which contains the bitmask of the socket descriptors.

Discussion

`FD_ISSET()` tests whether the bit corresponding to the socket descriptor *socket* is set in the specified *bitmask*. If set, `FD_ISSET()` returns the value `1` (boolean TRUE for Pascal), otherwise, it returns `0` (boolean FALSE for Pascal).

HP 1000 Specific Information

- This call is provided as a preprocessor macro for C users and as a function call for Pascal and FORTRAN users.
- Users must be careful about passing a valid socket descriptor to this routine. Any socket descriptor outside the range of 0 to 30 will be silently rounded to the closest value in the range.
- Pascal users should declare this function as returning a boolean value.
- Pascal and FORTRAN users need to use the `AddressOf()` function to get the pointer to the bitmask.

FD_SET()

Sets the specified socket descriptor's bit in the bitmask.

Syntax

```
FD_SET (socket, bitmask)

int      socket;
fd_set   *bitmask;
```

Parameters

socket	Socket descriptor of a local socket.
bitmask	Pointer to a variable of type `fd_set` which contains the bitmask of the socket descriptors.

Discussion

`FD_SET()` sets the bit corresponding to the socket descriptor, *socket*, in the specified *bitmask*. This bitmask can then be used in a `select()` call. `Select()` calls are used to provide synchronous socket I/O multiplexing. Refer to the `select()` call in Section 4 for more information.

Note that `FD_SET()` does not clear the bitmask before setting the bit corresponding to the socket descriptor, *socket*.

HP 1000 Specific Information

- The `FD_SET()` call is provided as a preprocessor macro for C users and as a procedure call for Pascal and FORTRAN users.
- Since there is no return value, users will have to be careful about passing a valid socket descriptor to this routine. Any socket descriptor, *socket*, outside the range of 0 to 30 will be silently rounded to the closest value in the range.
- Pascal and FORTRAN users need to use the `AddressOf()` function to get the pointer to the bitmask.

Clears the entire bitmask.

Syntax

```
    FD_ZERO (bitmask)

    fd_set  *bitmask;
```

Parameters

bitmask	Pointer to a variable of type `fd_set` which contains the bitmask of the socket descriptors.

Discussion

`FD_ZERO()` clears all the bits in the specified *bitmask*. It is recommended that you clear the entire bitmask before using it, to ensure that the right bits are set when the bitmask is used.

To clear specific bits in a bitmask, use `FD_CLR()`.

HP 1000 Specific Information

- The `FD_ZERO()` call is provided as a preprocessor macro for C users and as a procedure call for Pascal and FORTRAN users.
- Since there is no return value, users will have to be careful about passing a valid socket descriptor to this routine. Any socket descriptor, *socket*, outside the range of 0 to 30 will be silently rounded to the closest value in the range.
- Pascal and FORTRAN users need to use the `AddressOf()` function to get the pointer to the bitmask.

7

Advanced Topics

This section covers advanced topics for programming with BSD IPC on HP 1000. This section explains the following:

- Setting and Getting Socket Options.
- Nonblocking I/O.

Setting and Getting Socket Options

The operation of sockets is controlled by socket level options. The socket options are defined in the include files `<socket.h>`, `SOCKET.PASI`, and `SOCKET.FTNI`, for C, Pascal, and FORTRAN programs, respectively. (These files are shown in Appendix B, "Database and Header Files.")

You can get the current status of an option with the `getsockopt()` call, and you can set the value of an option with the `setsockopt()` call. The following socket options are currently supported on HP 1000 BSD IPC:

`SO_KEEPALIVE`	Sets a timer for 90 minutes for connected sockets. After 90 minutes expires and if the connection has been idle for this period, `SO_KEEPALIVE` forces a transmission every 60 seconds, for up to 7 minutes, after which the idle connection is shutdown. If this option is toggled off, an indefinite idle time is allowed. This option is set by default.
`SO_RCVBUF`	Changes the buffer size of a socket's receive socket buffer. The default buffer size is 4096 bytes. The buffer size can only be changed before connection is established.
`SO_REUSEADDR`	Allows local address reuse. This allows multiple sockets to be bound to the same local port number.
`SO_SNDBUF`	Changes the buffer size of a socket's send socket buffer. The default buffer size is 4096 bytes. The buffer size can only be changed before connection is established.

The `getsockopt()` and `setsockopt()` calls are covered in Section 4, "BSD IPC Calls."

Note The `SO_LINGER` option (available on HP-UX) is not provided on the HP 1000.

NonBlocking I/O

Sockets are created in blocking mode by default. You can specify that a socket be put in nonblocking mode by using the `fcntl()` call with the *`status`* flag set to `O_NONBLOCK`. (See `fcntl()` call in Section 4 for more information.)

If a socket is in nonblocking mode, the following calls are affected:

`accept()`

`Accept()` is used by a listening server process to accept a connection request from a client process. Normally, in blocking mode, `accept()` blocks until there is a connection request from a client process.

If you are in nonblocking mode and no pending connections are present on the queue, `accept()` returns `-1` in *`newsocket`* and *`errno`* contains an `EAGAIN` error.

It is possible to determine if a listening socket has pending connection requests ready for an `accept()` call by using `select()` for reading.

`connect()`

The `connect()` call is used by a client process to initiate a connection request to the server process. The `connect()` call normally blocks until the connection completes.

In nonblocking mode, if the connection cannot be completed immediately, it returns an `EINPROGRESS` error in *`errno`*. In this case, the `select()` call can be used to determine if the connection has completed by selecting it for write.

`recv()`

Normally, in blocking mode, if no data is available to be received, `recv()` blocks and waits for data to arrive.

If nonblocking I/O is enabled, the `recv()` request will complete in one of three ways:

1. If there is enough data to satisfy the entire request, `recv()` will complete successfully, having read the entire data in the buffer.
2. If there is not enough data available to satisfy the entire request, `recv()` will complete successfully, having read as much data as possible and returns the number of bytes it was able to read.
3. If there is no data available, `recv()` will return `-1`, with *`errno`* set to `EAGAIN`.

By selecting the socket for read indication, the `select()` call may be used to determine when a socket has data available to be read by a `recv()`.

send()

Normally, in blocking mode, `send()` blocks until the specified number of bytes have been queued to be sent.

If nonblocking mode is used, `send()` will complete in one of three ways:

1. If there is enough space available in the system to buffer all the data, `send()` will complete successfully, having written out all of the data, and return the number of bytes written.

2. If there is not enough space in the buffer to write out the entire request, `send()` will complete successfully, having written as much data as possible, and return the number of bytes it was able to write.

3. If there is no space in the system to buffer any of the data, `send()` will return `-1`, having written no data, with *errno* set to `EAGAIN`.

By selecting the socket for write indication, the `select()` call may be used to determine when a socket has data available to be sent by a subsequent `send()` call.

A

Example Programs

This appendix contains example BSD IPC programs. It provides example server programs and client programs written in C, Pascal, and FORTRAN. Example load files are also included.

Example Server Program in C

```
/* BSDSERVER.C 91790-18296 REV.6200 <940914.1450>
 *
 *     NAME   : BSDSERVER.C
 *     SOURCE : 91790-18296
 *
 * This is a server example program for BSD sockets.
 * See the program ”bsdclient.c” for the client example.
 * This program creates a socket, binds it to a well known port,
 * waits for a connection from the client, receives and echoes data
 * over the socket connection to the client.
 *
 * To link this program, see the attached ”bsdserver.lod” file.
 * To run this program, use the following command:
 *           bsdserver
 */
#include <types.h>
#include <socket.h>
#include <in.h>
#include <stdio.h>
#include <time.h>
#include <fcntl.h>
#include <errno.h>


#define  SERVER_PORT  20000    /* server port address */
#define  BUFLEN       7500     /* max bytes received */
#define  SETSOCKLEN   7500     /* send, recv buffer size for setsockopt() */
#define  AF_INETLEN   16       /* length of internet address structure */
#define  BACKLOG      3        /* max connections on listen socket */
void cleanup();

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

```
{
    struct sockaddr_in addr;
    int  af, type, proto;
    int  sd, b, l, a1, sl, sss, ssr, r, s;
    int  addrlen;
    int  nfds;
    fd_set readfds, writefds, exceptfds;
    char buf[BUFLEN];
    int opt;
    int optlen;
    int totalr;
    long flags;

    /* Create a socket */
    af     = AF_INET;           /* domain */
    type   = SOCK_STREAM;       /* type of socket */
    proto  = IPPROTO_TCP;       /* protocol */
    sd = socket(af, type, proto);
    if (sd == -1)
    {
        fprintf(stdout,"%s: Socket error %d\n",argv[0],errno);
        exit();
    }

    /* Set send and receive buffer size */
    opt =  SETSOCKLEN;
    optlen = 2;
    sss = setsockopt(sd,SOL_SOCKET,SO_SNDBUF,(char *)(&opt),optlen);
    if (sss == -1)
    {
        fprintf(stdout,"%s: Setsockopt sendbuf error %d\n",argv[0],errno);
        cleanup(sd, -1);
    }

    ssr = setsockopt(sd,SOL_SOCKET,SO_RCVBUF,(char *)(&opt),optlen);
    if (ssr == -1)
    {
        fprintf(stdout,"%s: Setsockopt recvbuf error %d\n",argv[0],errno);
        cleanup(sd, -1);
    }

    /* Bind the socket to a local port address */
    addr.sin_family = af;
    addr.sin_port   = SERVER_PORT;
    addrlen         = AF_INETLEN;
    b = bind(sd,&addr,addrlen);
    if (b == -1)
    {
        fprintf(stdout,"%s: Bind error %d\n", argv[0],errno);
        cleanup(sd, -1);
    }

    /* Set up the socket in listen mode so as to be able */
    /*   to receive connection requests.                 */
    l = listen (sd, BACKLOG);
    if (l == -1)
    {
        fprintf(stdout,"%s: Listen error %d\n",argv[0], errno);
        cleanup(sd, -1);
    }

    /* Accept incoming connection */
    addrlen = sizeof(addr);
    a1 = accept(sd, &addr, &addrlen);
    if (a1 == -1)
    {
        fprintf(stdout,"%s: Accept error %d\n",argv[0],errno);
        cleanup(sd, -1);
    }

    totalr = 0;
    while (totalr < BUFLEN)
    {
        nfds = a1 + 1;
        FD_ZERO(&readfds);
        FD_ZERO(&writefds);
        FD_ZERO(&exceptfds) ;
        FD_SET(a1, &readfds);
        FD_SET(a1, &exceptfds);
        sl = select(nfds,&readfds,&writefds,&exceptfds,(struct timeval *)
NULL);
        if (sl == -1)
        {
            fprintf(stdout,"%s: Select error %d\n",argv[0],errno);
            break;
        }

        /* Check if socket is exceptional selected */
        if (FD_ISSET(a1,&exceptfds))
        {
            fprintf(stdout,"%s: Unexpected exceptional signal\n",argv[0]);
            break;
        }

        /* Check if socket is read selected */
        if (FD_ISSET(a1,&readfds))
        {
            flags = 0;  /* clear all the flags */

            /* Receive data from client */
            if ((r = recv(a1,&buf[totalr],BUFLEN-totalr,flags)) == -1)
            {
                fprintf(stdout,"%s: Recv error %d\n",argv[0],errno);
                break;
            }

            /* Print out the data received */
            buf[r-1] = '\0';
            fprintf(stdout,"Received: len = %d, string = %s\n",r,buf);
            totalr += r;

        }   /* end of if read selected */
    }   /* end of while */

    /* Send data back to the client */
    if ((s = send(a1,&buf[0],totalr,flags)) == -1)
    {
       fprintf(stdout,”%s: Send error %d\n”,argv[0],errno);
    }

    cleanup(sd,a1);

}   /* end of main */

/**************************************************************************/

void cleanup(s1,s2)
int s1,s2;

{
    /* Shut down connected socket */
    if ( s2 != -1 )
        if (shutdown( s2, 0) == -1)
            fprintf(stdout,”%s: Shutdown (a) error\n”,errno);

    /* Shut down listened socket */
    if (shutdown( s1, 0) == -1)
        fprintf(stdout,”%s: Shutdown (sd) error\n”,errno);

    exit();
}
```

Example Client Program in C

```
/* BSDCLIENT.C 91790-18295 REV.6200 <940914.1449>
 *
 *     NAME   : BSDCLIENT.C
 *     SOURCE : 91790-18295
 *
 * This is a client example program for BSD sockets.
 * See the program ”bsdserver.c” for the server example.
 * This program takes the name of a remote system as an argument.
 * The routine gethostbyname() is used to resolve the IP address of the
 * server machine.  A socket is created, binded to a known port.
 * After establishing a connection with the server,  an initialized buffer of
 * BUFLEN bytes is sent to the server which will be echoed back and compared
 * to the input buffer.  If the sent and received buffers are identical,
 * a match message will be printed on the screen, otherwise a mismatch
 * message along with the sent and received buffers will be printed.
 *
 * To link this program, see the attached ”bsdclient.lod” file.
 * To run this program:
 *       - make sure that the remote node exist in the file ”/etc/hosts”,
 *       - start the server program on the remote node,
 *       - start this program by typing in the string: ”bsdclient nodename”.
 *
 */

#include <types.h>
#include <socket.h>
#include <in.h>
#include <stdio.h>
#include <fcntl.h>
#include <time.h>
#include <errno.h>
#include <netdb.h>
#include <string.h>

#define  CLIENT_PORT  20001    /* client port address */
#define  SERVER_PORT  20000    /* server port address */
#define  BUFLEN       7500     /* data buffer size */
#define  SETSOCKLEN   7500     /* send, recv buffer size for setsockopt() */
#define  AF_INETLEN   16       /* length of internet address structure */

void cleanup();

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

{
    struct sockaddr_in addr;
    struct hostent *hostEntry;
    int  addrlen;
    int  af, type, proto;
    int  sd, b, c, sss, ssr, s, r;
    char sbuf[BUFLEN];
    char rbuf[BUFLEN];
    char *rhost;
    char ready;
    int  opt;
    int  optlen;
    int  i, j, totalr;
    long flags;

    /* Remind user to start server process */
    fprintf(stdout, ”Have you started the server program?(y/n): ”);
    fscanf(stdin, ”%c”, &ready);
    if (ready != ’y’)
    {
        fprintf(stdout,”Start server program then restart client\n”);
        exit();
    }

    /* Get remote host name from run string */
    if (argc < 2)
        fprintf(stdout, ”Usage: %s remote_hostname\n”, argv[0]);
    else
        rhost = argv[1];

    /* Use gethostbyname() to look up the remote host in /etc/hosts file */

    if ( !(hostEntry = gethostbyname(rhost)))
    {
        fprintf(stdout,”%s: Gethostbyname error \n”,argv[0]);
        exit();
    }

    /* Create a socket */
    af     = AF_INET;           /* domain */
    type   = SOCK_STREAM;       /* type of socket */
    proto  = IPPROTO_TCP;       /* protocol*/
    sd = socket(af, type, proto);
    if (sd == –1)
    {
        fprintf(stdout,”%s: Socket error %d\n”,argv[0],errno);
        exit();
    }

    /* Set send and receive buffer size */
    opt =  SETSOCKLEN;
    optlen = 2;
    sss = setsockopt(sd,SOL_SOCKET,SO_SNDBUF,(char *)(&opt),optlen);
    if (sss == –1)
    {
        fprintf(stdout,”%s: Setsockopt sendbuf error %d\n”,argv[0],errno);
        cleanup(sd);
    }
```

```
ssr = setsockopt(sd,SOL_SOCKET,SO_RCVBUF,(char *)(&opt),optlen);
if (ssr == -1)
{
    fprintf(stdout,"%s: Setsockopt recvbuf error %d\n",argv[0],errno);
    cleanup(sd);
}

/* Bind the socket to a local port address */
addr.sin_family = af;
addr.sin_port   = CLIENT_PORT;
addrlen         = AF_INETLEN;
b = bind(sd,&addr,addrlen);
if (b == -1)
{
    fprintf(stdout,"%s: Bind error %d\n",argv[0],errno);
    cleanup(sd);
}

/* Set up address structure and connect to remote host */
addr.sin_port = SERVER_PORT;

/* Now insert IP address of remote host from hostent structure. */
/* Note the casting operations done in order to obtain a 32 bit */
/* IP address from a char pointer.                              */
addr.sin_addr.s_addr = * (u_long *) *(hostEntry->h_addr_list);
c = connect(sd, &addr, addrlen);
if (c == -1)
{
    fprintf(stdout,"%s: Connect error %d\n",argv[0],errno);
    cleanup(sd);
}

/* Initialize the data buffer before sending */
for (i = 0; i < (BUFLEN - 1); i++)
{
   sbuf[i] = (i % 75) + 48;
}
sbuf[i] = '\0';

flags = 0;     /* initialize flags to 0 */

/* Send data to server */
if ((s = send(sd,&sbuf[0],BUFLEN,flags)) == -1)
{
    fprintf(stdout,"%s: Send error %d\n",argv[0],errno);
    cleanup(sd);
}

/* Receive data from the server */
totalr = 0;
while(totalr < BUFLEN)
{
    if ((r = recv(sd,&rbuf[totalr],BUFLEN-totalr,flags)) == -1)
    {
        fprintf(stdout,"%s: Recv error %d\n",argv[0],errno);
```

```
            cleanup(sd);
        }
        totalr += r;
    }

    /* NULL terminate the sent and received strings */
    sbuf[s-1] = '\0';
    rbuf[totalr-1] = '\0';

    if (!strcmp(sbuf,rbuf))
    {
        fprintf(stdout,"sent and received data MATCHED\n");
    }
    else
    {
        fprintf(stdout,"sent and received data MISMATCHED\n");
        fprintf(stdout,"Sent: len = %d, string = %s\n",s,sbuf);
        fprintf(stdout,"Received: len = %d, string = %s\n",totalr,rbuf);
    }

    cleanup(sd);

}   /* end of main */

/*****************************************************************************/

void cleanup(sd)
int sd;
{
    /* Shutdown the socket */
    if (shutdown(sd,2) == -1)
        fprintf(stdout,"%s: Shutdown error\n",errno);

    exit();
}
```

Example Server Program in Pascal

```
{ Pascal compiler options }
$PASCAL '91790-18293 REV.6200 <940914.1500>'
$CDS ON $
$DEBUG ON$
$LINESIZE 70$

{
{       NAME: BSDSERVER
{     SOURCE: 91790-18293
{      RELOC: NONE
{       PGMR: RR
{
{     This program is the server example program for BSD socket.
{     A socket is created and binded to a well known port,
{     the socket waits for a connection from the client and echoes all
{     the data that is received over the socket connection from the client.
{
{     To link this program, see the attached "bsdserver.lod" file.
{}

PROGRAM bsdserver(input,output);

LABEL 99;

IMPORT
{ This is the file that needs to be searched in order to
{ resolve references to ERRNO and ERRNO2. The file is
{ called errnodec.rel and is shipped with the product
{ in the /NS1000/REL directory. The network manager will
{ have to copy this file from the /NS1000/REL directory to
{ the /INCLUDES directory or the /LIBRARIES directory in order
{ for the search to be successful in the default case.
{ Look at the $SEARCH$ compiler directive notes in the
{ Pascal/1000 manual.
{}
  $search 'errnodec.rel'$ errnodec;

{ This is the include file for Berkeley sockets. It contains
{ all the data structures and constant definitions needed for
{ sockets programming. It is shipped as /NS1000/INCLUDE/socket.pasi.
{ The network manager will need to copy this file into the
{ include search path as defined in the $INCLUDE directive in the
{ Pascal/1000 manual. The recommended target directory is /INCLUDES.
{ For the alpha sites, the install_ns1000.cmd file copies this
{ file into the /INCLUDE directory. Note that this is the C
{ include directory and NOT the Pascal one which is /INCLUDES.
{}
  $include 'socket.pasi'$

CONST
   SERV_PORT  = 2000;      { Server port number }
   NUMCONN    = 3;         { Max number of outstanding connections }
   NULL_PTR   = 0;         { for infinite timeout value }
   SETSOCKLEN = 7000;      { amount of data to be received }
   RCVLEN     = 7000;

TYPE
   CharArrayType = PACKED ARRAY [1..RCVLEN] of CHAR;

   DataType    = RECORD
    CASE INTEGER OF
     1: ( int1   : int);
     2: ( bytes  : CharArrayType);
    END;

   OptType     = RECORD
    CASE INTEGER OF
      1: ( int1   : int);
      2: ( bytes  : PACKED ARRAY [1..4] OF CHAR);
    END;

VAR
    af                   : int;
    so_type              : int;
    protocol             : int;
    addr                 : sockaddr_in;
    addrlen              : u_short;
    sd                   : int;
    sss,ssr,b,l,c,a,sl,sh : int;
    optlen               : int;
    opt                  : OptType;
    rmask,wmask,emask    : fd_setType;
    s,r,flags            : int;
    rdata                : Datatype;

{ This is an include file for all the socket calls in terms
{ of the data structures described in the socket.pasi file.
{ This file is shipped as /NS1000/INCLUDE/extcalls.pasi.
{ The network manager will have to copy it into an include
{ directory search path as for the socket.pasi file above.
{}

$INCLUDE  'extcalls.pasi' $

$TITLE 'FORWARD DECLARATIONS',PAGE$
{ ---------------------------------------------------------- }
{  FORWARD DECLARATIONS                                       }
{ ---------------------------------------------------------- }

PROCEDURE cleanup
   ( sd, a : int );
   FORWARD;

{ ------------------------------------------------------------------------ }
{ Procedure to set up the connection to server }
```

```
PROCEDURE setup;

VAR
    len   : int;
BEGIN
af                := AF_INET;
so_type           := SOCK_STREAM;
protocol          := IPPROTO_TCP;
addrlen           := 16;
addr.sin_family := AF_INET;

{ create the socket }
sd := socket(af,so_type,protocol);
IF ( sd = -1 ) THEN
   BEGIN
      writeln(output,' Socket: errno = ',errno,' errno2 = ',errno2);
      GOTO 99;
   END;


{ Set socket send buffer size }
optlen := 2;
opt.int1 := SETSOCKLEN;
sss := setsockopt(sd,SOL_SOCKET,SO_SNDBUF,ByteAdrOf(opt.int1,0),optlen);
IF ( sss = -1 ) THEN
   BEGIN
      writeln(output,' SetSockOptSnd: errno= ',errno,' errno2= ',errno2);
      cleanup(sd,-1);
   END;

{ Set socket receive buffer size }
optlen := 2;
opt.int1 := SETSOCKLEN;
ssr := setsockopt(sd,SOL_SOCKET,SO_RCVBUF,ByteAdrOf(opt.int1,0),optlen);
IF ( ssr = -1 ) THEN
   BEGIN
      writeln(output,' SetSockOptRcv: errno= ',errno,' errno2= ',errno2);
      cleanup(sd,-1);
   END;

{ Bind the listening socket to a known port address }
addr.sin_port := SERV_PORT;
b := bind(sd,AddressOf(addr.int1),addrlen);
IF ( b = -1 ) THEN
   BEGIN
      writeln(output,' Bind: errno = ',errno,' errno2 = ',errno2);
      cleanup(sd,-1);
   END;

{ Set up the socket to accept incoming connections }
l := listen(sd,NUMCONN);
IF ( l = -1 ) THEN
   BEGIN
```

```
         writeln(output,' Listen: errno = ',errno,' errno2 = ',errno2);
         cleanup(sd,-1);
      END;

   { Block until a connection request comes in }
   len := addrlen; { type coercion }
   a := accept(sd,AddressOf(addr.int1),AddressOf(len));
   IF ( a = -1 ) THEN
      BEGIN
         writeln(output,' Accept: errno = ',errno,' errno2 = ',errno2);
         cleanup(sd,-1);
      END;

   END;  { setup }

{ ---------------------------------------------------------------------- }
{ Procedure to receive and echo data back }

   PROCEDURE recv_send;

   VAR
      totalr : int;
   BEGIN
      totalr := 0;
      WHILE (totalr < RCVLEN) DO
         BEGIN
            { set up bitmasks to wait for data }
            FD_ZERO(AddressOf(rmask.int1));
            FD_ZERO(AddressOf(wmask.int1));
            FD_ZERO(AddressOf(emask.int1));
            FD_SET(a,AddressOf(rmask.int1));
            FD_SET(a,AddressOf(emask.int1));

            { Select on the accepted connection for a read.  }
            { The count parameter should be one plus the     }
            { max socket descriptor value. We are interested }
            { in sd. hence, count should be sd + 1.          }
            sl := select(a + 1,AddressOf(rmask.int1),
                               AddressOf(wmask.int1),
                               AddressOf(emask.int1),
                               NULL_PTR);
            IF ( sl = -1 ) THEN
               BEGIN
                  writeln(output,' Select: errno = ',errno,' errno2 =
',errno2);
                  cleanup(sd,a);
               END;

            { Check to see if the socket is exceptional selected }
            IF (FD_ISSET(a,AddressOf(emask.int1))) THEN
               BEGIN
                  writeln(output,'Select: UNEXPECTED EXC SIGNAL');
                  cleanup(sd,a);
               END;
            { If the socket is read selected, receive data }
            IF (FD_ISSET(a,AddressOf(rmask.int1))) THEN
               BEGIN
                  { receive data from client }
                  flags := 0;
                  r :=
recv(a,ByteAdrOf(rdata.int1,totalr),RCVLEN-totalr,flags);
                  totalr := totalr + r;
                  IF ( r = -1 ) THEN
                     BEGIN
                        writeln(output,'Recv: errno=',errno,'errno2=',errno2);
                        cleanup(sd,a);
                     END;

                  { Print out the data received }
                  writeln(output,' Number of bytes receive:',r);
                  writeln(output,' Receive:');
                  writeln(output, rdata.bytes);
                  writeln;
               END;  { IF read selected }

         END; { end while }

      { Send back the data received }
      flags := 0;
      s := send(a,ByteAdrOf(rdata.int1,0),RCVLEN,flags);
      IF ( s = -1 ) THEN
        writeln(output,'Send: errno=',errno,'errno2=',errno2);

   END; { recv_send }

{ ------------------------------------------------------------------------ }
{ Procedure to cleanup - shut down socket }

   PROCEDURE cleanup( sd, a : int );

   BEGIN

      { close the connection }
      IF ( a <> -1) THEN
         BEGIN
            sh := shutdown(a,0);
            IF ( sh = -1 )  THEN
               writeln(output,' Shutdown(a):errno=',errno,'errno2=',errno2);
         END;

      { close the connection }
      sh := shutdown(sd,0);
      IF ( sh = -1 )  THEN
         writeln(output,' Shutdown (sd): errno = ',errno,' errno2 = ',errno2);

      GOTO 99;

   END;  { procedure cleanup }

{ ------------------------------------------------------------------------ }
{ Main program }
```

```
BEGIN
   { listen on a known port number and accept
   { incoming connection requests.
   {}
   setup;

   { Send data to the remote port }
   recv_send;

   { done }
   cleanup(sd,a);

99:

END.
```

Example Client Program in Pascal

```
{ Pascal compiler options }
$PASCAL '91790-18292 REV.6200 <940914.1500>'
$CDS ON $
$DEBUG ON$
$LINESIZE 70$

{
{       NAME: BSDCLIENT
{     SOURCE: 91790-18292
{      RELOC: NONE
{       PGMR: RR
{
{     This program is the client example program for BSD socket.
{     See the program "bsdserver.pas" for the server example.
{     This program prompts the user for the name of the server machine.
{     The routine gethostbyname() is used to resolve the IP address of the
{     server machine.  A socket is created, binded to a known port.
{     After establishing a connection with the server, an initialized data
{     buffer is sent to the server which will be echoed back and compared
{     to the input buffer.  If the sent and received buffers are identical,
{     a match message will be printed on the screen, otherwise a mismatch
{     message along with the sent and received buffers will be output.
{
{     To link this program, see the attached "bsdclient.lod" file.
{     To run this program:
{       - make sure that the remote node exists in the file "/etc/hosts",
{       - start the server program on the remote node,
{       - start this program by typing in the string: "bsdclient".
{}

PROGRAM bsdclient(input,output);

LABEL 99;

IMPORT
{ This is the file that needs to be searched in order to
{ resolve references to ERRNO and ERRNO2. The file is
{ called errnodec.rel and is shipped with the product
{ in the /NS1000/REL directory. The network manager will
{ have to copy this file from the /NS1000/REL directory to
{ the /INCLUDES directory or the /LIBRARIES directory in order
{ for the search to be successful in the default case.
{ Look at the $SEARCH$ compiler directive notes in the
{ Pascal/1000 manual.
{}
  $search 'errnodec.rel'$ errnodec;

{ This is the include file for Berkeley sockets. It contains
{ all the data structures and constant definitions needed for
{ sockets programming. It is shipped as /NS1000/INCLUDE/socket.pasi.
{ The network manager will need to copy this file into the
{ include search path as defined in the $INCLUDE directive in the
{ Pascal/1000 manual. The recommended target directory is /INCLUDES.
{ For the alpha sites, the install_ns1000.cmd file copies this
{ file into the /INCLUDE directory. Note that this is the C
{ include directory and NOT the Pascal one which is /INCLUDES.
{}
  $include 'socket.pasi'$

CONST
   C_PORT     =  2001;       { Local port number }
   SERV_PORT  =  2000;       { Remote port number }
   NULL_PTR   =  0;          { for infinite timeout value }
   SENLEN     =  7000;       { size of data buffer in send }
   SETSOCKLEN =  7000;       { amount of data to be received }
   INETLEN    =  40;         { length of inetstr }

TYPE
   StrType       = String[20];
   InetArrayType = PACKED ARRAY [1..INETLEN] of CHAR;
   CharArrayType = PACKED ARRAY [1..SENLEN] of CHAR;

   DataType    = RECORD
    CASE INTEGER OF
     1: ( int1   : int);
     2: ( bytes  : CharArrayType);
    END;

   inetStrType = RECORD
    CASE INTEGER OF
     1: ( int1:  int);
     2: ( str:   InetArrayType);
    END;

   lptrType  = ^long;
   iptrType  = ^int;

   SomePtrType = RECORD
    CASE INTEGER OF
     1: ( int1   : int);
     2: ( iptr   : iptrType);
     3: ( lptr   : lptrType);
     4: ( cptr   : int);
     5: ( hptr   : HostentptrType);
    END;

   Unsigned16Type = RECORD
    CASE INTEGER OF
     1: (int1   : integer);
     2: (hiword : int;
         loword : int);
    END;

   OptType     = RECORD
    CASE INTEGER OF
```

```
      1: ( int1   : int);
      2: ( bytes  : PACKED ARRAY [1..4] OF CHAR);
    END;

VAR
    af                      : int;
    so_type                 : int;
    protocol                : int;
    sd,sss,ssr,b,l,c,sh     : int;
    addr                    : sockaddr_in;
    addrlen                 : u_short;
    ipaddr                  : in_addr;
    s,r,len,flags           : int;
    optlen                  : int;
    opt                     : OptType;
    sdata,rdata             : DataType;
    zerop                   : SomePtrType;
    inetp                   : SomePtrType;
    inetstr                 : inetStrType;
    h                       : HostentPtrType;
    tlog                    : int;
    prmpt                   : StrType;
    hoststr                 : StrType;
    tempvar                    : Unsigned16Type;

{ This is the include file for all the socket calls in terms
{ of the data structures described in the socket.pasi file.
{ This file is shipped as /NS1000/INCLUDE/extcalls.pasi.
{ The network manager will have to copy it into an include
{ directory serach path as for the socket.pasi file above.
{}

$INCLUDE  'extcalls.pasi' $

$TITLE 'FORWARD DECLARATIONS',PAGE$
{ ---------------------------------------------------------- }
{  FORWARD DECLARATIONS                                      }
{ ---------------------------------------------------------- }

PROCEDURE cleanup;
   FORWARD;

{ ----------------------------------------------------------------------- }
{ Procedure to get hostname and set up the IP address }

   PROCEDURE get_param;

   CONST
       BLANK   = ' ';
   VAR
       i   : int;
   BEGIN
      { Remind user to start the server program }
      SetStrlen(prmpt,0);
      writeln(output,'Have you started the server program?(y/n): _');
```

```
readln(input,prmpt);
IF (prmpt <> 'y') THEN
   BEGIN
      writeln(output,'Start server program and restart client');
      GOTO 99;
   END;

{ Prompt user for the hostname }
SetStrlen(hoststr,0);
writeln(output,'Enter remote host name :');
readln(input,hoststr);
hoststr := Strrtrim(Strltrim(hoststr));
i := strpos(hoststr,BLANK);

{ Convert the Pascal string into a C-style string }
IF (i <> 0) THEN
   BEGIN
      strmove((i - 1),hoststr,1,inetstr.str,1);
      { Now end the string with a C-style NULL character }
      inetstr.str[i] := chr(0);
   END
ELSE
   BEGIN
      strmove(strlen(hoststr),hoststr,1,inetstr.str,1);
      { Now end the string with a C-style NULL character }
      inetstr.str[strlen(hoststr) + 1] := chr(0);
   END;

{ Map the hostname to the IP address with gethostbyname. }
{ This call looks at /ETC/HOSTS.                         }
{ A char pointer is required, so use ByteAdrOf().        }
{ This is defined in "extcalls.pasi".                    }

h := gethostbyname(ByteAdrOf(inetstr.int1,0));
zerop.hptr := h;
IF ( zerop.int1 = NULL_PTR) THEN
   BEGIN
      writeln(output,'Error in gethostbyname');
      GOTO 99;
   END;

af               := AF_INET;
so_type          := SOCK_STREAM;
protocol         := IPPROTO_TCP;
addrlen          := 16;
addr.sin_family := AF_INET;

{ Here we need to do a little manipulation in order to    }
{ extract the host IP address. h_addr_list in the hostent }
{ structure is a pointer to an integer array of character }
{ pointers which in turn point to the 32 bit IP address.  }
{ So, first we'll extract the integer pointer.            }

inetp.int1 :=  h^.h_addr_list;

{ Dereference this to get the character pointer to the IP address. }
```

```
      inetp.int1 :=  inetp.iptr^;

      { Divide this value by 2, since this is a character pointer and  }
      { we really want to dereference the IP address which is a 32 bit }
      { integer.  Incidentally, we are also guaranteed that the IP     }
      { address will be stored in a location that is word aligned.     }
      { To avoid sign extension for address manipulation, tempvar is used. }

      tempvar.hiword := 0;
      tempvar.loword := inetp.int1;
      inetp.int1 := tempvar.int1 DIV 2;

      { Now dereference this to get the IP address }
      ipaddr.S_addr := inetp.lptr^;

   END;  { get_param }

{ ------------------------------------------------------------------------ }
{ Procedure to setup the connection }

   PROCEDURE setup;

   BEGIN
      { create the socket }
      sd := socket(af,so_type,protocol);
      IF ( sd = -1 ) THEN
         BEGIN
            writeln(output,' Socket: errno = ',errno,' errno2 = ',errno2);
            GOTO 99;
         END;

      { Set socket send buffer size }
      optlen := 2;
      opt.int1 := SETSOCKLEN;
      sss := setsockopt(sd,SOL_SOCKET,SO_SNDBUF,ByteAdrOf(opt.int1,0),optlen);
      IF ( sss = -1 ) THEN
         BEGIN
            writeln(output,' SetSockOptSnd: errno= ',errno,' errno2=
',errno2);
           cleanup;
         END;

      { Set socket receive buffer size }
      optlen := 2;
      opt.int1 := SETSOCKLEN;
      ssr := setsockopt(sd,SOL_SOCKET,SO_RCVBUF,ByteAdrOf(opt.int1,0),optlen);
      IF ( ssr = -1 ) THEN
         BEGIN
            writeln(output,' SetSockOptRcv: errno= ',errno,' errno2=
',errno2);
           cleanup;
         END;

      { bind the socket to a port address }
      addr.sin_port  := C_PORT;
      b := bind(sd,AddressOf(addr.int1),addrlen);
      IF ( b = -1 ) THEN
         BEGIN
            writeln(output,' Bind: errno = ',errno,' errno2 = ',errno2);
            cleanup;
         END;

      {Connect it to the remote side }
      addr.sin_port := SERV_PORT;
      addr.sin_addr := ipaddr;
      c := connect(sd,AddressOf(addr.int1),addrlen);
      IF ( c = -1 ) THEN
         BEGIN
            writeln(output,' Connect: errno = ',errno,' errno2 = ',errno2);
            cleanup;
         END;
   END;  { procedure setup }

{ ------------------------------------------------------------------------ }
{ Procedure to send and receive data }

   PROCEDURE send_recv;

   VAR
      i,j,totalr : int;

   BEGIN
     { Initialize send data buffer }
     FOR i := 1 TO (SENLEN -1) DO
        BEGIN
           j := (i mod 75) + 48;
           sdata.bytes[i] := chr(j);
        END;

     { Send data to server }
     flags := 0;
     s := send(sd,ByteAdrOf(sdata.int1,0),SENLEN,flags);
     IF ( s = -1 ) THEN
        BEGIN
           writeln(output,' Send: errno = ',errno,' errno2 = ',errno2);
           cleanup;
        END;

     { Receive data echoed from server }
     totalr := 0;
     WHILE (totalr < SENLEN) DO
        BEGIN
           flags := 0;
           r := recv(sd,ByteAdrOf(rdata.int1,totalr),SENLEN-totalr,flags);
           IF ( r = -1 ) THEN
              BEGIN
                 writeln(output,'Recv:errno = ',errno,'errno2=',errno2);
                 cleanup;
              END;
           totalr := totalr + r;
        END; { while loop }
     { Compare receive and send buffers }
     IF ( rdata.bytes = sdata.bytes ) THEN
        BEGIN
           writeln(output,'Send and Receive data MATCHED');
        END
     ELSE
        BEGIN
           writeln(output,'Send and Receive data MISMATCHED');
           writeln(output,' Send: len = ', s);
           writeln(output,' Send data :',sdata.bytes);
           writeln;
           writeln(output,' Receive: len = ', r);
           writeln(output,' Recv data:',rdata.bytes);
           writeln;
        END;
   END; { procedure send_recv }

{ ------------------------------------------------------------------------ }
{ Procedure to cleanup - shut down socket }

   PROCEDURE cleanup;

   BEGIN

      sh := shutdown(sd,2);
      IF ( sh = -1 ) THEN
         writeln(output,' Shutdown (sd): errno = ',errno,' errno2 = ',errno2);

      GOTO 99;

   END; { procedure cleanup }

{ ------------------------------------------------------------------------ }
{ Main program }

BEGIN
   { get the hostname and set up IP address }
   get_param;

   { set up the connection to the server }
   setup;

   { receive data from the remote side }
   send_recv;

   { done }
   cleanup;
99:

END.
```

Example Server Program in FORTRAN

```
FTN77,L,S
$cds on
$files(1,1)

      PROGRAM BSDSERVER(4,99),91790-18291 REV.6200 <930517.1000>
C
C       NAME: BSDSERVER
C     SOURCE: 91790-18291
C      RELOC: NONE
C       PGMR: RR
C
C     This program is the server example program for BSD socket.
C     A socket is created and binded to a well known port,
C     the socket waits for a connection from the client and echoes all
C     the data that is received over the socket connection from the client.
C     After all the data is received the socket is shut down and the program
C     exits.
C
C     To link this program, see the attached "bsdserver.lod" file.
C

      IMPLICIT None

C     Include the FORTRAN header file for Berkeley Sockets here.
$LIST OFF
      INCLUDE socket.ftni
$LIST ON

C     VARIABLE DECLARATIONS:
      INTEGER  NULL
      PARAMETER (NULL = 0)

C     SOCKET()
      INTEGER  AF
      INTEGER  SO_TYPE
      INTEGER  PROTO
      INTEGER  SD

C     BIND() : SERV_PORT is the server port address
      INTEGER  SERV_PORT
      PARAMETER (SERV_PORT = 10000)
      INTEGER  ADDRLEN
      INTEGER  B

      INTEGER*4 FLAGS

C     LISTEN()
      INTEGER   BACKLOG
      INTEGER   L

C     ACCEPT()
      INTEGER  A
```

```
C     SETSOCKOPT()
      INTEGER   SSS
      INTEGER   SSR
      INTEGER   SETSOCKLEN   $ PARAMETER (SETSOCKLEN = 7000)
      INTEGER   OPTLEN
      INTEGER*2 OPTINT

C     RECV()
      INTEGER   BUF_WORDLEN       $ PARAMETER (BUF_WORDLEN = 3500)
      INTEGER   BUF_BYTELEN       $ PARAMETER (BUF_BYTELEN = 7000)
      INTEGER*2 RDATA(BUF_WORDLEN)
      INTEGER   OFFSET
      INTEGER   R

C     SEND()
      CHARACTER SDATA(BUF_BYTELEN)
      EQUIVALENCE (SDATA, RDATA)
      INTEGER*2 i
      INTEGER   S

C     BITMASK routines and SELECT()
      INTEGER*4 RMAP
      INTEGER*4 WMAP
      INTEGER*4 EMAP
      INTEGER   SE
      INTEGER*4 LONGNULL
      PARAMETER (LONGNULL = 0)

C     SHUTDOWN()
      INTEGER   SH

      CHARACTER BLANK*1


C     Create a socket for the client. The value returned will
C     be used in the bind local to bind the client to a specific
C     port and then in a connect call to connect to the server.
      AF      = AF_INET
      SO_TYPE = SOCK_STREAM
      PROTO   = IPPROTO_TCP
      SD  = SOCKET(AF,SO_TYPE,PROTO)
      IF (SD .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in Socket', errno
          STOP
      ENDIF

C     Set the socket send buffer size.
      OPTLEN = 2
      OPTINT = SETSOCKLEN
      SSS = SETSOCKOPT(SD,SOL_SOCKET,SO_SNDBUF,
     +                 ByteAdrOf(OPTINT,0),OPTLEN)
      IF (SSS .EQ. -1) THEN
         WRITE(1,*) 'BSDCLIENT : Error in Setsockopt Send', errno
         STOP
      ENDIF
```

```
C     Set the socket receive buffer size.
      OPTLEN = 2
      OPTINT = SETSOCKLEN
      SSR = SETSOCKOPT(SD,SOL_SOCKET,SO_RCVBUF,
     +                 ByteAdrOf(OPTINT,0),OPTLEN)
      IF (SSR .EQ. -1) THEN
         WRITE(1,*) 'BSDCLIENT : Error in Setsockopt Receive', errno
         STOP
      ENDIF

C     Bind the socket to a specific port address. This is not
C     necessary but is shown as an example. Use the socket
C     descriptor obtained from a previous socket() call.
      SIN_FAMILY = AF_INET
      SIN_PORT   = SERV_PORT
      ADDRLEN    = 16
      B  = BIND(SD, AddressOf(SOCKADDR_IN),ADDRLEN)
      IF (B .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in Bind', errno
          GOTO 99
      ENDIF

C     Ready the socket for accepting incoming connection requests
      BACKLOG = 3
      L = LISTEN(SD,BACKLOG)
      IF (L .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in Listen', errno
          GOTO 99
      ENDIF

C     Accept incoming connections
      ADDRLEN    =  16
      A = ACCEPT(SD,AddressOf(SOCKADDR_IN),AddressOf(ADDRLEN))
      IF (A .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in accept', errno
          GOTO 99
      ENDIF

C     If we have come this far, then the connection has been
C     established and we are ready to receive data
C
C     Loop until receive all data.

      OFFSET = 0
      DO WHILE (OFFSET .LT. BUF_BYTELEN)
         CALL FD_ZERO(AddressOf(RMAP))
         CALL FD_ZERO(AddressOf(WMAP))
         CALL FD_ZERO(AddressOf(EMAP))
         CALL FD_SET(A,AddressOf(RMAP))
         CALL FD_SET(A,AddressOf(EMAP))

C     Select on the accepted connection for a read
C     or an exceptional signal indefinitely. The
C     count parameter should be (A+1) since
C     the count for the socket descriptors starts
```

```
C     from 0.

         SE = SELECT(A + 1,AddressOf(RMAP),
     >                AddressOf(WMAP),
     >                AddressOf(EMAP),
     >                LONGNULL)
         IF (SE .EQ. -1) THEN
            WRITE(1,*) 'BSDSERVER : Error in select', errno
            GOTO 88
         ENDIF

C     If the exceptional bit is set then the client has
C     shutdown and so, let's get rid of our sockets also.

         IF (FD_ISSET(A,AddressOf(EMAP)) .GT. 0) THEN
            GOTO 88
         ENDIF

C     Data is first received by the server
C     on the newly established connection.

         IF (FD_ISSET(A,AddressOf(RMAP)) .GT. 0) THEN
            FLAGS  = 0
            R = RECV(A,ByteAdrOf(RDATA,OFFSET),SETSOCKLEN-OFFSET,FLAGS)
            IF (R .EQ. -1) THEN
               WRITE(1,*) 'BSDSERVER : Error in recv', errno
               GOTO 88
            ENDIF
         ENDIF

         WRITE(1,*) 'receive: '
         WRITE(1,*) 'len = '
         WRITE(1,*) R
         WRITE(1,*) 'string = '
         WRITE(1,'(35A2)') (RDATA(i), i=1,BUF_WORDLEN)
         OFFSET = OFFSET + R
         WRITE(1,*) 'offset : ' , OFFSET

      END DO

C     Now send the data back to the client
      FLAGS  = 0
      OFFSET = 0
      S = SEND(A,ByteAdrOf(RDATA,OFFSET),BUF_BYTELEN,FLAGS)
      IF (S .EQ. -1) THEN
         WRITE(1,*) 'BSDSERVER : Error in send', errno
         GOTO 88
      ENDIF

C     Shut down sockets before exit

88    SH = SHUTDOWN(A,0)
      IF (SH .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in shutdown(A)', errno
      ENDIF
```

```
99    SH = SHUTDOWN(SD,0)
      IF (SH .EQ. -1) THEN
          WRITE(1,*) 'BSDSERVER : Error in shutdown(SD)', errno
      ENDIF

      END
```

Example Client Program in FORTRAN

```
FTN77,L,S
$cds on
$files(1,1)

      PROGRAM BSDCLIENT(4,99),91790-18290 REV.6200 <940914.1500>
C
C       NAME: BSDCLIENT
C     SOURCE: 91790-18290
C      RELOC: NONE
C       PGMR: RR
C
C     This program is the client examples program for BSD socket.
C     See the program "bsdserver.ftn" for the server example.
C     This program takes the name of a remote system as an argument.
C     The routine gethostbyname() is used to resolve the IP address of the
C     server machine.  A socket is created, binded to a known port.
C     After establishing a connection with the server, an initialized data
C     buffer is sent to the server which will be echoed back and compared
C     to the input buffer.  If the sent and received buffers are identical,
C     a match message will be printed on the screen, otherwise a mismatch
C     message along with the sent and received buffers will be printed.
C
C     To link this program, see the attached "bsdclient.lod" file.
C     To run this program:
C       - make sure that the remote node exists in the file "/etc/hosts",
C       - start the server program on the remote node,
C       - start this program by typing in the string: "bsdclient nodename".
C
C     A special technique is used to dereference pointers. This
C     basically involves declaring the entire data segment as an array
C     and accessing it as such. So in order to dereference a pointer,
C     the pointer is simply used as an index into the array
C     mentioned above and its value is obtained. For example, to
C     declare the array, use the alias directive to name an array
C     as a common block starting from absolute address 0. This
C     is followed by a COMMON declaration of the same name.
C     The elements of this common block need to be declared as
C     an array. Since FORTRAN does not perform array bounds
C     checking, this array can then be indexed with any value.
C     (See the chapter on Using FORTRAN 77 in the FORTRAN reference
C     manual for more details).
C
C     In order to obtain the double word value from an array declared
C     as single word elements, we need to use the ".DLD" instruction.
C     Since this is not a FORTRAN symbol, it is aliased to an
C     acceptable symbol.
C
C     Alias the ".DLD" RTE-A instruction to dld in order to
C     obtain the 32 bit IP address from the memory array whose elements
C     are declared as 16 bit integers.
C
```

```
$alias dld = '.DLD',direct

      IMPLICIT None

C     Include the FORTRAN header file for Berkeley Sockets here.
$LIST OFF
      INCLUDE socket.ftni
$LIST ON

C     VARIABLE DECLARATIONS:
      INTEGER  NULL
      PARAMETER (NULL = 0)
      INTEGER  HOSTPTR
      INTEGER*4 IPADDR

C     In order to dereference pointers, declare MEM to be a common
C     block starting from absolute address 0 and consisting of
C     an array of 16 bit elements. FORTRAN does not do array bounds
C     checking. Hence, to dereference any pointer,"p", simply
C     use MEM(p).

$alias /MEM/ = 0
      COMMON /MEM/MEM(0:1)
      INTEGER MEM
      INTEGER*4 DLD

C     SOCKET()
      INTEGER  AF
      INTEGER  SO_TYPE
      INTEGER  PROTO
      INTEGER  SD

C     BIND() : C_PORT is the client port address
      INTEGER  C_PORT
      PARAMETER (C_PORT = 10001)
      INTEGER  ADDRLEN
      INTEGER  B

C     CONNECT() : SERV_PORT is the server port address
      INTEGER  SERV_PORT
      PARAMETER (SERV_PORT  = 10000)
      INTEGER  C

C     SETSOCKOPT()
      INTEGER   SSS
      INTEGER   SSR
      INTEGER   SETSOCKLEN   $ PARAMETER (SETSOCKLEN = 7000)
      INTEGER   OPTLEN
      INTEGER*2 OPTINT

C     SEND()
      INTEGER   BUF_WORDLEN       $ PARAMETER (BUF_WORDLEN = 3500)
      INTEGER   BUF_BYTELEN       $ PARAMETER (BUF_BYTELEN = 7000)
      INTEGER*2 SDATA(BUF_WORDLEN)
      CHARACTER sbuffer(BUF_BYTELEN)
      EQUIVALENCE (sbuffer,SDATA)
      INTEGER   OFFSET
      INTEGER   S
      INTEGER*2 i
      INTEGER*4 FLAGS

C     RECV()
      INTEGER*2 RDATA(BUF_WORDLEN)
      INTEGER   R

C     SHUTDOWN()
      INTEGER   SH

      CHARACTER YES*1
      CHARACTER prmt*1

C     FOR GETST CALL:
      INTEGER*2 maxlen
      INTEGER*2 tlog
      INTEGER*2 nodename(32)
      CHARACTER cname(64)
      EQUIVALENCE (nodename(1),cname(1))
      DATA  YES/'y'/

C     Remind the user to start the server program
      WRITE(1,*) 'Have you started the server program?(y/n): _'
      READ (1, '(A1)') prmt
      IF (prmt .NE. YES) THEN
          WRITE (1,*) 'Start server program and restart client'
          STOP
      ENDIF

C     Get remote host name from command string
      maxlen = -48
      CALL GETST(nodename,maxlen,tlog)
      IF (tlog .EQ. 0) THEN
          WRITE (1,*) 'BSDclient : Usage: ru,BSDCLIENT nodename'
          STOP
      ENDIF

C     Null terminate the nodename string
C     The BSD IPC utilities require a null-terminated string.
      cname(tlog+1) = CHAR(0)

C     Use gethostbyname to get a pointer to hostent.
      HOSTPTR = GETHOSTBYNAME(ByteAdrOf(nodename(1),0))
      IF (HOSTPTR .EQ. NULL) THEN
          WRITE(1,*) 'BSDCLIENT : Error in Gethostbyname', errno
          STOP
      ENDIF

C     HOSTPTR is a pointer to a structure of type HOSTENT.
C     Since FORTRAN does not allow structures, we use
C     offsets to obtain values from the fields of the
C     structure. First we need to get the value stored in the
C     H_ADDR_LIST field. Since it is the fourth field
```

```
C     in the hostent structure and each field is a 16 bit
C     integer,
C             X = MEM(HOSTPTR + 4)
C     gives the value of H_ADDR_LIST in X. This in turn is a
C     pointer to an array of pointers. Each element in the
C     array points to an IP address. We are interested
C     in only the first one. So,
C             Y = MEM(X),
C     gives us this pointer to the IP address. Since
C     the IP address must also be guaranteed to be word aligned,
C     we can obtain the word address from Y by simply using the
C     logical shift.  Note that logical shift must be used here
C     instead of division by 2 since Y is a 16 bit integer and
C     sign extend could occur when the double load instruction
C     is used to access the 32 IP address.  The logical
C     shift (ishft), shift right one bit to accomplish the task.
C     Hence,
C             Z = ishft(Y, -1)
C     gives the word aligned pointer to the IP address.
C     We can now access the 32 bit IP address from the two memory
C     locations : MEM(Z) and MEM(Z + 1). However, it
C     is simpler to use the double load instruction which
C     transfers a 32 bit quantity. Hence,
C             IPADDR = DLD(MEM(Z)),
C     should return the 32 bit IP address in IPADDR which
C     has been declared as INTEGER*4. Putting it all
C     together, we get:
C
C       IPADDR = DLD(MEM(ISHFT(MEM(MEM(HOSTPTR + 4)),-1)))
C

      IPADDR = DLD(MEM(ISHFT(MEM(MEM(HOSTPTR + 4)),-1)))

C     Create a socket for the client. The value returned will
C     be used in the bind local to bind the client to a specific
C     port and then in a connect call to connect to the server.
      AF      = AF_INET
      SO_TYPE = SOCK_STREAM
      PROTO   = IPPROTO_TCP
      SD  = SOCKET(AF,SO_TYPE,PROTO)
      IF (SD .EQ. -1) THEN
          WRITE(1,*) 'BSDCLIENT : Error in Socket', errno
          STOP
      ENDIF

C     Set the socket send buffer size.
      OPTLEN = 2
      OPTINT = SETSOCKLEN
      SSS = SETSOCKOPT(SD,SOL_SOCKET,SO_SNDBUF,
     +                  ByteAdrOf(OPTINT,0),OPTLEN)
      IF (SSS .EQ. -1) THEN
         WRITE(1,*) 'BSDCLIENT : Error in Setsockopt Send', errno
         STOP
      ENDIF
```

```
C     Set the socket recv buffer size.
      OPTLEN = 2
      OPTINT = SETSOCKLEN
      SSR = SETSOCKOPT(SD,SOL_SOCKET,SO_RCVBUF,
     +                 ByteAdrOf(OPTINT,0),OPTLEN)
      IF (SSR .EQ. -1) THEN
         WRITE(1,*) 'BSDCLIENT : Error in Setsockopt Receive', errno
         STOP
      ENDIF

C     Bind the socket to a specific port address. This is not
C     necessary but is shown as an example. Use the socket
C     descriptor obtained from a previous socket() call.
      SIN_FAMILY = AF_INET
      SIN_PORT   = C_PORT
      ADDRLEN    = 16
      B  = BIND(SD, AddressOf(SOCKADDR_IN),ADDRLEN)
      IF (B .EQ. -1) THEN
          WRITE(1,*) 'BSDCLIENT : Error in Bind', errno
          GOTO 99
      ENDIF

C     Connect to the server
      SIN_FAMILY = AF_INET
      SIN_PORT   = SERV_PORT
      SIN_ADDR   = IPADDR
      ADDRLEN    = 16
      C = CONNECT(SD,AddressOf(SOCKADDR_IN),AddressOf(ADDRLEN))
      IF (C .EQ. -1) THEN
          WRITE(1,*) 'BSDCLIENT : Error in connect', errno
          GOTO 99
      ENDIF

C     If we have come this far, then the connection has been
C     established and we are ready to send and receive data
C

C     Initialize the data buffer.
      DO 10 i = 1,BUF_BYTELEN,1
         sbuffer(i) = CHAR(MOD(i,75) + 48)
10       CONTINUE

C     Data is sent to server on the newly established connection.

      FLAGS  = 0
      OFFSET = 0
      S = SEND(SD,ByteAdrOf(SDATA,OFFSET),BUF_BYTELEN,FLAGS)
      IF (S .EQ. -1) THEN
          WRITE(1,*) 'BSDCLIENT : Error in send', errno
          GOTO 99
      ENDIF

C     Loop until receives all data.

C     OFFSET = 0
```

```
      DO WHILE (OFFSET .LT. BUF_BYTELEN)
          FLAGS  = 0
          R = RECV(SD,ByteAdrOf(RDATA,OFFSET),BUF_BYTELEN-OFFSET,FLAGS)
          IF (R .EQ. -1) THEN
             WRITE(1,*) 'BSDCLIENT : Error in recv', errno
             GOTO 99
          ENDIF
          OFFSET = OFFSET + R
      END DO

C     Check for data mismatch
          IF (SDATA .EQ. RDATA) THEN
              WRITE(1,*) 'send and receive data MATCHED'
C             Print out data sent
              WRITE(1,*) 'send'
              WRITE(1,*) 'len = '
              WRITE(1,*) S
              WRITE(1,*) 'string = '
              WRITE(1,'(35A2)') (SDATA(i), i=1,BUF_WORDLEN)

          ELSE
              WRITE(1,*) 'send and receive data MISMATCHED'

C             Print out data sent
              WRITE(1,*) 'send'
              WRITE(1,*) 'len = '
              WRITE(1,*) S
              WRITE(1,*) 'string = '
              WRITE(1, '(35A2)') (SDATA(i), i=1,BUF_WORDLEN)

C             Print out data receive
              WRITE(1,*) 'receive'
              WRITE(1,*) 'len = '
              WRITE(1,*) R
              WRITE(1,*) 'string = '
              WRITE(1, '(35A2)') (RDATA(i), i=1,BUF_WORDLEN)

          ENDIF

C     Shutdown socket before exit
99    SH = SHUTDOWN(SD,2)
      IF (SH .EQ. -1) THEN
          WRITE(1,*) 'BSDCLIENT : Error in shutdown', errno
      ENDIF

      END
```

Example Server Load File

```
* BSDSERVER.LOD 91790-17111 REV.6200 <941003.0908>
***************************************************************
*
*       LINK COMMAND FILE FOR BSDSERVER EXAMPLE
*
***************************************************************
* (c) COPYRIGHT HEWLETT PACKARD COMPANY 1988.  ALL RIGHTS      *
* RESERVED.  NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,       *
* REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT *
* THE PRIOR WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.        *
***************************************************************
*                                                             *
* NAME:   BSDSERVER.LOD                                       *
* SOURCE: 91790-17111 (load file)                             *
* RELOC.: NONE                                                *
*                                                             *
***************************************************************
*
ec
li /libraries/bsd_cds.lib
li /libraries/hpc.lib
li /libraries/bigns_cds.lib
st 10000
re bsdserver.rel
en bsdserver.run
```

Example Client Load File

```
* BSDCLIENT.LOD 91790-17110 REV.6200 <941003.0910>
***************************************************************
*
*       LINK COMMAND FILE FOR BSDCLIENT EXAMPLE
*
***************************************************************
* (c) COPYRIGHT HEWLETT PACKARD COMPANY 1988.  ALL RIGHTS      *
* RESERVED.  NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,       *
* REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT *
* THE PRIOR WRITTEN CONSENT OF HEWLETT-PACKARD COMPANY.        *
***************************************************************
*                                                             *
* NAME:   BSDCLIENT.LOD                                       *
* SOURCE: 91790-17110 (load file)                             *
* RELOC.: NONE                                                *
*                                                             *
***************************************************************
*
ec
li /libraries/bsd_cds.lib
li /libraries/hpc.lib
li /libraries/bigns_cds.lib
st 16000
re bsdclient.rel
en bsdclient.run
```

Database and Header Files

This appendix describes the database and header files used in BSD IPC for C, Pascal, and FORTRAN programming.

HP 1000 BSD IPC provides several database files for static lookup and compatibility with UNIX-based BSD IPC programming.

Header files provide standard definition of macros and variables used in programming. The sources of the header files used in BSD IPC are listed in this appendix for easy referencing.

Database Files

The database files needed for BSD IPC are:

- `/etc/hosts`
- `/etc/networks`
- `/etc/protocols`
- `/etc/services`

These database files are described below.

Note Template files for the four database files are provided in the product tape under either the `/NS1000/EXAMPLES` subdirectory or the `/ARPA1000/EXAMPLES` subdirectory. The network manager should modify them to fit your local node's configuration, and move them to the `/ETC` directory on your system.

/etc/hosts File

The `/etc/hosts` file contains a list of accessible hosts. The `/etc/hosts` file associates IP addresses with mnemonic host names and alias names. It contains the names of other nodes in the network with which your local system can communicate.

When creating an `/etc/hosts` file, you can either enter the known nodes in the format shown below, or copy the file from another node and modify it. A template `/etc/hosts` file is provided in `/NS1000/EXAMPLES/hosts` file, shown in Figure B-1 below.

Each node in the `/etc/hosts` file has a one line entry. Each entry in the file must be in the following format:

```
    IP_address host [aliases]
```

Parameters

IP_address	The IP address that uniquely identifies the node. `IP_address` must be in internet "dot" notation: e.g., `192.6.1.1`. Refer to "IP Addresses" in the Glossary for more information on IP Addresses.
host	Name of the host. Host name can contain any printable character except spaces, newline, or the comment character (#). *Naming Convention*: The first nine characters should be unique for each network host.
aliases	Common name or names for the node. An *alias* is a substitute for *host*. *Aliases* are optional and are separated by spaces. *Naming Convention*: The first nine characters should be unique for each network host.

The `/etc/hosts` file is used by the following BSD IPC utilities:

- `endhostent()`—which closes the `/etc/hosts` file.
- `gethostbyaddr()`—which returns host information from the specified IP address.
- `gethostbyname()`—which returns host information from the specified host name.
- `gethostent()`—which reads the next line of the `/etc/hosts` file and returns host information on that host.
- `sethostent()`—which opens and rewinds the `/etc/hosts` file.

For more information about these utilities, refer to Section 5, "BSD IPC Utilities."

```
#
# HOSTS 91790-18298 REV.6100 <930928.1544>
# SOURCE: 91790-18298
#
# This is an example /ETC/HOSTS file. The format of this file is:
# <IP address> <name1> <name2> ....
#
# The <IP address> must be in dotted decimal notation. The
# names are simple ASCII strings separated by blanks. The first name
# is the official name of the host and all other names on the same
# line are considered aliases for the same host. All characters after
# the comment character "#" and up to the end of the line
# are ignored when processing this file
# with the Berkeley utility routines.
#
#
15.10.56.1 host1 alias11 alias12 alias13 # host1 and its aliases
15.10.56.2 host2 alias21 alias22 alias23 # host2 and its aliases
```

Figure B-1. Template /etc/hosts File

/etc/networks File

The `/etc/networks` file associates network addresses with mnemonic names and alias names. The `/etc/networks` file contains the name and address of known internet networks with which your local host can communicate. The network address is the part of an IP address that defines to which network a host belongs. Refer to "IP Addresses" in the glossary for more information.

You must create and configure this file for your local host if you want to use symbolic network names instead of addresses. A template `/etc/networks` file is provided in `/NS1000/EXAMPLES/networks` file, shown in Figure B-2 below.

Each network has a one line entry in the `/etc/networks` file, in the following format.

```
    network_name  network_address [aliases]
```

Parameters

network_name	Name of the internet network. Network names can contain any printable character except spaces, newline, or the comment character (#).
network_address	Network address that uniquely identifies the network. The network address must be in "dot" notation. See "IP Addresses" in the Glossary for more information on network addresses.
aliases	Common name or names for the network. An *alias* is a substitute for *network_name*. *Aliases* are optional.

The `/etc/networks` file is used by the following BSD IPC utilities:

- `getnetent()`—reads the next line of `/etc/networks` file and returns network information on that network.
- `getnetbyaddr()`—returns network information on the specified network number.
- `getnetbyname()`—returns network information on the specified network name.
- `setnetent()`—opens and rewinds the `/etc/networks` file.
- `endnetent()`—closes the `/etc/networks` file.

For more information about these utilities, refer to Section 5, "BSD IPC Utilities."

```
#
# NETWORKS 91790-18299 REV.6100 <930928.1545>
# SOURCE: 91790-18299
#
#
# This is an example file for /ETC/NETWORKS.
#
# The form for each entry is:
# <official network name>    <network number>    <aliases>
#
# The network number must be in dotted decimal notation and must
# reflect only the network portion of an IP address.
#
# The comment character is "#". All characters after the comment
# character are ignored up to the end of the line. The protocol
# name to be provided with the port number is optional.
#
loop    192.46.4    testlan
local    15         locallan
```

Figure B-2. Template /etc/networks File

/etc/protocols File

The `/etc/protocols` file contains the names and protocol numbers of all the protocols known to the local host.

A template `/etc/protocols` file is provided in the `/NS1000/EXAMPLES/protocols` file, shown in Figure B-3 below. Each protocol has a one line entry in the `/etc/protocols` file, in the following format.

```
protocol_name  protocol_number [aliases]
```

Parameters

protocol_name	Name of the protocol. Protocol names can contain any printable character except spaces, newline, or the comment character (#). Example: tcp.
protocol_number	Protocol number that identifies this protocol.
aliases	Common name or names for the protocol. An *alias* is a substitute for *protocol_name*. *Aliases* are optional.

The `/etc/protocols` file is used by the following BSD IPC utilities:

- `getprotoent()` —reads the next line of `/etc/protocols` file and returns protocol information on that entry.
- `getprotobynumber()` —returns protocol information on the specified protocol number.
- `getprotobyname()` —returns protocol information on the specified protocol name.
- `setprotoent()` —opens and rewinds the `/etc/protocols` file.
- `endprotoent()` —closes the `/etc/protocols` file.

For more information about these utilities, refer to Section 5, “BSD IPC Utilities.”

```
#
# PROTOCOLS 91790-18300 REV.6100 <931001.1004>
# SOURCE: 91790-18300
#
#
# This is an example file for /ETC/PROTOCOLS.
#
# The form for each entry is:
# <official protocol name> <protocol number> <aliases>
#
# The comment character is “#”. All characters after the comment
# character are ignored up to the end of the line. The protocol
# name to be provided with the port number is optional.
#
# Internet (IP) protocols
#
icmp    1       ICMP    # internet control message protocol
tcp     6       TCP     # transmission control protocol
udp     17      UDP     # user datagram protocol
```

Figure B-3. Template /etc/protocols File

/etc/services File

The `/etc/services` file associates port numbers with mnemonic service names and alias names. The `/etc/services` file contains the names, protocol names, and port numbers of all services known to the local host.

You must configure this file for your local host. A template `/etc/services` file is provided in file `/NS1000/EXAMPLES/services` , as shown in Figure B-4 below. Each service has a one line entry in the `/etc/services` file, in the following format.

```
    service_name  portnumber/protocol [aliases]
```

Parameters

service_name	Name of the service. Service names can contain any printable character except spaces, newline, or the comment character (#).
portnumber	Port number of the service. All requests for this service must use this port number.
protocol	Name of the protocol that the service uses. The protocol name is separated from the port number by a slash (e.g., 514/tcp). The protocol name is also listed in the `/etc/protocols` file.
aliases	Common name or names for the service. An *alias* is a substitute for *service_name*. *Aliases* are optional.

The `/etc/services` file is used by the following BSD IPC utilities:

- `getservent()` —reads the next line of `/etc/services` file and returns service information on that entry.
- `getservbyport()` —returns service information on the specified port number.
- `getservbyname()` —returns service information on the specified service name.
- `setservent()` —opens and rewinds the `/etc/services` file.
- `endservent()` —closes the `/etc/services` file.

For more information about these utilities, refer to Section 5, "BSD IPC Utilities."

```
# SERVICES 91790-18301 REV.6200 <941109.1843>
# SOURCE: 91790-18301
#
# This file is an example of the /ETC/SERVICES file.
#
# This file maps service names to TCP protocol addresses ("ports").
# The format for each entry is:
#    <service>   <portnumber>/<protocol>   <aliases>
#
ftp        21/tcp                      # File Transfer Protocol
telnet     23/tcp                      # TELNET virtual terminal
smtp       25/tcp    mail              # Simple Mail Transfer Protocol
time       37/tcp    timeserver        # Time
printer   515/tcp    spooler           # Remote print spooling
```

Figure B-4. Template /etc/services File

BSD IPC Header Files

Header files provides standard definitions of macros and variables used in programming. HP 1000 BSD IPC supplies several header files for programming in C, Pascal, and FORTRAN. These header files can be found in the `/NS1000/INCLUDE` directory. As part of installation, these header files are copies over to the `/INCLUDE` directory on the system, if it exists.

Header Files for C Programming

HP 1000 BSD IPC provides several header files for programming in C. Table B-1 lists the C program header files and their uses.

Table B-1. Header Files for C Programming

Header File	Use
`<errno.h>`	Defines the error codes and error mnemonics for HP 1000 BSD IPC.
`<fcntl.h>`	Defines the parameter values for `fcntl()`.
`<in.h>`	Defines standard values used for TCP/IP and Internet addressing.
`<netdb.h>`	Defines the data structures (`hostent`, `netent`, `protoent`, and `servent`) used for the database files (`/etc/hosts`, `/etc/networks`, `/etc/protocols`, and `/etc/services`), respectively.
`<socket.h>`	Defines standard values used for BSD IPC sockets.
`<types.h>`	Defines the data types used for BSD IPC programming.

The source code for these header files are listed on the following pages.

<errno.h> Include File for C

```
/* IN.H 91790-18302 REV.6200 <950120.1054>
 *
 *     NAME   : ERRNO.H
 *     SOURCE : 91790-18302
 *
 * Error number/name header file
 */

/* From C compiler */

extern int errno;                   /* global error location */
extern int errno2;                  /* RTE error number */

#ifndef _errheader
#define _errheader

#define ENFILE   1                  /* file table overflow */
#define EFNAME   2                  /* missing/invalid file name */
#define EOPTIONS 3                  /* fopen options are bad */
#define ENOENT   4                  /* undefined file/directory */
#define ECREATE  5                  /* could not create file */
#define EREAD1   6                  /* can't read after write without seek */
#define EREAD2   7                  /* attempted read of write only file */
#define EREAD3   8                  /* can't write after read without seek */
#define EREAD4   9                  /* can't write to read only file */
#define ENOBUF  10                  /* can't allocate IO buffer from heap */
#define EREADERR 11                 /* RTE says there is an error */

#define E2BIG    12                 /* arg list for spawns > 255 bytes */
#define EINVAL   13                 /* invalid modefalg for spawns */
#define ENOEXEC  14                 /* not executable or invalid format */
#define ENOMEM   15                 /* not enough memory for runstring */

#ifndef EDOM
#define EDOM   100
#endif
#ifndef ERANGE
#define ERANGE 101
#endif

/* Added for BSD IPC */

#define EINTR            201
#define EAGAIN           202
#define EFAULT           203
#define EMFILE           204
#define EPIPE            205
#define EMSGSIZE         215
#define ENOTSOCK         216
#define EDESTADDRREQ     217
#define EPROTOTYPE       219
#define ENOPROTOOPT      220
```

```
#define EPROTONOSUPPORT  221
#define ESOCKTNOSUPPORT  222
#define EOPNOTSUPP       223
#define EAFNOSUPPORT     225
#define EADDRINUSE       226
#define EADDRNOTAVAIL    227
#define ECONNRESET       232
#define ENOBUFS          233
#define EISCONN          234
#define ENOTCONN         235
#define ESHUTDOWN        236
#define ETIMEDOUT        238
#define ECONNREFUSED     239
#define EREMOTERELEASE   240
#define EHOSTDOWN        241
#define EHOSTUNREACH     242
#define EALREADY         244
#define EINPROGRESS      245
#define EWOULDBLOCK      246
#define EINTERR          299

#endif
```

<fcntl.h> Include File for C

```
/* FCNTL.H 91790-18285 REV.5240 <910328.1746>
* -----------------------------------------------------------
*
* (c) COPYRIGHT HEWLETT PACKARD COMPANY 1986. ALL RIGHTS
* RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
* REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
* THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
*
* -----------------------------------------------------------
*/

/*
 *    NAME   : FCNTL.H
 *    SOURCE : 91790-18285
*/

#ifndef _SYS_FCNTL_INCLUDED
#define _SYS_FCNTL_INCLUDED

/* cmd values for fcntl() */
#  define    F_GETFD     1    /* Get file descriptor flags */
#  define    F_SETFD     2    /* Set file descriptor flags */
#  define    F_GETFL     3    /* Get file flags */
#  define    F_SETFL     4    /* Set file flags */

/* Socket Status Flags Used for fcntl() */
#  define O_NONBLOCK    0200000    /* No delay */

#endif /* _SYS_FCNTL_INCLUDED */
```

<in.h> Include File for C

```
/* IN.H 91790-18283 REV.5240 <910328.1746>

* -----------------------------------------------------------
*
* (c) COPYRIGHT HEWLETT PACKARD COMPANY 1986. ALL RIGHTS
* RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
* REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
* THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
*
* -----------------------------------------------------------
*/

/*
*      NAME   : IN.H
*      SOURCE : 91790-18283
*/

#ifndef _SYS_IN_INCLUDED      /* allow multiple includes of this file */
#define _SYS_IN_INCLUDED      /* without causing compilation errors */

/*
 * Protocol numbers defined for use in the IP header protocol field.
 */

#define    IPPROTO_ICMP    1            /* control message protocol */
#define    IPPROTO_GGP     3            /* gateway^2 (deprecated) */
#define    IPPROTO_TCP     6            /* tcp */
#define    IPPROTO_EGP     8            /* exterior gateway protocol*/
#define    IPPROTO_PUP    12            /* pup */
#define    IPPROTO_UDP    17            /* user datagram protocol */
#define    IPPROTO_HELLO  63            /* exterior gateway protocol*/
#define    IPPROTO_ND     77            /* UNOFFICIAL net disk proto*/
#define    IPPROTO_PXP    241           /* HPPXP */

#define    IPPROTO_RAWIP  253           /* raw packet to IP*/
#define    IPPROTO_RAWIF  254           /* raw packet to interface */
#define    IPPROTO_RAW    255           /* raw protocol packet */
#define    IPPROTO_MAX    256

/*
 * Port/socket numbers: network standard functions
 */
#define    IPPORT_ECHO           7
#define    IPPORT_DISCARD        9
#define    IPPORT_SYSTAT        11
#define    IPPORT_DAYTIME       13
#define    IPPORT_NETSTAT       15
#define    IPPORT_FTP           21
#define    IPPORT_TELNET        23
#define    IPPORT_SMTP          25
#define    IPPORT_TIMESERVER    37
```

```
#define    IPPORT_NAMESERVER    42
#define    IPPORT_WHOIS         43

/*
 * UNIX TCP sockets
 */
#define    IPPORT_EXECSERVER    512
#define    IPPORT_LOGINSERVER   513
#define    IPPORT_CMDSERVER     514
#define    IPPORT_EFSSERVER     520

/*
 * Ports < IPPORT_RESERVED are reserved for
 * privileged processes (e.g. root).
 */
#define    IPPORT_RESERVED        1024

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        /* can be used for most tcp & ip code */
#define   s_host   S_un.S_un_b.s_b2   /* host on imp */
#define   s_net    S_un.S_un_b.s_b1   /* network */
#define   s_imp    S_un.S_un_w.s_w2   /* imp */
#define   s_impno  S_un.S_un_b.s_b4   /* imp # */
#define   s_lh     S_un.S_un_b.s_b3   /* logical host */
};

/*
 * Definitions of bits in internet address integers.
 */
#define    IN_CLASSA(i)          ((((long)(i))&0x80000000)==0)
#define    IN_CLASSA_NET         0xff000000
#define    IN_CLASSA_NSHIFT      24
#define    IN_CLASSA_HOST        0x00ffffff

#define    IN_CLASSB(i)          ((((long)(i))&0xc0000000)==0x80000000)
#define    IN_CLASSB_NET         0xffff0000
#define    IN_CLASSB_NSHIFT      16
#define    IN_CLASSB_HOST        0x0000ffff

#define    IN_CLASSC(i)          ((((long)(i))&0xc0000000)==0xc0000000)
#define    IN_CLASSC_NET         0xffffff00
#define    IN_CLASSC_NSHIFT      8
#define    IN_CLASSC_HOST        0x000000ff

#define    INADDR_ANY            (u_long)0x00000000
#define    INADDR_BROADCAST     (u_long)0xffffffff     /* must be masked */

#define    IN_MASK(i) \
((IN_CLASSC(i)) ? IN_CLASSC_NET \
        : ((IN_CLASSB(i)) ? IN_CLASSB_NET \
                  : IN_CLASSA_NET))

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

/*
 * Macros for number representation conversion.
 */
#define    ntohl(x)    (x)
#define    ntohs(x)    (x)
#define    htonl(x)    (x)
#define    htons(x)    (x)
#endif    /* not _SYS_IN_INCLUDED */
```

<netdb.h> Include File for C

```
/* NETDB.H 91790-18282 REV.5240 <910402.1332>
 *
 *----------------------------------------------------------
 *
 * (c) COPYRIGHT HEWLETT PACKARD COMPANY 1986. ALL RIGHTS
 * RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
 * REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
 * THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
 *
 * ----------------------------------------------------------
 */

/*
 *     NAME   : NETDB.H
 *     SOURCE : 91790-18282
 */

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 addresses from name server */
};

/*
 * Assumption here is that a network number
 * fits in 32 bits
 */
struct    netent {
    char         *n_name;        /* official name of net */
    char        **n_aliases ;    /* alias list */
    int           n_addrtype;    /* net address type */
    unsigned long n_net;         /* network # */
};

struct    servent {
    char    *s_name;        /* official service name */
    char   **s_aliases;     /* alias list */
    int      s_port;        /* port # */
    char    *s_proto;       /* protocol to use */
};

struct    protoent {
    char     *p_name;         /* official protocol name */
    char    **p_aliases;      /* alias list */
    int       p_proto;        /* protocol # */
};
```

<socket.h> Include File for C

```
/* SOCKET.H 91790-18281 REV.6200 <950323.1000>
 * -------------------------------------------------------------
 *
 * (c) COPYRIGHT HEWLETT PACKARD COMPANY 1991. ALL RIGHTS
 * RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
 * REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
 * THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
 *
 * -------------------------------------------------------------
 */

/*
 *     NAME   : SOCKET.H
 *     SOURCE : 91790-18281
 */

#ifndef _SYS_SOCKET_INCLUDED       /* allow multiple includes of this file */
#define    _SYS_SOCKET_INCLUDED    /* without causing compilation errors */

/*
 * Types
 */

#define    SOCK_STREAM     1        /* stream socket */
#define    SOCK_DGRAM      2        /* datagram socket */
#define    SOCK_RAW        3        /* raw-protocol interface */
#define    SOCK_RDM        4        /* reliably-delivered message */
#define    SOCK_SEQPACKET  5        /* sequenced packet stream */

/*
 * Option flags per-socket.
 */

#define    SO_DEBUG         0x01        /* turn on debugging info recording */
#define    SO_REUSEADDR     0x04        /* allow local address reuse */
#define    SO_KEEPALIVE     0x08        /* keep connections alive */
#define    SO_DONTROUTE     0x10        /* just use interface addresses */
#define    SO_BROADCAST     0x20        /* permit sending of broadcast msgs */
#define    SO_SNDBUF        0x1001      /* send buffer size */
#define    SO_RCVBUF        0x1002      /* receive buffer size */

/*
 * Address families.
 */

#define    AF_UNSPEC        0        /* unspecified */
#define    AF_UNIX          1        /* local to host (pipes, portals) */
#define    AF_INET          2        /* internetwork: UDP, TCP, etc. */
#define    AF_IMPLINK       3        /* arpanet imp addresses */
#define    AF_PUP           4        /* pup protocols: e.g. BSP */
#define    AF_CHAOS         5        /* mit CHAOS protocols */
#define    AF_NS            6        /* XEROX NS protocols */
```

```
#define    AF_NBS           7        /* nbs protocols */
#define    AF_ECMA          8        /* european computer manufacturers */
#define    AF_DATAKIT       9        /* datakit protocols */
#define    AF_CCITT         10        /* CCITT protocols, X.25 etc */
#define    AF_SNA           11        /* IBM SNA */
#define    AF_8023          12        /* raw IEEE 802.3 */
#define    AF_OSI           13
#define    AF_OSI_TEST      14

#define SOCK_ADDR_DATA_LEN    14              /* length of sa_data */
#define SOCK_ADDR_DATA_OFF    sizeof(u_short) /* offset to sa_data */

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

struct sockproto {
    u_short    sp_family;                     /* address family */
    u_short    sp_protocol;                   /* protocol */
};

/*
 * Protocol families, same as address families for now.
 */
#define    PF_UNSPEC      AF_UNSPEC
#define    PF_UNIX        AF_UNIX
#define    PF_INET        AF_INET
#define    PF_IMPLINK     AF_IMPLINK
#define    PF_PUP         AF_PUP
#define    PF_CHAOS       AF_CHAOS
#define    PF_NS          AF_NS
#define    PF_NBS         AF_NBS
#define    PF_ECMA        AF_ECMA
#define    PF_DATAKIT     AF_DATAKIT
#define    PF_CCITT       AF_CCITT
#define    PF_SNA         AF_SNA
#define    PF_8023        AF_8023
#define    PF_OSI         AF_OSI
#define    PF_OSI_TEST    AF_OSI_TEST

/*
 * Level number for (get/set)sockopt() to apply to socket itself.
 */
#define    SOL_SOCKET    0xffff        /* options for socket level */

/*
 * Maximum queue length specifiable by listen.
 */
#define    SOMAXCONN    5

struct msghdr {
    caddr_t    msg_name;           /* optional address */
    int        msg_namelen;        /* size of address */
    struct     iovec *msg_iov;     /* scatter/gather array */
    int        msg_iovlen;         /* # elements in msg_iov */
    caddr_t    msg_accrights;      /* access rights sent/received */
    int        msg_accrightslen;
};

#define    MSG_OOB          0x1        /* process out-of-band data */
#define    MSG_PEEK         0x2        /* peek at incoming message */
#define    MSG_DONTROUTE    0x4        /* send without using routing tables */

#define    MSG_MAXIOVLEN    16

/*
 * User settable options ( used with setsockopt).
 */
#define TCP_NODELAY          0x01       /* don't delay send to coalesce */
#define TCP_MAXSEG           0x02       /* set Maximum segment size */


  /* BSDIPC system calls */
  extern int accept();
  extern int bind();
  extern int connect();
  extern u_long fcntl();
  extern int getpeername();
  extern int getsockname();
  extern int getsockopt();
  extern int setsockopt();
  extern int listen();
  extern int recv();
  extern int recvmsg();
  extern int recvfrom();
  extern int send();
  extern int sendmsg();
  extern int sendto();
  extern int shutdown();
  extern int socket();

  /* BSDIPC libary routines */
  extern struct hostent *gethostent();
  extern struct hostent *gethostbyname();
  extern struct hostent *gethostbyaddr();
  extern int sethostent();
  extern int endhostent();
  extern struct netent *getnetent();
  extern struct netent *getnetbyname();
  extern struct netent *getnetbyaddr();
  extern int setnetent();
  extern int endnetent();
  extern struct protoent *getprotoent();
  extern struct protoent *getprotobyname();
  extern struct protoent *getprotobynumber();
  extern int setprotoent();
  extern int endprotoent();
  extern struct servent *getservent();
  extern struct servent *getservbyname();
  extern struct servent *getservbyport();
  extern int setservent();
  extern int endservent();
  extern u_long inet_addr();
  extern u_long inet_network();
  extern char *inet_ntoa();
  extern struct in_addr inet_makeaddr();
  extern u_long inet_lnaof();
  extern u_long inet_netof();
#endif    /* not _SYS_SOCKET_INCLUDED */
```

<types.h> Include File for C

```
/* TYPES.H 91790-18280 REV.6100 <930928.1551>
* -----------------------------------------------------------
*
* (c) COPYRIGHT HEWLETT PACKARD COMPANY 1986. ALL RIGHTS
* RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
* REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
* THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
*
* -----------------------------------------------------------
*/

/*
*     NAME   : TYPES.H
*     SOURCE : 91790-18280
*/

#ifndef _SYS_TYPES_INCLUDED
#define _SYS_TYPES_INCLUDED

typedef char *caddr_t;

typedef    unsigned char     u_char;
typedef    unsigned short    u_short;
typedef    unsigned int      u_int;
typedef    unsigned long     u_long;

/*
 * these macros are used for select().  select() uses bit masks of socket
 * descriptors in longs.  These macros manipulate such bit fields.
 */
#define FD_SETSIZE 32

typedef long fd_mask;
#define NFDBITS (sizeof(fd_mask) * 8)      /* 8 bits per byte */
#define howmany(x,y)  (((x)+((y)-1))/(y))

typedef struct fd_set {
  fd_mask fds_bits[howmany(FD_SETSIZE, NFDBITS)];
} fd_set;

#define FD_SET(n,p)     ((p)->fds_bits[(n)/NFDBITS] |= (1 << ((n) % NFDBITS)))
#define FD_CLR(n,p)     ((p)->fds_bits[(n)/NFDBITS] &= ~(1 << ((n) % NFDBITS)))
#define FD_ISSET(n,p)   ((p)->fds_bits[(n)/NFDBITS] & (1 << ((n) % NFDBITS)))
#define FD_ZERO(p)      memset((char *)(p), (char) 0, sizeof(*(p)))

/*
 * Define timeval structure to be used in select().
 */
struct timeval {
  unsigned longtv_sec;           /* seconds */
  longtv_usec;                   /* and microseconds */
};
```

```
/*
 * Define iovec structure to be used in sendmsg() and recvmsg().
 */
struct iovec {
        caddr_t  iov_base;
        int      iov_len;
};

#endif   /* _SYS_TYPES_INCLUDED */
```

Header Files for Pascal Programming

Two header files are provided BSD IPC programming in Pascal, `EXTCALLS.PASI` and `SOCKET.PASI.`

The following is the `SOCKET.PASI` file:

```
{ SOCKET.PASI 91790-18278 REV.6200 <950120.1055> }
{ -----------------------------------------------------------

  (c) COPYRIGHT HEWLETT PACKARD COMPANY 1991. ALL RIGHTS
  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
  REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
  THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.

  ------------------------------------------------------------ }

{}
{    NAME : SOCKET.PASI
{  SOURCE : 91790-18278
{}

{########### Types.h #########}
CONST
  MAX_16   = 32767;

TYPE
  short      = -32768..32767;
  int        = -32768..32767;
  long       = INTEGER;
  u_char     = 0..255;
  u_short    = short;
  u_long     = long;

  iovec   = RECORD
       iov_base   : int;       { Byte pointer }
       iov_len    : int;
     END;

  timeval =  RECORD
    CASE INTEGER OF
      1: (int1 : int);
      2: (tv_sec : long;
          tv_usec : long);
    END;

  fd_setType  = RECORD
    CASE INTEGER OF
    1: (int1     : int);
    2: (bitmask  : INTEGER);
    END;

{############ Socket.h #########}
```

```
CONST
   SOCK_STREAM = 1;
   SOCK_DGRAM  = 2;
   SOCK_RAW    = 3;

{ Option Flags }

{ Address families }
   AF_UNSPEC   = 0;
   AF_UNIX     = 1;
   AF_INET     = 2;
   AF_OSI      = 13;

CONST
   SOCK_ADDR_DATA_LEN = 14;
   SOCK_ADDR_WORD_OFF =  1;
   SOCK_ADDR_BYTE_OFF =  2;

TYPE
   sockaddr = PACKED RECORD
     CASE INTEGER OF
     1: (int1 : int);
     2: (sa_family  :  int;
         sa_data    :  PACKED ARRAY [1..SOCK_ADDR_DATA_LEN] of CHAR);
     END;

   sockproto = PACKED RECORD
     CASE INTEGER OF
     1: (int1 : int);
     2: (sp_family      :  int;          { Address family }
         sp_protocol    : int);          { Protocol }
     END;

{ Protocol Families }
CONST
   PF_UNSPEC = AF_UNSPEC ;
   PF_UNIX   = AF_UNIX   ;
   PF_INET   = AF_INET   ;

{ Flags for RECV and SEND calls }
CONST
   MSG_OOB         = 1;
   MSG_PEEK        = 2;
   MSG_DONTROUTE   = 4;

{ Getsockopt level }
CONST
   SOL_SOCKET      = -1;

{ Socket level options }
CONST
   SO_REUSEADDR    = 4;
   SO_KEEPALIVE    = 8;
   SO_SNDBUF       = hex1('1001');
   SO_RCVBUF       = hex1('1002');
```

```
{ TCP level options }
   TCP_NODELAY     = 1;
   TCP_MAXSEG      = 2;

{ Declarations for SENDMSG and RECVMSG }
TYPE

   msghdr =  RECORD
      CASE INTEGER OF
        1 : ( int1  : int );
        2 : ( msg_name         : int;         { Byte pointer to caddr_t }
              msg_namelen      : int;
              msg_iov          : int;         { Word pointer to iovec }
              msg_iovlen       : int;
              msg_accrights    : int;         { Byte pointer to caddr_t}
              msg_accrightslen : int);
      END;

{############## in.h ############# }
CONST
   IPPROTO_TCP = 6;        { TCP }

TYPE
   in_addr = PACKED RECORD
     CASE INTEGER OF
        1 : ( int1  : int );
        2 : (
                 s_b1 : CHAR;
                 s_b2 : CHAR;
                 s_b3 : CHAR;
                 s_b4 : CHAR) ;
        3 : ( s_w1 : int;
              s_w2 : int);
        4 : ( S_addr : long);
     END;

   sockaddr_in  = RECORD
      CASE INTEGER OF
        1 : ( int1 : int );
        2 : ( sin_family   :  int;
              sin_port     :  int;
              sin_addr     :  in_addr;
              sin_zero     :  PACKED ARRAY [1..8] of char );
      END;

{############### fcntl.h ##########}

{ Command options }
CONST
  F_GETFL     = 3;
  F_SETFL     = 4;

{ Command arguments }
```

```
CONST
  O_NONBLOCK    = octal('200000');

{############# netdb.h ###########}
TYPE
  hostent =  RECORD
    CASE INTEGER OF
      1 : ( int1 : int);
      2 : ( h_name      : int;        { Byte pointer }
            h_aliases   : int;        { Word pointer to byte address array }
            h_addrtype  : int;
            h_length    : int;
            h_addr_list : int);       { Word pointer to word address array }
    END;

  hostentPtrType = ^hostent;

  netent  =  RECORD
    CASE INTEGER OF
      1 : ( int1 : int);
      2 : ( n_name     : int;       { Byte pointer }
            n_aliases  : int;       { Word pointer to byte address array}
            n_addrType : int;
            n_net      : long);
     END;

  netentPtrType = ^netent;

  protoent =  RECORD
  CASE INTEGER OF
      1 : ( int1 : int);
      2 : ( p_name     : int;       { Byte pointer }
            p_aliases  : int;       { Word pointer to byte address array }
            p_proto    : int);
     END;

  protoentPtrType = ^protoent;

  servent = RECORD
    CASE INTEGER OF
      1 : ( int1 : int);
      2 : ( s_name     : int;       { Byte pointer }
            s_aliases  : int;       { Word pointer to byte address array }
            s_port     : int;
            s_proto    : int);      { Byte pointer }
     END;

  serventPtrtype = ^servent;

{ ################### errno.h ########### }
CONST
  ENFILE          =  1;
  EINVAL          =  13;
  EINTR           =  201;
  EAGAIN          =  202;
```

```
EFAULT          =  203;
EMFILE          =  204;
EPIPE           =  205;
EMSGSIZE        =  215;
ENOTSOCK        =  216;
EDESTADDRREQ    =  217;
EPROTOTYPE      =  219;
ENOPROTOOPT     =  220;
EPROTONOSUPPORT =  221;
ESOCKTNOSUPPORT =  222;
EOPNOTSUPP      =  223;
EAFNOSUPPORT    =  225;
EADDRINUSE      =  226;
EADDRNOTAVAIL   =  227;
ECONNRESET      =  232;
ENOBUFS         =  233;
EISCONN         =  234;
ENOTCONN        =  235;
ESHUTDOWN       =  236;
ETIMEDOUT       =  238;
ECONNREFUSED    =  239;
EREMOTERELEASE  =  240;
EHOSTDOWN       =  241;
EHOSTUNREACH    =  242;
EALREADY        =  244;
EINPROGRESS     =  245;
EWOULDBLOCK     =  246;
EINTERR         =  299;
```

The following is the source of EXTCALLS.PASI.

```
{ EXTCALLS.PASI 91790-18279 REV.6200 <950331.0841> }
{ --------------------------------------------------------------

  (c) COPYRIGHT HEWLETT PACKARD COMPANY 1986. ALL RIGHTS
  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
  REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
  THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.

  -------------------------------------------------------------- }

{}
{     NAME : EXTCALLS.PASI
{   SOURCE : 91790-18279
{
{ This file contains the Pascal external declarations for the Berkeley
{ Socket routines. Use this as an include file in the Pascal main
{ program after the VAR decalartions.
{}

{
{    Socket routines
{}

FUNCTION Accept
  (     callsd        : int;
        paddr         : int;       { Word pointer }
        paddrlen      : int        { Word pointer }) : int;
EXTERNAL;

FUNCTION Bind
   (    sd          : int;
        paddr       : int;       { Word pointer }
        addrlen     : int  ): int;
EXTERNAL;

FUNCTION Connect
   (       srcsd      : int;
           paddr      : int;       { Word pointer }
           addrlen    : int   ): int;
EXTERNAL;

FUNCTION fcntl
    (     sd      : int;
          cmd     : int;
          arg     : long ) : long;
EXTERNAL;

FUNCTION getpeername
    (      sd       : int;
           paddr    : int;       { Word pointer }
           paddrlen : int        { Word pointer } ) : int;
EXTERNAL;
```

```
FUNCTION getsockname
    (      sd       : int;
           paddr    : int;       { Word pointer }
           paddrlen : int        { Word pointer }) : int;
EXTERNAL;

FUNCTION getsockopt
   (      sd           : int;
          level        : int;
          optname      : int;
          pbyte        : int;       { Byte pointer }
          poptlen      : int        { Word pointer }) : int;
EXTERNAL;

FUNCTION Listen
   (    sd          : int;
        backlog     : int  ): int;
EXTERNAL;

FUNCTION recv
   (      sd           : int;
          pbyte        : int;       { Byte pointer }
          dlen         : int;
          flags        : long ): int;
EXTERNAL;

FUNCTION recvmsg
   (      sd           : int;
          pmsg         : int;       { Word pointer }
          flags        : long ): int;
EXTERNAL;

FUNCTION recvfrom
   (      socket       : int;
      buffer       : int;
      len          : int;
      flags        : long;
      from         : int;
      fromlen      : int ): int;
EXTERNAL;

FUNCTION Select
   (      sdbound    : int;
          pcreadmap  : int;       { Word pointer }
          pcwritemap : int;       { Word pointer }
          pcexceptmap: int;       { Word pointer }
          ptimeval   : int        { Word pointer }): int;
EXTERNAL;

FUNCTION Send
   (       sd      : int;
           pbyte   : int;       { Byte pointer }
           dlen    : int;
           flags   : long) : int;
```

```
EXTERNAL;

FUNCTION sendmsg
   (      sd           : int;
          pmsg         : int;       { Word pointer }
          flags        : long ): int;
EXTERNAL;

FUNCTION sendto
   (      socket       : int;
          buffer       : int;
          len          : int;
          flags        : long;
          tohost       : int;
          tolen        : int ): int;
EXTERNAL;

FUNCTION setsockopt
   (      sd           : int;
          level        : int;
          optname      : int;
          pbyte        : int;       { Byte pointer }
          optlen       : int ) : int;
EXTERNAL;

FUNCTION Shutdown
   (       sd       : int;
           how      : int): int;
EXTERNAL;

FUNCTION Socket
  (     af            : int;
        socket_kind   : int;
        protocol      : int) : int;
EXTERNAL;

{ ########################
{   INET routines
{ ########################
{}

FUNCTION inet_addr
    (     cp          : int        { Byte pointer }): in_addr;
EXTERNAL;

FUNCTION inet_lnaof
    (    ipaddr     : in_addr): long;
EXTERNAL;

FUNCTION inet_makeaddr
    (    netaddr    : long;
         hostaddr   : long): in_addr;
EXTERNAL;
```

```
FUNCTION inet_netof
    (    ipaddr     : in_addr): long;
EXTERNAL;

FUNCTION inet_network
    (     cp          : int        { Byte pointer }): long;
EXTERNAL;

FUNCTION inet_ntoa
    (    ipaddr     : in_addr): int;       { Byte pointer }
EXTERNAL;

{ ########################
{   GETSERV*  routines
{ ########################
{}

FUNCTION getservent : ServentPtrType;
EXTERNAL;

FUNCTION getservbyname
   (   name     : int;       { Byte pointer }
       proto    : int        { Byte pointer }): ServentPtrType;
EXTERNAL;

FUNCTION getservbyport
   (   port     : int;
       proto    : int        { Byte pointer }): ServentPtrType;
EXTERNAL;

FUNCTION setservent
   (  stayopen  : int): int;
EXTERNAL;

FUNCTION endservent: ServentPtrType;
EXTERNAL;

{ ########################
{   GETPROTO*  routines
{ ########################
{}
FUNCTION getprotoent : ProtoentPtrType;
EXTERNAL;

FUNCTION getprotobyname
   (   name     : int        { Byte pointer }): ProtoentPtrType;
EXTERNAL;

FUNCTION getprotobynumber
   (   proto    : int): ProtoentPtrType;
EXTERNAL;

FUNCTION setprotoent
   (  stayopen  : int): int;
EXTERNAL;
```

```
FUNCTION endprotoent: ProtoentPtrType;
EXTERNAL;

{ ########################
{   GETNET*  routines
{ ########################
{}
FUNCTION getnetent : NetentPtrType;
EXTERNAL;

FUNCTION getnetbyname
   (   name     : int        { Byte pointer }): NetentPtrType;
EXTERNAL;

FUNCTION getnetbyaddr
   (   net    : long;
       family : int): NetentPtrType;
EXTERNAL;

FUNCTION setnetent
   (  stayopen  : int): int;
EXTERNAL;

FUNCTION endnetent: NetentPtrType;
EXTERNAL;

{ ########################
{   GETHOST*  routines
{ ########################
{}

FUNCTION gethostent : HostentPtrType;
EXTERNAL;

FUNCTION gethostbyname
   (   name     : int        { Byte pointer }): HostentPtrType;
EXTERNAL;

FUNCTION gethostbyaddr
   (   hostaddr: int;       { Byte pointer }
       len     : int;
       family  : int): HostentPtrType;
EXTERNAL;

FUNCTION sethostent
   (  stayopen  : int): int;
EXTERNAL;

FUNCTION endhostent: HostentPtrType;
EXTERNAL;
```

```
{ ########################
{   Bitmask manipulation routines
{ ########################
{}

PROCEDURE FD_SET
    (      sd       : int;
           pbitmask : int        { Word pointer });
EXTERNAL;

PROCEDURE FD_CLR
    (      sd       : int;
           pbitmask : int        { Word pointer });
EXTERNAL;

FUNCTION FD_ISSET
    (     sd        : int;
          pbitmask  : int        { Word pointer }): BOOLEAN;
EXTERNAL;

PROCEDURE FD_ZERO
    (      pbitmask : int        { Word pointer });
EXTERNAL;

{ ########################
{   Miscellaneous  routines
{ ########################
{}

PROCEDURE free
    (     ptr       : int);
EXTERNAL;

FUNCTION ByteAdrOf
     ( VAR int1      : int;
           offset    : int): int;       { Byte pointer }
EXTERNAL;

FUNCTION AddressOf
      ( VAR int1     : int): int;       { Word pointer }
EXTERNAL;
```

Header File for FORTRAN Programming

One header file, `SOCKET.FTNI`, is provided for BSD IPC programming in FORTRAN.

```
C SOCKET.FTNI 91790-18288 REV.6200 <941115.1711>
C ----------------------------------------------------------
C
C  (c) COPYRIGHT HEWLETT PACKARD COMPANY 1991. ALL RIGHTS
C  RESERVED. NO PART OF THIS PROGRAM MAY BE PHOTOCOPIED,
C  REPRODUCED OR TRANSLATED TO ANOTHER PROGRAM LANGUAGE WITHOUT
C  THE PRIOR WRITTEN CONSENT OF THE HEWLETT-PACKARD COMPANY.
C
C  ----------------------------------------------------------

C      NAME : SOCKET.FTNI
C    SOURCE : 91790-18288

C    ################## ERRNO.H ##########
$ALIAS /ERRNO/='ERRNO',NOALLOCATE
$ALIAS /ERRNO2/='ERRNO2',NOALLOCATE

        INTEGER   ERRNO, ERRNO2
        COMMON    /ERRNO/ ERRNO
        COMMON    /ERRNO/ ERRNO2

C    ################## TYPES.H ##########
        INTEGER     IOVEC(2)
        INTEGER     IOV_BASE,IOV_LEN
        EQUIVALENCE (IOV_BASE,IOVEC(1)),(IOV_LEN, IOVEC(2))

        INTEGER*4   TIMEVAL(2)
        INTEGER*4   TV_SEC,TV_USEC
        EQUIVALENCE (TIMEVAL(1),TV_SEC),(TIMEVAL(2),TV_USEC)

        INTEGER*4   FD_SETTYPE
        INTEGER*4   BITMASK
        EQUIVALENCE (FD_SETTYPE,BITMASK)

C     ################## SOCKET.H ########
        INTEGER   SOCK_STREAM $ PARAMETER (SOCK_STREAM = 1)
        INTEGER   SOCK_DGRAM  $ PARAMETER (SOCK_DGRAM  = 2)
        INTEGER   SOCK_RAW    $ PARAMETER (SOCK_RAW    = 3)

        INTEGER   AF_UNSPEC   $PARAMETER (AF_UNSPEC   = 0)
        INTEGER   AF_UNIX     $PARAMETER (AF_UNIX     = 1)
        INTEGER   AF_INET     $PARAMETER (AF_INET     = 2)
        INTEGER   AF_OSI      $PARAMETER (AF_OSI      = 13)

        INTEGER   SOCK_ADDR_DATA_LEN
        PARAMETER (SOCK_ADDR_DATA_LEN = 14)
        INTEGER   SOCK_ADDR_WORD_OFF
        PARAMETER (SOCK_ADDR_WORD_OFF = 1)
        INTEGER   SOCK_ADDR_BYTE_OFF
        PARAMETER (SOCK_ADDR_BYTE_OFF = 2)
```

```
INTEGER   SOCKADDR((SOCK_ADDR_DATA_LEN + 1)/2)
INTEGER   SA_FAMILY
CHARACTER SA_DATA*(SOCK_ADDR_DATA_LEN)
EQUIVALENCE (SOCKADDR(1),SA_FAMILY)
EQUIVALENCE (SOCKADDR(2),SA_DATA)

INTEGER   SOCKPROTO(2)
INTEGER   SP_FAMILY
INTEGER   SP_PROTOCOL
EQUIVALENCE (SOCKPROTO(1),SP_FAMILY), (SOCKPROTO,SP_PROTOCOL)

INTEGER   PF_UNSPEC   $ PARAMETER (PF_UNSPEC   = AF_UNSPEC)
INTEGER   PF_UNIX     $ PARAMETER (PF_UNIX     = AF_UNIX)
INTEGER   PF_INET     $ PARAMETER (PF_INET     = AF_INET)

INTEGER   MSG_OOB     $ PARAMETER (MSG_OOB     = 1)
INTEGER   MSG_PEEK    $ PARAMETER (MSG_PEEK   =  2)
INTEGER   MSG_DONTROUTE     $ PARAMETER (MSG_DONTROUTE = 4)

INTEGER   SOL_SOCKET  $ PARAMETER (SOL_SOCKET = -1)

INTEGER   SO_REUSEADDR  $ PARAMETER (SO_REUSEADDR = 4)
INTEGER   SO_KEEPALIVE  $ PARAMETER (SO_KEEPALIVE = 8)
INTEGER   SO_SNDBUF     $ PARAMETER (SO_SNDBUF    = 4097)
INTEGER   SO_RCVBUF     $ PARAMETER (SO_RCVBUF    = 4098)

INTEGER   TCP_NODELAY   $ PARAMETER (TCP_NODELAY  = 1)
INTEGER   TCP_MAXSEG    $ PARAMETER (TCP_MAXSEG   = 2)

INTEGER   MSGHDR(6)
INTEGER   MSG_NAME,MSG_NAMELEN,MSG_IOV,MSG_IOVLEN
INTEGER   MSG_ACCRIGHTS,MSG_ACCRIGHTSLEN
EQUIVALENCE (MSGHDR(1),MSG_NAME)
EQUIVALENCE (MSGHDR(2),MSG_NAMELEN)
EQUIVALENCE (MSGHDR(3),MSG_IOV)
EQUIVALENCE (MSGHDR(4),MSG_IOVLEN)
EQUIVALENCE (MSGHDR(5),MSG_ACCRIGHTS)
EQUIVALENCE (MSGHDR(6),MSG_ACCRIGHTSLEN)

INTEGER   IPPROTO_TCP   $ PARAMETER (IPPROTO_TCP  = 6)
INTEGER   IPPROTO_UDP   $ PARAMETER (IPPROTO_UDP  = 17)

INTEGER   IN_ADDR(2)
INTEGER   S_W1,S_W2
INTEGER*4 S_ADDR
EQUIVALENCE (IN_ADDR(1),S_W1,S_ADDR)
EQUIVALENCE (IN_ADDR(2),S_W2)

INTEGER   SOCKADDR_IN(8)
INTEGER   SIN_FAMILY,SIN_PORT
INTEGER*4 SIN_ADDR
CHARACTER SIN_ZERO*(8)
EQUIVALENCE (SOCKADDR_IN(1),SIN_FAMILY)
EQUIVALENCE (SOCKADDR_IN(2),SIN_PORT)
```

```
          EQUIVALENCE (SOCKADDR_IN(3),SIN_ADDR)
          EQUIVALENCE (SOCKADDR_IN(5),SIN_ZERO)

          INTEGER   F_GETFL   $ PARAMETER (F_GETFL = 3)
          INTEGER   F_SETFL   $ PARAMETER (F_SETFL = 4)

          INTEGER*4   O_NONBLOCK
          PARAMETER (O_NONBLOCK = 65536)

          INTEGER   HOSTENT(5)
          INTEGER   H_NAME,H_ALIASES,H_ADDRTYPE
          INTEGER   H_LENGTH,H_ADDR_LIST
          EQUIVALENCE (HOSTENT(1),H_NAME)
          EQUIVALENCE (HOSTENT(2),H_ALIASES)
          EQUIVALENCE (HOSTENT(3),H_ADDRTYPE)
          EQUIVALENCE (HOSTENT(4),H_LENGTH)
          EQUIVALENCE (HOSTENT(5),H_ADDR_LIST)

          INTEGER   NETENT(5)
          INTEGER   N_NAME,N_ALIASES,N_ADDRTYPE
          INTEGER*4 N_NET
          EQUIVALENCE (NETENT(1),N_NAME)
          EQUIVALENCE (NETENT(2),N_ALIASES)
          EQUIVALENCE (NETENT(3),N_ADDRTYPE)
          EQUIVALENCE (NETENT(4),N_NET)

          INTEGER   PROTOENT(3)
          INTEGER   P_NAME,P_ALIASES,P_PROTO
          EQUIVALENCE (PROTOENT(1),P_NAME)
          EQUIVALENCE (PROTOENT(2),P_ALIASES)
          EQUIVALENCE (PROTOENT(3),P_PROTO)

          INTEGER   SERVENT(4)
          INTEGER   S_NAME,S_ALIASES,S_PORT,S_PROTO
          EQUIVALENCE (SERVENT(1),S_NAME)
          EQUIVALENCE (SERVENT(2),S_ALIASES)
          EQUIVALENCE (SERVENT(3),S_PORT)
          EQUIVALENCE (SERVENT(4),S_PROTO)

C     ################### ERRNO.H ############

      INTEGER  ENFILE           $PARAMETER (ENFILE          =  1)
      INTEGER  EINVAL           $PARAMETER (EINVAL          =  13)
      INTEGER  EINTR            $PARAMETER (EINTR           =  201)
      INTEGER  EAGAIN           $PARAMETER (EAGAIN          =  202)
      INTEGER  EFAULT           $PARAMETER (EFAULT          =  203)
      INTEGER  EMFILE           $PARAMETER (EMFILE          =  204)
      INTEGER  EPIPE            $PARAMETER (EPIPE           =  205)
      INTEGER  EMSGSIZE         $PARAMETER (EMSGSIZE        =  215)
      INTEGER  ENOTSOCK         $PARAMETER (ENOTSOCK        =  216)
      INTEGER  EDESTADDRREQ     $PARAMETER (EDESTADDRREQ    =  217)
      INTEGER  EPROTOTYPE       $PARAMETER (EPROTOTYPE      =  219)
      INTEGER  ENOPROTOOPT      $PARAMETER (ENOPROTOOPT     =  220)
      INTEGER  EPROTONOSUPPORT  $PARAMETER (EPROTONOSUPPORT =  221)
      INTEGER  ESOCKTNOSUPPORT  $PARAMETER (ESOCKTNOSUPPORT =  222)
      INTEGER  EOPNOTSUPP       $PARAMETER (EOPNOTSUPP      =  223)
      INTEGER  EAFNOSUPPORT     $PARAMETER (EAFNOSUPPORT    =  225)
      INTEGER  EADDRINUSE       $PARAMETER (EADDRINUSE      =  226)
      INTEGER  EADDRNOTAVAIL    $PARAMETER (EADDRNOTAVAIL   =  227)
      INTEGER  ECONNRESET       $PARAMETER (ECONNRESET      =  232)
      INTEGER  ENOBUFS          $PARAMETER (ENOBUFS         =  233)
      INTEGER  EISCONN          $PARAMETER (EISCONN         =  234)
      INTEGER  ENOTCONN         $PARAMETER (ENOTCONN        =  235)
      INTEGER  ESHUTDOWN        $PARAMETER (ESHUTDOWN       =  236)
      INTEGER  ETIMEDOUT        $PARAMETER (ETIMEDOUT       =  238)
      INTEGER  ECONNREFUSED     $PARAMETER (ECONNREFUSED    =  239)
      INTEGER  EREMOTERELEASE   $PARAMETER (EREMOTERELEASE  =  240)
      INTEGER  EHOSTDOWN        $PARAMETER (EHOSTDOWN       =  241)
      INTEGER  EHOSTUNREACH     $PARAMETER (EHOSTUNREACH    =  242)
      INTEGER  EALREADY         $PARAMETER (EALREADY        =  244)
      INTEGER  EINPROGRESS      $PARAMETER (EINPROGRESS     =  245)
      INTEGER  EWOULDBLOCK      $PARAMETER (EWOULDBLOCK     =  246)
      INTEGER  EINTERR          $PARAMETER (EINTERR         =  299)

C  TYPE ALL THE SOCKET FUNCTIONS
        INTEGER   ACCEPT
        INTEGER   BIND
        INTEGER   CONNECT
        INTEGER*4 FCNTL
        INTEGER   GETPEERNAME
        INTEGER   GETSOCKNAME
        INTEGER   GETSOCKOPT
        INTEGER   LISTEN
        INTEGER   RECV
        INTEGER   RECVFROM
        INTEGER   RECVMSG
        INTEGER   SELECT
        INTEGER   SEND
        INTEGER   SENDTO
        INTEGER   SENDMSG
        INTEGER   SETSOCKOPT
        INTEGER   SHUTDOWN
        INTEGER   SOCKET

C   TYPE ALL THE INET* FUNCTIONS
        INTEGER*4   INET_ADDR
        INTEGER*4   INET_LNAOF
        INTEGER*4   INET_MAKEADDR
        INTEGER*4   INET_NETOF
        INTEGER*4   INET_NETWORK
        INTEGER     INET_NTOA

C    TYPE ALL THE GETSERV* FUNCTIONS
        INTEGER   GETSERVENT
        INTEGER   GETSERVBYNAME
        INTEGER   GETSERVBYPORT
        INTEGER   SETSERVENT
        INTEGER   ENDSERVENT
```

```
C    TYPE ALL THE GETPROTO* FUNCTIONS
        INTEGER   GETPROTOENT
        INTEGER   GETPROTOBYNAME
        INTEGER   GETPROTOBYNUMBER
        INTEGER   SETPROTOENT
        INTEGER   ENDPROTOENT

C    TYPE ALL THE GETNET* FUNCTIONS
        INTEGER   GETNETENT
        INTEGER   GETNETBYNAME
        INTEGER   GETNETBYADDR
        INTEGER   SETNETENT
        INTEGER   ENDNETENT

C    TYPE ALL THE GETHOST* FUNCTIONS
        INTEGER   GETHOSTENT
        INTEGER   GETHOSTBYNAME
        INTEGER   GETHOSTBYADDR
        INTEGER   SETHOSTENT
        INTEGER   ENDHOSTENT

C    TYPE ALL THE BITMASK FUNCTIONS
        INTEGER   FD_SET
        INTEGER   FD_CLR
        INTEGER   FD_ISSET
        INTEGER   FD_ZERO

C    MISCELLANEOUS ROUTINES
        INTEGER   FREE
        INTEGER   BYTEADROF
        INTEGER   ADDRESSOF
```

C

Error Messages

Errno is a standard error variable used in UNIX programming. For portability, the C library (HPC.LIB) also returns error values in a global variable called *errno*. This appendix contains the list of error messages for HP 1000 BSD IPC.

Value of *errno*	Error Mnemonic	Meaning
1	[ENFILE]	Currently there are no resources available.
13	[EINVAL]	One of the following occurred: The value of a specified parameter is invalid. The socket is not a BSD IPC socket. The socket has already been shut down. The socket is not ready to accept connections yet. A `listen()` call must be done before an `accept()` call. The socket is already bound to an address.
202	[EAGAIN]	Nonblocking I/O is enabled and: 1. for `accept()`, no connection is present to be accepted. 2. for `send()`, the socket does not have space to accept any data at all.
203	[EEFAULT]	For `getsockopt()` and `setsockopt()`, the *optval* or *optlen* parameter is not valid.
204	[EMFILE]	The maximum number of socket descriptors for this process are already currently open. This could happen since sockets need to be created for internal use.
205	[EPIPE]	An attempt was made to send on a socket whose connection has been shut down by the remote peer process.
215	[EMSGSIZE]	In nonblocking mode, the socket requires that messages be sent atomically, and the message size exceeded the outbound buffer size.
216	[ENOTSOCK]	The socket descriptor, *socket*, is not a valid socket descriptor.
220	[ENOPROTOOPT]	The requested socket option is currently not set.
221	[EPROTONOSUPPORT]	The specified protocol is not supported.
222	[ESOCKTNOSUPPORT]	The specified socket type is not supported in this address family.

Value of *errno*	Error Mnemonic	Meaning
223	[EOPNOTSUPP]	The socket descriptor, *socket*, does not support this call or a parameter in this call.
225	[EAFNOSUPPORT]	Addresses in the specified address family cannot be used with this socket.
226	[EADDRINUSE]	The specified address is already in use.
227	[EADDRNOTAVAIL]	The specified address is invalid or not available.
232	[ECONNRESET]	Connection has been aborted by the remote process.
233	[ENOBUFS]	No buffer space is available. The call cannot be completed.
234	[EISCONN]	The socket is already connected.
235	[ENOTCONN]	The socket has not been connected yet.
236	[ESHUTDOWN]	The network software on the system is not running. Or the socket has already been shut down for send or receive.
238	[ETIMEDOUT]	Connection establishment timeout without establishing a connection.
239	[ECONNREFUSED]	The attempt to connect was rejected by the server.
240	[EREMOTERELEASE]	The remote side has done a send shutdown; hence, there will be no more data to receive.
241	[EHOSTDOWN]	The network software on the local host is not running.
242	[EHOSTUNREACH]	The network software was unable to determine a route to the destination host.
245	[EINPROGRESS]	Nonblocking I/O is enabled and the connection has been initiated. This is not a failure. Use `select()` to find out when the connection is complete.
299	[EINTERR]	This error requires HP notification.

D

Definition of Terms

Address Family Type

The socket address family type defines the address format to be used for the socket. The two address families provided for BSD IPC are:

- `AF_INET`: the Internet address family, which defines an address structure (`sockaddr_in`) of 16 bytes. `AF_INET` is the only address family type currently supported on the HP 1000 BSD IPC.

- `AF_UNIX`: the UNIX Domain address family, which defines an address structure of 110 bytes. `AF_UNIX` is used within a single machine, usually running UNIX operating system.

Internet Dot Notation

Specifies the format in which Internet (IP) addresses are specified. IP addresses using the "dot" notation take one of the following forms:

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

When four parts are specified, each is interpreted as a byte of data and assigned, from left to right, to the four bytes of an Internet (IP) address.

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

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

When only one part is given, the value is stored directly in the network address without any byte rearrangement.

All numbers supplied as parts in a "dot" notation may be decimal, octal, or hexadecimal, as specified in the C language (i.e., 0X or 0x implies hexadecimal; a leading 0 implies octal; otherwise, the number is interpreted as decimal).

IP Address

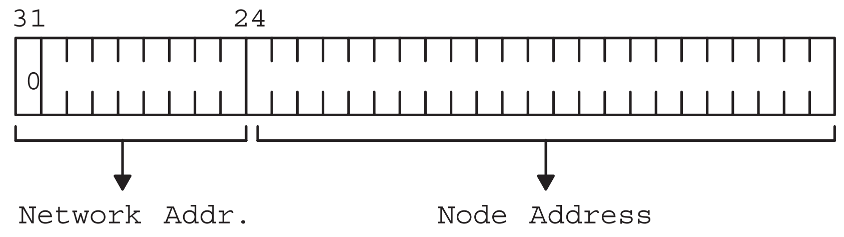
Also known as *Internet Addresses*. IP addresses are used in the internet network to identify a node within the network.

An IP address consists of two parts: a *network address*, which identifies the network; and a *node address*, which identifies a node within a network. A network address is concatenated with a node address to form the IP address and uniquely identify a node within a network within an internet.

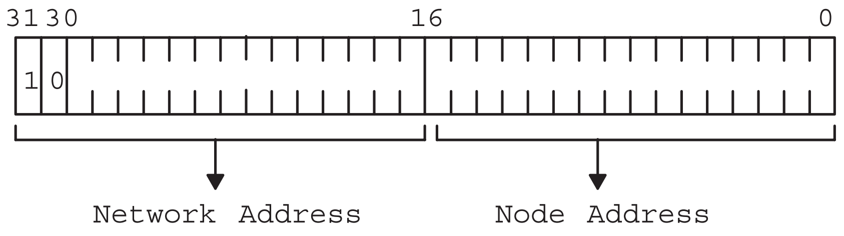
If you have several networks, you may want to consider using subnetting. In this case, an IP address has three parts: a network number, a *subnet* number, and a node number. Using subnet addressing is optional. Refer to “IP Addresses with Subnetting” after reading the following IP addressing information.

There are three IP address classes, each accommodating a different number of network and node addresses. The address classes are defined by the most significant bits of the address, as follows:

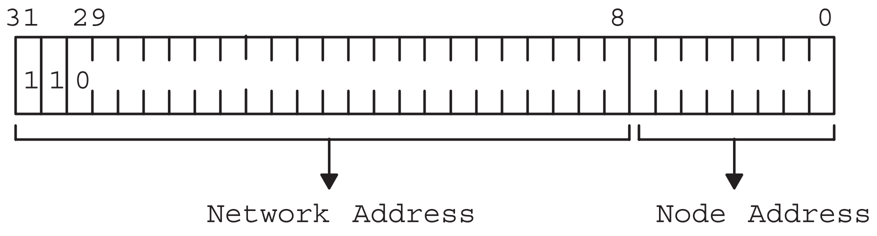
Class A address—



Class B address—



Class C address—



The address classes can also be broken down by address ranges. IP addresses are typically represented by converting the bits to decimal values an octet (8 bits) at a time, and separating each octet’s decimal value by a period (.). Therefore, IP addresses are typically of the following form:

nnn.nnn.nnn.nnn

where *nnn* is a number from 000 to 255, inclusive.

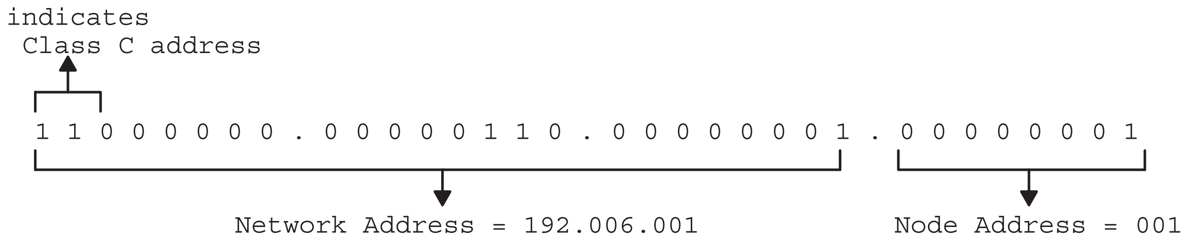
Table D-1 lists the number of networks and nodes and the address ranges for each address class.

Table D-1. IP Address Classes

Class	Networks	Nodes per Network	Address Range
A	127	16777215	000.000.000.000*—127.255.255.255
B	16383	65535	128.000.000.000—191.255.255.255
C	2097151	255	192.000.000.000—223.255.255.255
Reserved	–	–	224.000.000.000—255.255.255.255*

*Do not assign the network and node addresses to all zeros or all ones; they are reserved. The address of all ones (255.255.255.255) is used internally for broadcasting.

To determine a network address and node address from an IP address, you must separate the network and node address fields. For example, the bit representation of IP address `192.006.001.001` is separated as follows:



IP Addresses, with Subnetting

Subnetting is an optional addressing scheme that allows you to partition the *node address* portion of an internet address (IP address) into discrete *subnetworks*. The term, *subnetted network*, is sometimes used for a subnetwork.

The node address is divided into a *subnet number* and a *node number* to identify the node within this subnetwork. The *network address* portion of the IP address remains unchanged.

```
IP Address = Network Address + Node Address
           = Network Address + (Subnet Number + Node Number)
```

A subnetted network can communicate with a non-subnetted network.

Subnetting allows you to use one network address for two or more physically distinct networks. For example, if you have a large installation with many interconnected nodes, you could run into hardware configuration restrictions or performance degradation if you tried to place all nodes on the same physical network. With subnetting, you can install several smaller physical networks (connected via gateways) and have them all share the same network address. You then use different subnet addresses for each of these physical networks. Each network would actually be a *subnetwork*. In summary, a network address would identify a group of networks, and the subnet numbers identify the subnetworks.

Because subnetting allows you to use fewer network addresses, you may hide the internal structure of your company's networks. So, instead of assigning different network addresses to each physical network, a company needs only one network address. People outside the company need only know one network address to be able to send to any node in the company.

Subnet numbers are used like network numbers to distinguish each subnetwork. All nodes on the same physical LAN are assigned the same subnet number. Nodes on a different LAN have a separate subnet number. If the two networks are connected via a gateway and are to be subnetted, then each node in both networks will share the same network address.

Determining the number of subnetworks and the number of nodes in each subnetwork depends on how many bits are used in the node address portion of the IP address. For example, a Class C IP address uses 8 bits for the *node address* portion. Of these 8 bits, if 3 bits are used for subnet numbers, then we can have 7 subnetworks and 29 nodes per subnetwork. Node addresses of all zeros (000.000.000.000) or all ones (255.255.255.255) are reserved and are not allowed.

Class C address—

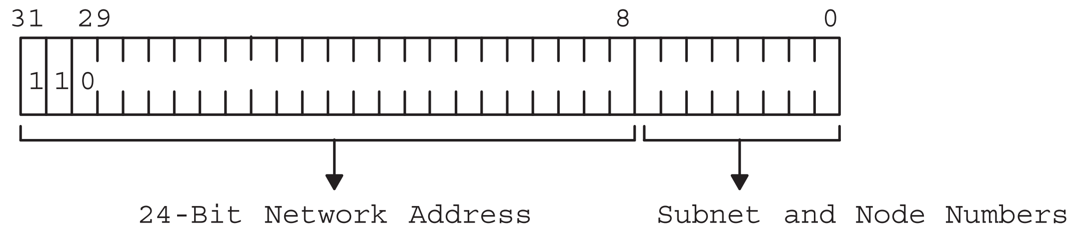


Table D-2 lists the valid IP addresses for up to 7 subnetworks and 29 nodes per subnetwork.

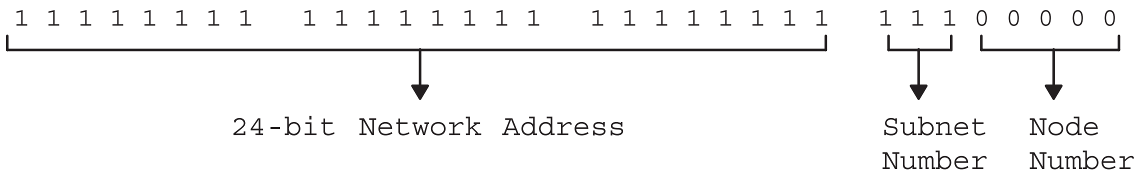
Table D-2. Subnet Addressing Example

Subnet Address (binary)	Subnet Address (decimal)	IP Address Range
000	0	000.000.000.000 is reserved and cannot be used
001	1	n.n.n.33—n.n.n.62
010	2	n.n.n.65—n.n.n.94
011	3	n.n.n.97—n.n.n.126
100	4	n.n.n.129—n.n.n.158
101	5	n.n.n.161—n.n.n.190
110	6	n.n.n.193—n.n.n.222
111	7	n.n.n.225—n.n.n.254
		255.255.255.255 is reserved and cannot be used; it is used internally for broadcasting

The vital part of subnetting is the 32-bit *subnet mask*. The subnet mask specifies the portion of the node address used to identify the subnet number. The remaining part of the node address is used to identify the node on that particular subnetwork.

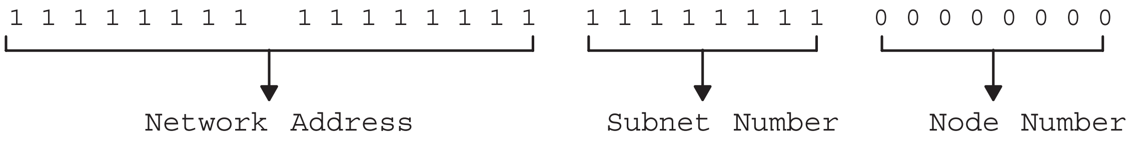
Bits, in the subnet mask, are set to 0 for the node number and 1 for the network address and subnet number. For the above example of a Class C IP address, the subnet mask would be as follows:

Class C address with subnet mask of 255.255.255.224 decimal—



In another example, given a Class B IP address with the following subnet mask means that the first two octets of the IP address identifies the main network, the third octet is the subnet number and the fourth octet identifies the node number:

Class B address—



All nodes in a subnetwork must use the same subnet mask.

The following example shows four subnetworks sharing a Class C address of 192.006.012.

Subnetwork A	nodes: 192.006.012.034 and 192.006.012.035 gateway node: G1 at 192.006.012.033 and 192.006.012.129 Subnet number = 1 (001 in binary) Node address range = 33-62 Subnet mask = 255.255.255.224
Subnetwork B	nodes: 192.006.012.067 and 192.006.012.066 gateway node: G2 at 192.006.012.065 and 192.006.012.130 Subnet number = 2 (010 in binary) Node address range = 65-95 Subnet mask = 255.255.255.224
Subnetwork C	nodes: 192.006.012.098 and 192.006.012.099 gateway node: G3 at 192.006.012.097 and 192.006.012.131 Subnet number = 3 (011 in binary) Node address range = 97-127 Subnet mask = 255.255.255.224
Subnetwork D	gateway nodes: G1, G2, and G3 at 192.006.012.129, 192.006.012.131, and 192.006.012.130 Subnet number = 4 (100 in binary) Node address range = 129-158 Subnet mask = 255.255.255.224

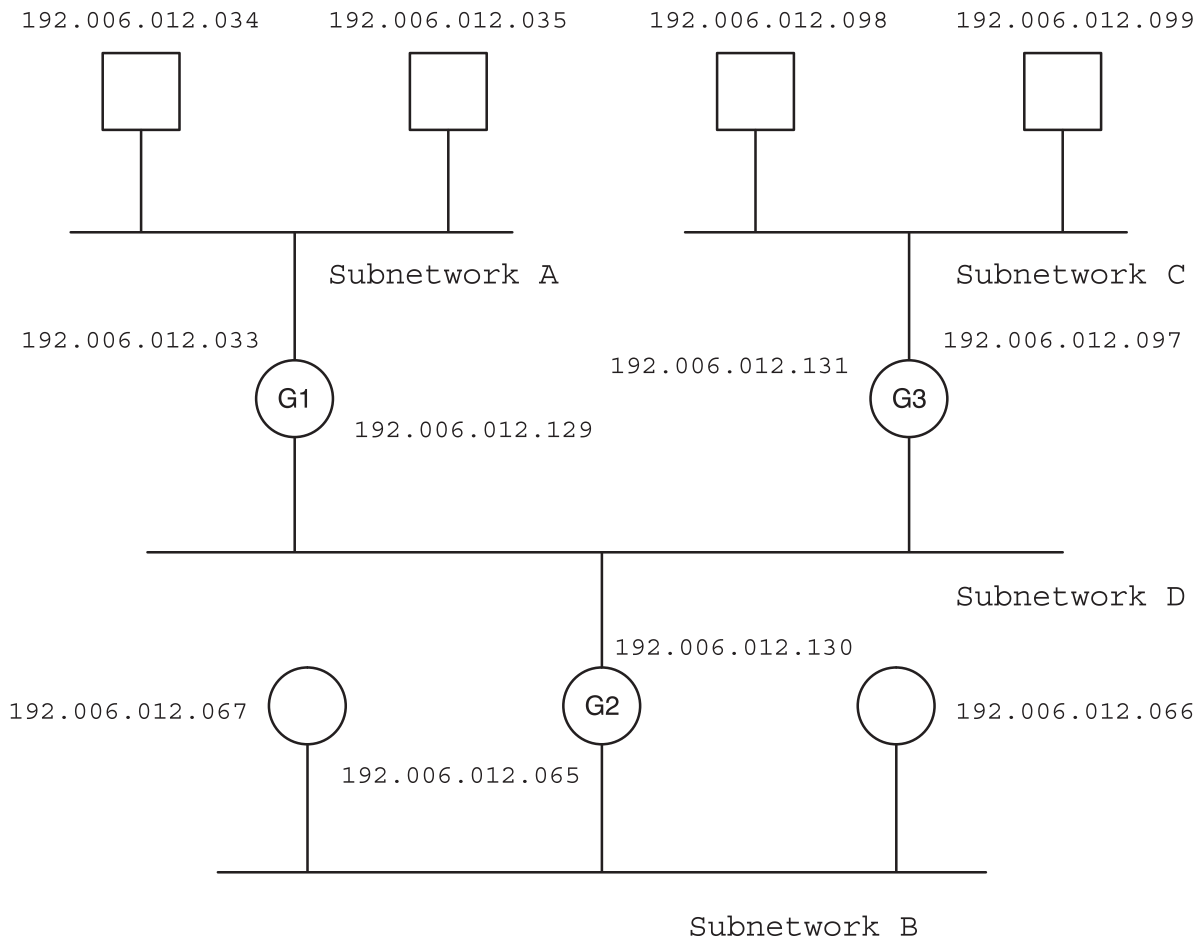


Figure D-1. Subnetted Network Example

IP Addresses, Assigning

You must assign an IP address as follows:

- You must assign an IP address for each NS-ARPA node in your network *except* nodes that only use DS/1000-IV Compatible Services.
- Do not assign any nodes with a reserved address of all zeros or all ones (see Table D-1).
- If a node is a member of more than one network, you must assign that node an IP address for each network of which it is a member.

When assigning IP addresses, you must determine network and node addresses, as described below.

To assign IP *network addresses*, follow these rules:

- Each network has a unique network address throughout the internet.
- All nodes in a network must have the same network address.

- If a node belongs to more than one network, it must have one and only one IP address for each network to which it belongs.
- Do not assign any networks with a reserved network address of all zeros or all ones (see Table D-1).

HP has obtained a block of Class C network addresses from DARPA to assign to HP customers. You can obtain Class C addresses that are unique within the ARPANET by contacting HP at the following address:

Network Administration Office, Dept. NET
Information Networks Division
Hewlett-Packard Company
19420 Homestead Road
Cupertino, California 95014

Note Although any address assigned from HP is unique within DARPA's ARPANET, this does not imply that your system is compatible with nor supported on the ARPANET.

If you attempt to communicate with external networks, you should consider the security risks of external networks accessing your network.

The IP addresses used in this manual are given as examples only. Do not use these addresses in your network.

To assign IP *node addresses*, follow these rules:

- Node addresses must be unique within each network, but they do not have to be unique throughout the internet. For example, you could have a node with node address 55 in Network 18 and a node with node address 55 in Network 21. You can assign node addresses according to your own needs, but they must be within the ranges for the IP address class that you are using.
- Do not assign any nodes with a reserved node address of all zeros or all ones (see Table D-1).
- If nodes support DS/1000-IV Compatible Services (RTE-RTE) or have RTR LIs, you may want to assign node addresses that are unique throughout the network and that correspond to the Router/1000 addresses.

Port Number

Port numbers are 16-bit integer numbers used to identify the services (user processes) on your system. The `/etc/services` file contains a list of services that are available on your NS-ARPA/1000 system. Each service has an assigned port number.

There are two types of port numbers: *well-known* port numbers and *ephemeral* (short-lived) port numbers. Well-known port numbers are Internet-specific port numbers which are used in the industry to identify standard server programs, such as FTP. For example, FTP servers are assigned the well-known port number `21`.

Because well-known port numbers are assigned to specific server processes, they are reserved and cannot be used by user programs. Port numbers 1-1023 are reserved and can only by used by superusers.

Ephemeral port numbers are used for client processes, because they are assigned to the processes only for the duration of the processes' run time. For a client process, you can use port number "0", and your host will automatically assign an available port number. To find out the assigned port number, use `getsockname()`.

Socket Address

The socket address is used to identify a BSD IPC socket on the HP 1000. The address is bound to a socket by the `bind()` call. A socket need to have a bound address before other processes can reference it and communicate with it. Socket addresses for HP 1000 BSD IPC are stored in a data variable of `sockaddr_in` type. The socket address consists of three fields:

- Address Family Type—`AF_INET`
- Port Number
- IP Address

Socket Type

Socket type defines the type of socket used. HP 1000 BSD IPC only supports socket type `SOCK_STREAM`, which defines a stream socket.

Socket types used in BSD IPC include

- `SOCK-STREAM`—stream socket.
- `SOCK_DGRAM`—datagram socket.
- `SOCK_RAW`—raw socket.
- `SOCK_SEQPACKET`—sequenced packet socket

Index