HP 9000 Networking

Berkeley IPC Programmer's Guide

HEWLETT
PACKARD

Edition 2
E0792

98194-60531
Printed in U.S.A. 07/92

Preface

This manual describes the Berkeley Software Distribution Interprocess Communications (BSD IPC) set of programming development tools for interprocess communication.

The *Berkeley IPC Programmer's Guide* is the primary reference manual for programmers who write or maintain Berkeley IPC applications on HP 9000 computers. This manual should also be read by node managers before designing an HP 9000 network so that they have a clear understanding of the features provided by BSD IPC.

This manual is written for intermediate to advanced programmers and assumes that:

- Your node or system manager has brought up the network and has installed the HP 9000 product on your local host.
- You have asked your node manager for all the login names you may be associated with.
- You have asked your node manager what other hosts or nodes your HP 9000 product can communicate with.

This manual is organized as follows:

Chapter 1, "BSD IPC Concepts," provides an overview of the Client-Server Model.

Chapter 2, "BSD IPC Using Internet Stream Sockets," describes the steps involved in creating an Internet stream socket BSD IPC connection using the AF_INET address family.

Chapter 3, "Advanced Topics for Stream Sockets," explains socket options, synchronous I/O multiplexing with *select*, sending and receiving data asynchronously, nonblocking I/O, using shutdown, using read and write to make stream sockets transparent, and sending and receiving out of band data.

Chapter 4, "BSD IPC Using Internet Datagram Sockets," describes the steps involved in creating an Internet datagram socket BSD IPC connection.

Chapter 5, "Advanced Topics for Internet Datagram Sockets," includes information about default socket address, synchronous I/O multiplexing with *select* and broadcast addresses.

Chapter 6, "BSD IPC Using UNIX Domain Stream Sockets," describes the steps involved in creating a UNIX Domain stream socket BSD IPC connection between two processes executing on the same node.

Chapter 7, "BSD IPC Using UNIX Domain Datagram Sockets" describes the steps required to exchange data between AF_UNIX datagram sockets on the same node without establishing a connection.

Chapter 8, "Programming Hints," contains information about troubleshooting, port addresses, diagnostic utilities, Internet daemon, *inetd*, and system and library calls.

Appendix A, "Example Programs," contains sample programs found in the */usr/netdemo/socket* directory.

Appendix B, "Portability Issues," describes implementation differences between 4.3 BSD IPC and HP-UX IPC.

Contents

Figures

Tables

1 BSD IPC Concepts

Note The information contained in this manual applies to both the Series 300/400 and Series 600/700/800 HP 9000 computer systems. Any differences in installation, configuration, or operation are specifically noted.

This manual describes HP's implementation of the 4.3 Berkeley Software Distribution Interprocess Communication (BSD IPC) facilities. BSD IPC is a set of programming development tools for interprocess communication. HP's implementation of BSD IPC is a full set of sockets from the networking services originally developed by the University of California at Berkeley (UCB).

Note BSD IPC is a program development tool. Before you attempt to use BSD IPC, you may need to familiarize yourself with the C programming language and the HP-UX operating system. You could implement a BSD IPC application using FORTRAN or Pascal, but all library calls and include files are implemented in C.

The BSD IPC facility allows you to create distributed applications that pass data between programs (on the same computer or on separate computers on the network) without requiring an understanding of the many layers of networking protocols. This is accomplished by using a set of system calls. These system calls, when used in the correct sequence, allow you to create communication endpoints called **sockets** and transfer data between them.

This guide describes the steps involved in establishing and using BSD IPC connections. It also describes the protocols you must use and how the BSD IPC system calls interact. The details of each system call are described in the section 2 entries of the *Networking Reference* manual.

To understand the general model for BSD IPC, you need to understand what is meant by a **socket**, a **socket descriptor** and **binding**. Read the following definitions before you read about the Client-Server model.

socket	Sockets are communication endpoints. A pair of connected sockets provides an interface similar to that of HP-UX pipes. A socket is identified by a socket descriptor.
socket descriptor	A socket descriptor is an HP-UX file descriptor that references a socket instead of an ordinary file. Therefore, it can be used for reading, writing, or most standard file system calls after a BSD IPC connection is established. All BSD IPC functions use socket descriptors as arguments.
binding	Before a socket can be accessed across the network, it must be bound to an address. Binding makes the socket accessible to other sockets on the network by establishing its address. Binding is explained in more detail throughout this chapter.

How You Can Use BSD IPC

The best example of how BSD IPC can be used is the ARPA/Berkeley Services. The services use BSD IPC to communicate between remote hosts. Using the BSD IPC facility, 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.

The Client-Server Model

Typical BSD IPC applications consist of two separate application level 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 process. Once the server process accepts a client process's request and establishes a connection, full-duplex (two-way) communication can occur between the two sockets.

This set of conventions must be implemented by both processes. Depending upon the needs of your application, your implementation of the model can be symmetric or asymmetric. In a symmetrical application of the model, either process can be a server or a client. In an asymmetrical application of the model, there is a clearly defined server process and client process. An example of an asymmetrical application is the *ftp* service.

Creating a Connection: the Client-Server Model

The following figures illustrate 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.

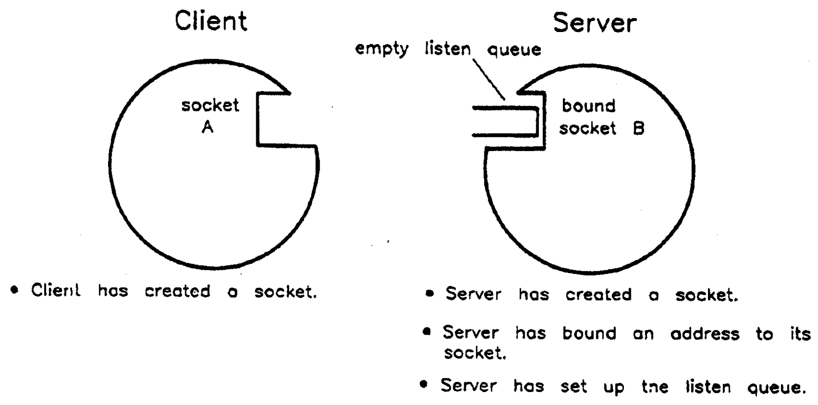
Client
socket A
Server
empty listen queue
bound socket B
• Client has created a socket.
• Server has created a socket.
• Server has bound an address to its socket.
• Server has set up the listen queue.

Figure 1-1. Client-Server in a Pre-Connection State

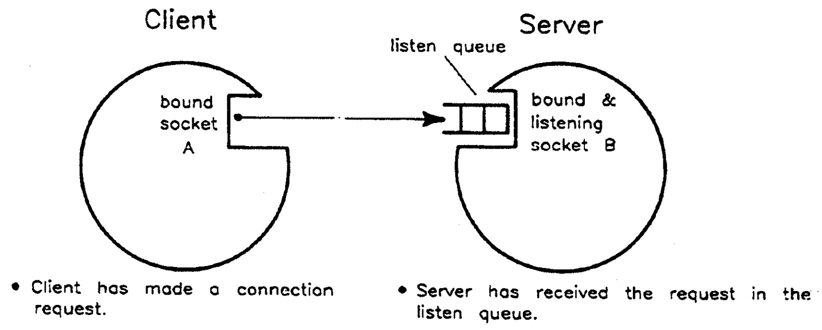
Client
bound socket A
Server
listen queue
bound & listening socket B
• Client has made a connection request.
• Server has received the request in the listen queue.

Figure 1-2. Client-Server at Time of Connection Request

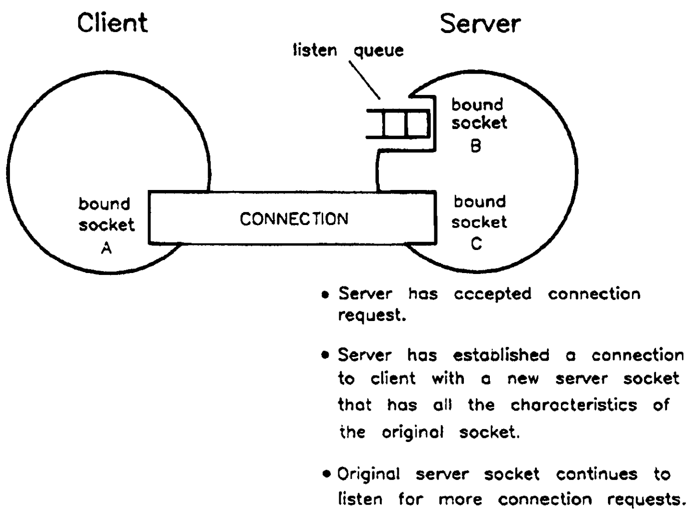
Client
Server
listen queue
bound socket B
bound socket A
CONNECTION
bound socket C
• Server has cccepted connection request.
• Server has established a connection to client with a new server socket that has all the characteristics of the original socket.
• Original server socket continues to listen for more connection requests.

Figure 1-3. Client-Server When Connection Is Established

A detailed description of the Client-Server model is discussed in chapter 2, "BSD IPC Using Internet Stream Sockets."

BSD IPC Library Routines

The library routines and system calls that you need to implement a BSD IPC application are described throughout this chapter. In addition, a complete list of all these routines and system calls is provided in the "Summary Tables for Library and System Calls" section of chapter 8, "Programming Hints."

The library routines are in the common "c" library named *libc.a*. Therefore, there is no need to specify any library name on the *cc* command line to use these library calls — *libc.a* is used automatically.

Key Terms and Concepts

The following list is meant to give you a basic understanding of the terms used to describe BSD IPC. Many of the terms have more detailed explanations within this manual in the places where the terms are used.

Communication Terms

packet	A message or data unit that is transmitted between communicating processes.
message	The data sent in one UDP packet.
channel	Communication path created by establishing a connection between sockets.
peer	The remote process with which a process communicates.

Addressing Terms

addressing	A means of labeling a socket so that it is distinguishable from other sockets on a host.
communication domain	A set of properties that describes the characteristics of processes communicating through sockets. The Internet (AF_INET) address family domain is supported. The UNIX Domain (AF_UNIX) address family domain is also supported, for local communication only.
address family	The address format used to interpret addresses specified in socket operations. The Internet address family (AF_INET) and the Berkeley UNIX Domain address family (AF_UNIX) are supported.
Internet address	A four-byte address that identifies a node on the network.

port	An address within a host that is used to differentiate between multiple sockets with the same Internet address. You can use port address values 1024 through 65535. (Port addresses 1 through 1023 are reserved for the super-user. Refer to page 8-3 for reserved ports.)
socket address	For the Internet address family (AF_INET), the socket address consists of the Internet address, port address and address family of a socket. The Internet and port address combination allows the network to locate a socket. For UNIX Domain (AF_UNIX), the socket address is the directory pathname bound to the socket.
binding	Associates a socket address with a socket. Once a socket address is bound, other sockets can connect to the socket and send data to or receive data from it.
association	A BSD IPC connection is defined by an association. An AF_INET association contains the (protocol, local address, local port, remote address, remote port)-tuple. An AF_UNIX association contains the (protocol, local address, peer address)-tuple. **Associations must be unique;** duplicate associations on the same host cannot exist. The tuple is created when the local and remote socket addresses are bound and connected. This means that the association is created in two steps, and there is a chance that two potential associations could be alike between steps. The host prevents duplicate associations by checking for uniqueness of the tuple at connection time, and reporting an error if the tuple is not unique.

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	Provides the underlying communication support for stream sockets. The Transmission Control Protocol (TCP) is used to implement reliable, sequenced, flow-controlled two-way communication based on byte streams similar to pipes. Refer to the *tcp(7p)* entry in the *HP-UX Reference Manual* for more information on TCP.
UDP	Provides the underlying communication support for datagram sockets. The User Datagram Protocol (UDP) is an unreliable protocol. A process receiving messages on a datagram socket could find messages are duplicated, out-of-sequence, or missing. Messages retain their record boundaries and are sent as individually addressed packets. There is no concept of a connection between the communicating sockets. Refer to the *udp(7p)* entry in the *HP-UX Reference Manual* for more information on UDP.

In addition, the UNIX Domain protocol may be used with AF_UNIX sockets for interprocess communication on the same node. Refer to the *unix(7p)* entry in the *HP-UX Reference Manual* for more information on the UNIX Domain protocol.

Using Socket Descriptors as File Descriptors

A socket descriptor is a special kind of HP-UX file descriptor; it can be used as though it were a file descriptor, but it references a socket instead of a file. System calls that use file descriptors (e.g. *read*, *write*, *select*) can be used with socket descriptors.

2

BSD IPC Using Internet Stream Sockets

This section describes the steps involved in creating an Internet stream socket BSD IPC connection using the AF_INET address family. If you want to use datagram sockets, skip to chapter 4, "BSD IPC Using Internet Datagram Sockets." If you want to use UNIX Domain sockets, skip to chapter 6, "BSD IPC Using UNIX Domain Stream Sockets."

As discussed in the "Protocols" section, Internet TCP stream sockets provide bidirectional, reliable, sequenced and unduplicated flow of data without record boundaries.

The following table lists the steps involved in creating and terminating a BSD IPC connection using stream sockets. Each step is described in more detail in the sections that follow the table.

Table 2-1. Building a BSD IPC Connection Using Internet Stream Sockets

Client Process Activity	System Call Used	Server Process Activity	System Call Used
create a socket	*socket()*	create a socket	*socket()*
bind a socket address (optional)	*bind()*	bind a socket address	*bind()*
		listen for incoming connection requests	*listen()*
request a connect-tion	*connect()*		
		accept connection	*accept()*
send data	*write()* or *send()*		
		receive data	*read()* or *recv()*
		send data	*write()* or *send()*
receive data	*read()* or *recv()*		
disconnect socket (optional)	*shutdown()* or *close()*	disconnect socket (optional)	*shutdown()* or *close()*

The following sections explain each of the activities mentioned in the previous table. The description of each activity specifies a system call and includes:

- What happens when the system call is used.
- When to make the call.
- What the parameters do.
- How the call interacts with other BSD IPC system calls.
- Where to find details on the system call.

The stream socket program examples are at the end of these descriptive sections. You can refer to the example code as you work through the descriptions.

Preparing Address Variables

Before you begin to create a connection, establish the correct variables and collect the information that you need to request a connection.

Your server process needs to:

- Declare socket address variables.
- Assign a wildcard address.
- Get the port address of the service that you want to provide.

Your client process needs to:

- Declare socket address variables.
- Get the remote host's Internet address.
- Get the port address for the service that you want to use.

These activities are described next. Refer to the program example at the end of this chapter to see how these activities work together.

Declaring Socket Address Variables

You need to declare a variable of type struct *sockaddr_in* to use for socket addresses.

For example, the following declarations are used in the example client program:

```
    struct sockaddr_in myaddr; /* for local socket address  */
    struct sockaddr_in peeraddr; /* for peer socket address  */
```

Sockaddr_in is a special case of *sockaddr* and is used with the AF_INET addressing domain. Both types are shown in this chapter, but *sockaddr_in* makes it easier to manipulate the Internet and port addresses. Some of the BSD IPC system calls are declared using a pointer to *sockaddr*, but you can also use a pointer to *sockaddr_in*.

The *sockaddr_in* address structure consists of the following fields:

short *sin_family*	Specifies the address family and should always be set to AF_INET.
u_short *sin_port*	Specifies the port address. Assign this field when you bind the port address for the socket or when you get a port address for a specific service.
struct inaddr *sin_addr*	Specifies the Internet address. Assign this field when you get the Internet address for the remote host.

The server process only needs an address for its own socket. Your client process may not need an address for its local socket.

Refer to the *inet(7f)* entry in the *HP-UX Reference Manual* for more information on *sockaddr_in*.

Getting the Remote Host's Internet Address

gethostbyname obtains the Internet address of the host and the length of that address (as the size of struct *in_addr*) from */etc/hosts.*

gethostbyname and its parameters are described in the following table.

INCLUDE FILES:	`#include <netdb.h>`

SYSTEM CALL:	`struct hostent *gethostbyname(name)`
	`char *name;`

Parameter	**Description of Contents**	**INPUT Value**
name	pointer to a valid host name (null-terminated string)	host name string

FUNCTION RESULT: pointer to struct hostent containing Internet address
NULL pointer (0) if failure occurs

EXAMPLE SYSTEM CALL:

```
#include <netdb.h>
struct hostent *hp; /* pointer to host info for remote host */
...
peeraddr.sin_family = AF_INET;
hp = gethostbyname (argv[1]);
peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
```

The *argv[1]* parameter is the host name specified in the client program command line.

Refer to the *gethostent(3n)* entry in the *HP-UX Reference Manual* for more information on *gethostbyname.*

Getting the Port Address for the Desired Service

When a server process is preparing to offer a service, it must get the port address for the service from */etc/services* so it can bind that address to its "listen" socket. If the service is not already in */etc/services*, you must add it.

When a client process needs to use a service that is offered by some server process, it must request a connection to that server process's "listening" socket. The client process must know the port address for that socket.

getservbyname obtains the port address of the specified service from */etc/services*.

getservbyname and its parameters are described in the following table.

INCLUDE FILES:

```
#include <netdb.h>
```

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
name	pointer to a valid service name	service name
proto	pointer to the protocol to be used	"tcp" or 0 if TCP is the only protocol for the service

FUNCTION RESULT: pointer to struct *servent* containing port address NULL pointer (0) if failure occurs

EXAMPLE SYSTEM CALL:

```
#include <netdb.h>
struct servent *sp; /* pointer to service info */
...
sp = getservbyname (''example'', ''tcp'');
peeraddr.sin_port = sp->s_port;
```

When to Get Server's Socket Address

Which Processes	When
server process	before binding the listen socket
client process	before client executes a connection request

Refer to the *getservent*(3n) entry in the *HP-UX Reference Manual* for more information on *getservbyname.*

Using a Wildcard Local Address

Wildcard addressing simplifies local address binding. When an address is assigned the value of INADDR_ANY, the host interprets the address as any valid address. This is useful for your server process when you are setting up the listen socket. It means that the server process does not have to look up its own Internet address. When INADDR_ANY is used as a host address, it also allows the server to listen to all network connections on the host. When a specific address is used in the bind, the server can only listen to that specific connection. Thus, INADDR_ANY is useful on a system in which multiple LAN cards are available, and messages for a given socket can come in on any of them.

For example, to bind a specific port address to a socket, but leave the local Internet address unspecified, the following source code could be used:

```
 #include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
 ...
struct sockaddr_in sin;
 ...
s = socket(AF_INET, SOCK_STREAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind (s, &sin, sizeof(sin));
```

Writing the Server Process

This section discusses the calls your server process must make to connect with and serve a client process.

Creating a Socket

The server process must call *socket* to create a communication endpoint.

Socket and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = socket(af, type, protocol)
int af, type, protocol;
```

Parameter	Description of Contents	INPUT Value
af	address family	AF_INET
type	socket type	SOCK_STREAM
protocol	underlying protocol to be used	0 (default) or value returned by *getprotobyname.*

FUNCTION RESULT: socket number (HP-UX file descriptor)
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
s = socket (AF_INET, SOCK_STREAM, 0);
```

The socket number returned is the socket descriptor for the newly created socket. This number is an HP-UX file descriptor and can be used for reading, writing or any standard file system calls after a BSD IPC connection is established. A socket descriptor is treated like a file descriptor for an open file.

When to Create Sockets

Which Processes	When
server process	before any other BSD IPC system calls

Refer to the *socket(2)* entry in the *HP-UX Reference Manual* for more information on *socket.*

Binding a Socket Address to the Server Process's Socket

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

The server process must bind a specific port address to this socket, which is used for listening. Otherwise, a client process would not know what port to connect to for the desired service.

Set up the address structure with a local address (as described in the "Preparing Address Variables" section) before you make a *bind* call. Use a wildcard address so your server process does not have to look up its own Internet address.

bind and its parameters are described in the following table.

INCLUDE FILES:	`#include <sys/types.h>`
	`#include <netinet/in.h>`
	`#include <sys/socket.h>`

SYSTEM CALL:	`bind (s, addr, addrlen)`
	`int s;`
	`struct sockaddr *addr;`
	`int addrlen;`

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket to be bound
addr	socket address	pointer to address to be bound to s
addrlen	length of socket address	size of struct *sockaddr_in*

FUNCTION RESULT:	0 if bind is successful –1 if failure occurs

EXAMPLE SYSTEM CALL:	`struct sockaddr_in myaddr;`
	`...`
	`bind (ls, &myaddr, sizeof(struct sockaddr_in));`

When to Bind Socket Addresses

Which Processes	When
server process	after socket is created and before any other BSD IPC system calls

Refer to the *bind(2)* entry in the *HP-UX Reference Manual* for more information on *bind.*

Setting the Server Up to Wait for Connection Requests

Once your server process has an address bound to it, it must call *listen* to set up a queue that accepts incoming connection requests. The server process then monitors the queue for requests (using *select(2)* or *accept*, which is described in "Accepting a Connection"). The server process cannot respond to a connection request until it has executed *listen.*

Listen and its parameters are described in the following table.

INCLUDE FILES:	none

SYSTEM CALL:	`listen(s, backlog)`
	`int s, backlog;`

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	server socket's descriptor
backlog	maximum number of connection requests in the queue at any time	size of queue (between 1 and 20)

FUNCTION RESULT:	0 if listen is successful –1 if failure occurs
EXAMPLE SYSTEM CALL:	`listen (ls, 5);`

Backlog is the number of unaccepted incoming connections allowed at a given time. Further incoming connection requests are rejected.

When to Set Server Up to Listen

Which Processes	When
server process	after socket is created and bound and before the server can respond to connection requests

Refer to the *listen(2)* entry in the *HP-UX Reference Manual* for more information on *listen.*

Accepting a Connection

The server process can accept any connection requests that enter its queue after it executes *listen*. *Accept* 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 address 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. (For information on nonblocking I/O, see the "Nonblocking I/O" section of chapter 3, "Advanced Topics for Stream Sockets.")

Accept and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = accept(ls,addr,addrlen)
int s;
int ls;
struct sockaddr *addr;
int *addrlen;
```

Parameter	Contents	INPUT Value	OUTPUT Value
s	socket descriptor of local socket	socket descriptor of server socket	unchanged
addr	socket address	pointer to address structure where address will be put	pointer to socket address of client socket that server's new socket is connected to
addrlen	length of address	pointer to the size of struct *sockaddr_in*	pointer to the actual length of address returned in addr

FUNCTION RESULT:	socket descriptor of new socket if accept is successful –1 if failure occurs

EXAMPLE SYSTEM CALL:

```
struct sockaddr_in peeraddr;
...
addrlen = sizeof(sockaddr_in);
s = accept (ls, &peeraddr, &addrlen);
```

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, 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.

When to Accept a Connection

Which Processes	When
server process	after executing the listen call

Refer to the *accept(2)* entry in the *HP-UX Reference Manual* for more information on *accept*.

Writing the Client Process

This section discusses the calls your client process must make to connect with and be served by a server process.

Creating a Socket

The client process must call *socket* to create a communication endpoint.

Socket and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = socket(af, type, protocol)
int af, type, protocol;
```

Parameter	Description of Contents	INPUT Value
af	address family	AF_INET
type	socket type	SOCK_STREAM
protocol	underlying protocol to be used	0 (default) or value returned by *getprotobyname*

FUNCTION RESULT: socket number (HP-UX file descriptor)
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
s = socket (AF_INET, SOCK_STREAM, 0);
```

The socket number returned is the socket descriptor for the newly created socket. This number is an HP-UX file descriptor and can be used for reading, writing or any standard file system calls after a BSD IPC connection is established. A socket descriptor is treated like a file descriptor for an open file.

When to Create Sockets

Which Processes	When
client process	before requesting a connection

Refer to the *socket(2)* entry in the *HP-UX Reference Manual* for more information on *socket.*

Requesting a Connection

Once the server process is listening for connection requests, the client process can request a connection with the *connect* call.

Connect and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
connect(s, addr, addrlen)
int s;
struct sockaddr *addr;
int addrlen;
```

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket requesting a connection
addr	pointer to the socket address	pointer to the socket address of the socket to which client wants to connect
addrlen	length of address	size of address structure pointed to by addr

FUNCTION RESULT:	0 if connect is successful –1 if failure occurs

EXAMPLE SYSTEM CALL:	`struct sockaddr_in peeraddr;`
	`...`
	`connect (s, &peeraddr, sizeof(struct sockaddr_in));`

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, see the "Nonblocking I/O" section of the "Advanced Topics for Stream Sockets" chapter.) When the connection is ready, the client process completes its *connect* call and the server process can complete its *accept* call.

Note The client process does not get feedback that the server process has completed the *accept* call. As soon as the *connect* call returns, the client process can send data. Local Internet and port addresses are bound when *connect* is executed if you have not already bound them yourself. These address values are chosen by the local host.

When to Request a Connection

Which Processes	When
client process	after socket is created and after server socket has a listening socket

Refer to the *connect*(2) entry in the *HP-UX Reference Manual* for more information on *connect.*

Sending and Receiving Data

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. Because the stream socket descriptors correspond to HP-UX file descriptors, you can use the *read* and *write* calls (in addition to *recv* and *send*) to pass data through a socket-terminated channel.

If you are considering the use of the *read* and *write* system calls instead of the *send* and *recv* calls described below, you should consider the following:

Advantage:	If you use *read* and *write* instead of *send* and *recv*, you can use a socket for *stdin* or *stdout.*
Disadvantage:	If you use *read* and write instead of *send* and *recv*, you cannot use the options specified with the *send* or *recv* *flags* parameter.

See the table called "Other System Calls" listed in chapter 8 "Programming Hints"for more information on which of these system calls are best for your application.

Sending Data

Send and its parameters are described in the following table.

INCLUDE FILES:	`#include <sys/types.h>`
	`#include <sys/socket.h>`

SYSTEM CALL:	`count = send(s,msg,len,flags)`
	`int s;`
	`char *msg;`
	`int len, flags;`

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket sending data
msg	pointer to data buffer	pointer to data to be sent
len	size of data buffer	size of msg
flags	settings for optional flags	0 or MSG_OOB

FUNCTION RESULT:	number of bytes actually sent –1 if failure occurs
EXAMPLE SYSTEM CALL:	`count = send (s, buf, 10, 0);`

Send blocks until the specified number of bytes have been queued to be sent, unless you are using nonblocking I/O. (For information on nonblocking I/O, see the "Nonblocking I/O" section of chapter 3, "Advanced Topics for Stream Sockets.")

When to Send Data

Which Processes	When
server or client process	after connection is established

Refer to the *send(2)* entry in the *HP-UX Reference Manual* for more information on *send.*

Receiving Data

Recv and its parameters are described in the following table.

INCLUDE FILES:	`#include <sys/types.h>`
	`#include <sys/socket.h>`

SYSTEM CALL:	`count = recv(s,buf,len,flags)`
	`int s;`
	`char *buf;`
	`int len, flags;`

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket receiving data
buf	pointer to data buffer	pointer to buffer that is to receive data
len	maximum number of bytes that should be received	size of data buffer
flags	settings for optional flags	0, MSG_OOB or MSG_PEEK

FUNCTION RESULT:	number of bytes actually received –1 if failure occurs
EXAMPLE SYSTEM CALL:	`count = recv(s, buf, 10, 0);`

Recv blocks until there is at least 1 byte of data to be received, unless you are using nonblocking I/O. (For information on nonblocking I/O, see the "Nonblocking I/O" section of the "Advanced Topics for Stream Sockets" chapter.) The host does not wait for *len* bytes to be available; if less than *len* bytes are available, that number of bytes are 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*.

Flag Options

The *flag* options are:

- 0 for no options.
- MSG_OOB for out of band data.
- MSG_PEEK for a nondestructive read.

Use the MSG_OOB option if you want to receive out of band data. Refer to the "Sending and Receiving Out of Band Data" section of chapter 3,"Advanced Topics for Stream Sockets," for more information.

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

When to Receive Data

Which Processes	When
server or client process	after connection is established

Refer to the *recv(2)* entry in the *HP-UX Reference Manual* for more information on *recv*.

Closing a Socket

In most applications, you do not have to worry about cleaning up your sockets. When you exit your program and your process terminates, the sockets are closed for you.

If you need to close a socket while your program is still running, use the *close* system call. For example, you may have a daemon process that uses *fork* to create the server process. The daemon process creates the BSD IPC connection and then passes the socket descriptor to the server. You then have more than one process with the same socket descriptor. The daemon process should do a *close* of the socket descriptor to avoid keeping the socket open once the server is through with it. Because the server performs the work, the daemon does not use the socket after the *fork*.

Close decrements the file descriptor reference count. Once this occurs, the calling process can no longer use that file descriptor.

When the last *close* is executed on a socket descriptor, any unsent data are sent before the socket is closed. Any unreceived data are lost. This delay in closing the socket can be controlled by the socket option SO_LINGER. Refer to the "Socket Options" section of the "Advanced Topics for Stream Sockets" chapter for information on the SO_LINGER option.

For syntax and details on *close*, refer to the *close(2)* entry in the *HP-UX Reference Manual.*

Additional options for closing sockets are discussed in the "Using Shutdown" section of the "Advanced Topics for Stream Sockets" chapter.

Example Using Internet Stream Sockets

Note These programs are provided as examples only of stream socket usage and are not Hewlett-Packard supported products.

These program examples demonstrate how to set up and use Internet stream sockets. These sample programs can be found in the */usr/netdemo/socket* directory. The client program is intended to run in conjunction with the server program. The client program requests a service called *example* from the server program.

The server process receives requests from the remote client process, handles the request and returns the results to the client process. Note that the server:

- Uses the wildcard address for the listen socket.
- Uses the *ntohs* address conversion call to show how to port to a host that requires it.
- Uses the SO_LINGER option for a graceful disconnect. The SO_LINGER option is discussed in the "Socket Options" section of the "Advanced Topics for Stream Sockets" chapter.

The client process creates a connection, sends requests to the server process and receives the results from the server process. Note that the client:

- Uses shutdown, which is discussed in the "Advanced Topics for Stream Sockets" chapter, to indicate that it is done sending requests.
- Uses *getsockname* to see what socket address was assigned to the local socket by the host.
- Uses the *ntohs* address conversion call to show how to port to a host that requires it.

Before you run the example programs, make the following entry in the two host's */etc/services* files:

```
example 22375/tcp
```

The source code for these two programs follows.

```
/*
 *                      S E R V . T C P
 *
 *      This is an example program that demonstrates the use of
 *      stream sockets as a BSD IPC mechanism.  This contains the server,
 *      and is intended to operate in conjunction with the client
 *      program found in client.tcp.  Together, these two programs
 *      demonstrate many of the features of sockets, as well as good
 *      conventions for using these features.
 *
 *      This program provides a service called  example .  In order for
 *      it to function, an entry for it needs to exist in the
 *      /etc/services file.  The port address for this service can be
 *      any port number that is likely to be unused, such as 22375,
 *      for example.  The host on which the client will be running
 *      must also have the same entry (same port number) in its
 *      /etc/services file.
 *
 */

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

int s;                          /* connected socket descriptor */
int ls;                         /* listen socket descriptor */

struct hostent *hp;             /* pointer to host info for remote host */
struct servent *sp;             /* pointer to service information */

long timevar;                   /* contains time returned by time() */
char *ctime();                  /* declare time formatting routine */

struct linger linger = {1,1};               /* allow a lingering, graceful close; */
                                /* used when setting SO_LINGER */

struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in peeraddr_in; /* for peer socket address */
```

```
/*
 *                                      M A I N
 *
 *      This routine starts the server.  It forks, leaving the child
 *      to do all the work, so it does not have to be run in the
 *      background.  It sets up the listen socket, and for each incoming
 *      connection, it forks a child process to process the data.  It
 *      will loop forever, until killed by a signal.
 */
main(argc, argv)
int argc;
char *argv[];
{
        int addrlen;
                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));
                /* Set up address structure for the listen socket. */
        myaddr_in.sin_family = AF_INET;
                /* The server should listen on the wildcard address,
                 * rather than its own Internet address.  This is
                 * generally good practice for servers, because on
                 * systems which are connected to more than one
                 * network at once will be able to have one server
                 * listening on all networks at once.  Even when the
                 * host is connected to only one network, this is good
                 * practice, because it makes the server program more
                 * portable.
                 */
        myaddr_in.sin_addr.s_addr = INADDR_ANY;
                /* Find the information for the  example  server
                 * in order to get the needed port number.
                 */
        sp = getservbyname ("example",  tcp );
        if (sp == NULL) {
                fprintf(stderr,  %s: host not found  ,
                                argv[0]);
                exit(1);
        }
        myaddr_in.sin_port = sp->s_port;

                /* Create the listen socket. */
        ls = socket (AF_INET, SOCK_STREAM, 0);
        if (ls == -1) {
                perror(argv[0]);
                fprintf(stderr,  %s: unable to create socket\n , argv[0]);
                exit(1);
        }
        /* Bind the listen address to the socket. */
if (bind(ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to bind address\n , argv[0]);
        exit(1);
}
        /* Initiate the listen on the socket so remote users
         * can connect.  The listen backlog is set to 5. 20
         * is the currently supported maximum.
         */
if (listen(ls, 5) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to listen on socket\n , argv[0]);
        exit(1);
}

        /* Now, all the initialization of the server is
         * complete, and any user errors will have already
         * been detected.  Now we can fork the daemon and
         * return to the user.  We need to do a setpgrp
         * so that the daemon will no longer be associated
         * with the user's control terminal.  This is done
         * before the fork, so that the child will not be
         * a process group leader.  Otherwise, if the child
         * were to open a terminal, it would become associated
         * with that terminal as its control terminal.  It is
         * always best for the parent to do the setpgrp.
         */
setpgrp();

switch (fork()) {
case -1:                /* Unable to fork, for some reason. */
        perror(argv[0]);
        fprintf(stderr,  %s: unable to fork daemon\n , argv[0]);
        exit(1);

case 0:                 /* The child process (daemon) comes here. */
                /* Close stdin and stderr so that they will not
                 * be kept open.  Stdout is assumed to have been
                 * redirected to some logging file, or /dev/null.
                 * From now on, the daemon will not report any
                 * error messages.  This daemon will loop forever,
                 * waiting for connections and forking a child
                 * server to handle each one.
                 */
        fclose(stdin);
        fclose(stderr);
```

```
                /* Set SIGCLD to SIG_IGN, in order to prevent
                 * the accumulation of zombies as each child
                 * terminates.  This means the daemon does not
                 * have to make wait calls to clean them up.
                 */
        signal(SIGCLD, SIG_IGN);
        for(;;) {
                        /* Note that addrlen is passed as a pointer
                         * so that the accept call can return the
                         * size of the returned address.
                         */
                addrlen = sizeof(struct sockaddr_in);
                        /* This call will block until a new
                         * connection arrives.  Then, it will
                         * return the address of the connecting
                         * peer, and a new socket descriptor, s,
                         * for that connection.
                         */
                s = accept(ls, &peeraddr_in, &addrlen);
                if ( s == -1) exit(1);
                switch (fork()) {
                case -1:        /* Can't fork, just continue. */
                        exit(1);
                case 0:         /* Child process comes here. */
                        server();
                        exit(0);
                default:        /* Daemon process comes here. */
                                /* The daemon needs to remember
                                 * to close the new accept socket
                                 * after forking the child.  This
                                 * prevents the daemon from running
                                 * out of file descriptors.  It
                                 * also means that when the server
                                 * closes the socket, that it will
                                 * allow the socket to be destroyed
                                 * since it will be the last close.
                                 */
                        close(s);
                }

        }

    default:                /* Parent process comes here. */
        exit(0);
    }
}
```

```
        /* Bind the listen address to the socket. */
if (bind(ls, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to bind address\n , argv[0]);
        exit(1);
}
        /* Initiate the listen on the socket so remote users
         * can connect.  The listen backlog is set to 5. 20
         * is the currently supported maximum.
         */
if (listen(ls, 5) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to listen on socket\n , argv[0]);
        exit(1);
}

        /* Now, all the initialization of the server is
         * complete, and any user errors will have already
         * been detected.  Now we can fork the daemon and
         * return to the user.  We need to do a setpgrp
         * so that the daemon will no longer be associated
         * with the user's control terminal.  This is done
         * before the fork, so that the child will not be
         * a process group leader.  Otherwise, if the child
         * were to open a terminal, it would become associated
         * with that terminal as its control terminal.  It is
         * always best for the parent to do the setpgrp.
         */
setpgrp();

switch (fork()) {
case -1:                /* Unable to fork, for some reason. */
        perror(argv[0]);
        fprintf(stderr,  %s: unable to fork daemon\n , argv[0]);
        exit(1);

case 0:                 /* The child process (daemon) comes here. */
                /* Close stdin and stderr so that they will not
                 * be kept open.  Stdout is assumed to have been
                 * redirected to some logging file, or /dev/null.
                 * From now on, the daemon will not report any
                 * error messages.  This daemon will loop forever,
                 * waiting for connections and forking a child
                 * server to handle each one.
                 */
        fclose(stdin);
        fclose(stderr);
```

```
                /* Set SIGCLD to SIG_IGN, in order to prevent
                 * the accumulation of zombies as each child
                 * terminates.  This means the daemon does not
                 * have to make wait calls to clean them up.
                 */
        signal(SIGCLD, SIG_IGN);
        for(;;) {
                        /* Note that addrlen is passed as a pointer
                         * so that the accept call can return the
                         * size of the returned address.
                         */
                addrlen = sizeof(struct sockaddr_in);
                        /* This call will block until a new
                         * connection arrives.  Then, it will
                         * return the address of the connecting
                         * peer, and a new socket descriptor, s,
                         * for that connection.
                         */
                s = accept(ls, &peeraddr_in, &addrlen);
                if ( s == -1) exit(1);
                switch (fork()) {
                case -1:        /* Can't fork, just continue. */
                        exit(1);
                case 0:         /* Child process comes here. */
                        server();
                        exit(0);
                default:        /* Daemon process comes here. */
                                /* The daemon needs to remember
                                 * to close the new accept socket
                                 * after forking the child.  This
                                 * prevents the daemon from running
                                 * out of file descriptors.  It
                                 * also means that when the server
                                 * closes the socket, that it will
                                 * allow the socket to be destroyed
                                 * since it will be the last close.
                                 */
                        close(s);
                }

        }

    default:                /* Parent process comes here. */
        exit(0);
    }
}
```

```
/*
 *                              S E R V E R
 *
 *      This is the actual server routine that the daemon forks to
 *      handle each individual connection.  Its purpose is to receive
 *      the request packets from the remote client, process them,
 *      and return the results to the client.  It will also write some
 *      logging information to stdout.
 *
 */
server()
{
        int reqcnt = 0;         /* keeps count of number of requests */
        char buf[10];           /* This example uses 10 byte messages. */
        char *inet_ntoa();
        char *hostname;         /* points to the remote host's name string */
        int len, len1;

                /* Close the listen socket inherited from the daemon. */
        close(ls);

                /* Look up the host information for the remote host
                 * that we have connected with.  Its Internet address
                 * was returned by the accept call, in the main
                 * daemon loop above.
                 */
        hp = gethostbyaddr ((char *) &peeraddr_in.sin_addr,
                                sizeof (struct in_addr),
                                peeraddr_in.sin_family);

        if (hp == NULL) {
                        /* The information is unavailable for the remote
                         * host.  Just format its Internet address to be
                         * printed out in the logging information.  The
                         * address will be shown in  Internet dot format .
                         */
                hostname = inet_ntoa(peeraddr_in.sin_addr);
        } else {
                hostname = hp->h_name;  /* point to host's name */
        }
                /* Log a startup message. */
        time (&timevar);
```

```
                /* The port number must be converted first to host byte
                 * order before printing.  On most hosts, this is not
                 * necessary, but the ntohs() call is included here so
                 * that this program could easily be ported to a host
                 * that does require it.
                 */
        printf("Startup from %s port %u at %s",
                hostname, ntohs(peeraddr_in.sin_port), ctime(&timevar));

                /* Set the socket for a lingering, graceful close.
                 * Since linger was set to 1 above, this will cause
                 * a final close of this socket to wait until all of the
                 * data sent on it has been received by the remote host.
                 */
        if (setsockopt(s, SOL_SOCKET, SO_LINGER, &linger,
                                        sizeof(linger)) == -1) {
errout:         printf("Connection with %s aborted on error\n", hostname);
                exit(1);
        }

                /* Go into a loop, receiving requests from the remote
                 * client.  After the client has sent the last request,
                 * it will do a shutdown for sending, which will cause
                 * an end-of-file condition to appear on this end of the
                 * connection.  After all of the client's requests have
                 * been received, the next recv call will return zero
                 * bytes, signalling an end-of-file condition.  This is
                 * how the server will know that no more requests will
                 * follow, and the loop will be exited.
                 */
        while (len = recv(s, buf, 10, 0)) {
                if (len == -1) goto errout; /* error from recv */
                        /* The reason this while loop exists is that there
                         * is a remote possibility of the above recv returning
                         * less than 10 bytes.  This is because a recv returns
                         * as soon as there is some data, and will not wait for
                         * all of the requested data to arrive.  Since 10 bytes
                         * is relatively small compared to the allowed TCP
                         * packet sizes, a partial receive is unlikely.  If
                         * this example had used 2048 bytes requests instead,
                         * a partial receive would be far more likely.
                         * This loop will keep receiving until all 10 bytes
                         * have been received, thus guaranteeing that the
                         * next recv at the top of the loop will start at
                         * the begining of the next request.
                         */
```

```
		while (len < 10) {
			len1 = recv(s, &buf[len], 10-len, 0);
			if (len1 == -1) goto errout;
			len += len1;
		}
			/* Increment the request count. */
		reqcnt++;
			/* This sleep simulates the processing of the
			 * request that a real server might do.
			 */
		sleep(1);
			/* Send a response back to the client. */
		if (send(s, buf, 10, 0) != 10) goto errout;
	}

		/* The loop has terminated, because there are no
		 * more requests to be serviced.  As mentioned above,
		 * this close will block until all of the sent replies
		 * have been received by the remote host.  The reason
		 * for lingering on the close is so that the server will
		 * have a better idea of when the remote has picked up
		 * all of the data.  This will allow the start and finish
		 * times printed in the log file to reflect more accurately
		 * the length of time this connection was used.
		 */
	close(s);

		/* Log a finishing message. */
	time (&timevar);
		/* The port number must be converted first to host byte
		 * order before printing.  On most hosts, this is not
		 * necessary, but the ntohs() call is included here so
		 * that this program could easily be ported to a host
		 * that does require it.
		 */
	printf("Completed %s port %u, %d requests, at %s\n",
		hostname, ntohs(peeraddr_in.sin_port), reqcnt, ctime(&timevar));
}
```

```
/*
 *                   C L I E N T . T C P
 *
 *      This is an example program that demonstrates the use of stream
 *      sockets as a BSD IPC mechanism.  This contains the client, and is
 *      intended to operate in conjunction with the server program found
 *      in serv.tcp.  Together, these two programs demonstrate many of the
 *      features of sockets, as well as good conventions for using these
 *      features.
 *
 *      This program requests a service called  example .  In order for it
 *      to function, an entry for it needs to exist in the /etc/services
 *      file.  The port address for this service can be any port number
 *      that is likely to be unused, such as 22375, for example.  The host
 *      on which the server will be running must also have the same entry
 *      (same port number) in its /etc/services file.
 *
 */

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

int s;                          /* connected socket descriptor */

struct hostent *hp;             /* pointer to host info for remote host */
struct servent *sp;             /* pointer to service information */

long timevar;                   /* contains time returned by time() */
char *ctime();                  /* declare time formatting routine */

struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in peeraddr_in; /* for peer socket address */

/*
 *                   M A I N
 *
 *      This routine is the client which request service from the remote
 *       example server .  It creates a connection, sends a number of
 *      requests, shuts down the connection in one direction to signal the
 *      server about the end of data, and then receives all of the responses.
 *      Status will be written to stdout.
 *
 *      The name of the system to which the requests will be sent is given
 *      as a parameter to the command.
 */
```

```
main(argc, argv)
int argc;
char *argv[];
{
        int addrlen, i, j;

                /* This example uses 10 byte messages. */
        char buf[10];

        if (argc != 2) {
                fprintf(stderr,  Usage:  %s <remote host>\n , argv[0]);
                exit(1);
        }
                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&peeraddr_in, 0, sizeof(struct sockaddr_in));

                /* Set up the peer address to which we will connect. */
        peeraddr_in.sin_family = AF_INET;
                /* Get the host information for the hostname that the
                 * user passed in.
                 */
        hp = gethostbyname (argv[1]);
                /* argv[1] is the host name. */
        if (hp == NULL) {
                fprintf(stderr,  %s: %s not found in /etc/hosts\n ,
                                argv[0], argv[1]);
                exit(1);
        }
        peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
                /* Find the information for the  example  server
                 * in order to get the needed port number.
                 */
        sp = getservbyname ("example",  tcp );
        if (sp == NULL) {
                fprintf(stderr,  %s: example not found in /etc/services\n ,
                                argv[0]);
                exit(1);
        }
        peeraddr_in.sin_port = sp->s_port;

                /* Create the socket. */
        s = socket (AF_INET, SOCK_STREAM, 0);
        if (s == -1) {
                perror(argv[0]);
                fprintf(stderr,  %s: unable to create socket\n , argv[0]);
                exit(1);
        }
        /* Try to connect to the remote server at the address
         * which was just built into peeraddr.
         */
if (connect(s, &peeraddr_in, sizeof(struct sockaddr_in)) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to connect to remote\n , argv[0]);
        exit(1);
}
        /* Since the connect call assigns a random address
         * to the local end of this connection, let's use
         * getsockname to see what it assigned.  Note that
         * addrlen needs to be passed in as a pointer,
         * because getsockname returns the actual length
         * of the address.
         */
addrlen = sizeof(struct sockaddr_in);
if (getsockname(s, &myaddr_in, &addrlen) == -1) {
        perror(argv[0]);
        fprintf(stderr,  %s: unable to read socket address\n , argv[0]);
        exit(1);
}

        /* Print out a startup message for the user. */
time(&timevar);
        /* The port number must be converted first to host byte
         * order before printing.  On most hosts, this is not
         * necessary, but the ntohs() call is included here so
         * that this program could easily be ported to a host
         * that does require it.
         */
printf("Connected to %s on port %u at %s",
                argv[1], ntohs(myaddr_in.sin_port), ctime(&timevar));


        /* This sleep simulates any preliminary processing
         * that a real client might do here.
         */
sleep(5);
```

```
                /* Send out all the requests to the remote server.
                 * In this case, five are sent, but any random number
                 * could be used.  Note that the first four bytes of
                 * buf are set up to contain the request number.  This
                 * number will be returned in the reply from the server.
                 */

                /* CAUTION: If you increase the number of requests sent
                 * or the size of the requests, you should be
                 * aware that you could encounter a deadlock
                 * situation.  Both the client's and server's
                 * sockets can only queue a limited amount of
                 * data on their receive queues.
                 */
        for (i=1; i<=5; i++) {
                *buf = i;
                if (send(s, buf, 10, 0) != 10) {
                        fprintf(stderr,  %s: Connection aborted on error   ,
                                        argv[0]);
                        fprintf(stderr,  on send number %d\n , i);
                        exit(1);
                }
        }

                /* Now, shutdown the connection for further sends.
                 * This will cause the server to receive an end-of-file
                 * condition after it has received all the requests that
                 * have just been sent, indicating that we will not be
                 * sending any further requests.
                 */
        if (shutdown(s, 1) == -1) {
                perror(argv[0]);
                fprintf(stderr,  %s: unable to shutdown socket\n , argv[0]);
                exit(1);
        }

                /* Now, start receiving all of the replys from the server.
                 * This loop will terminate when the recv returns zero,
                 * which is an end-of-file condition.  This will happen
                 * after the server has sent all of its replies, and closed
                 * its end of the connection.
                 */
        while (i = recv(s, buf, 10, 0)) {
                if (i == -1) {
errout:                 perror(argv[0]);
                        fprintf(stderr,  %s: error reading result\n , argv[0]);
                        exit(1);
                }
```

```
                        /* The reason this while loop exists is that there
                         * is a remote possibility of the above recv returning
                         * less than 10 bytes.  This is because a recv returns
                         * as soon as there is some data, and will not wait for
                         * all of the requested data to arrive.  Since 10 bytes
                         * is relatively small compared to the allowed TCP
                         * packet sizes, a partial receive is unlikely.  If
                         * this example had used 2048 bytes requests instead,
                         * a partial receive would be far more likely.
                         * This loop will keep receiving until all 10 bytes
                         * have been received, thus guaranteeing that the
                         * next recv at the top of the loop will start at
                         * the begining of the next reply.
                         */
                while (i < 10) {
                        j = recv(s, &buf[i], 10-i, 0);
                        if (j == -1) goto errout;
                        i += j;
                }
                        /* Print out message indicating the identity of
                         * this reply.
                         */
                printf("Received result number %d\n", *(int *)buf);
        }

                /* Print message indicating completion of task. */

        time(&timevar);
        printf("All done at %s", ctime(&timevar));
}
```

3

Advanced Topics for Stream Sockets

This chapter explains the following:

- Socket options.
- Synchronous I/O multiplexing with *select*.
- Sending and receiving data asynchronously.
- Nonblocking I/O.
- Using shutdown.
- Using *read* and *write* to make stream sockets transparent.
- Sending and receiving out-of-band data.

Socket Options

The operation of sockets is controlled by socket level options. The following options are supported for Internet stream sockets:

- SO_REUSEADDR.
- SO_KEEPALIVE.
- SO_DONTROUTE.
- SO_SNDBUF.

- SO_RCVBUF.
- SO_LINGER.
- SO_USELOOPBACK.
- SO_OOBINLINE.
- SO_SNDLOWAT.
- SO_RCVLOWAT.
- SO_SNDTIMEO.
- SO_RCVTIMEO.
- SO_TYPE.
- SO_ERROR.

All of these options may be used on either AF_INET or AF_UNIX sockets; the following, however, are really INET specific in their function and will not change UNIX socket behavior.

- SO_KEEPALIVE.
- SO_REUSEADDR.
- SO_DONTROUTE.
- SO_USELOOPBACK.
- SO_OOBINLINE.
- SO_SNDTIMEO.
- SO_RCVTIMEO.
- SO_BROADCAST.

In addition, the SO_DEBUG option is supported for compatibility only; it has no functionality.

The next section discusses how to set socket options and get the current value of a socket option. Following those discussions is a description of each available option. Refer to chapter 6 for a description of the SO_BROADCAST option.

Getting and Setting Socket Options

The socket options are defined in the *sys/socket.h* file. 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.

Setsockopt and its parameters are described in the following table:

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
s	socket descriptor	socket descriptor for which options are to be set
level	protocol level	SOL_SOCKET
optname	name of option	supported option name
optval	pointer to option input value	Must be at least size of (int). Holds either value to be set or boolean flag
optlen	length of optval	size of optval

FUNCTION RESULT:	0 if *setsockopt* is successful –1 if failure occurs
EXAMPLE SYSTEM CALL:	See the description of the SO_REUSEADDR option for an example.

Refer to the *getsockopt(2)* entry in the *HP-UX Reference Manual* for more information on *setsockopt.*

getsockopt and its parameters are described in the following table:

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	**Contents**	**INPUT Value**	**OUTPUT Value**
s	socket descriptor	socket descriptor for which option values are to be returned	unchanged
level	protocol level	SOL_SOCKET	unchanged
optname	name of option	supported option name	unchanged
optval	pointer to current value of option	pointer to buffer where option's current value is to be returned	pointer to buffer that contains current option value
optlen	pointer to length of optval	pointer to maximum number of bytes to be returned by optval	pointer to actual size of optval returned

FUNCTION RESULT: 0 if the option is set

If *getsockopt* fails for any reason, the function returns -1, and the option is not returned. An error code is stored in errno.

EXAMPLE SYSTEM CALL:

```
len = sizeof (optval))
getsockopt(s, SOL_SOCKET, SO_REUSEADDR, &optval, &len;)
```

optval may never be zero. It must always point to data sent with the socket option and must always be at least the size of an integer.

The following socket options set socket parameter values. *optval* is an integer containing the new value:

- SO_SNDBUF.
- SO_RCVBUF.
- SO_SNDLOWAT.
- SO_RCVLOWAT.
- SO_SNDTIMEO.
- SO_RCVTIMEO.

The following socket options toggle socket behavior. *optval* is an integer containing a boolean flag for the behavior (1 = on, 0 = off):

- SO_KEEPALIVE.
- SO_DEBUG.
- SO_DONTROUTE.
- SO_USELOOPBACK.
- SO_REUSEADDR.
- SO_OOBINLINE.

The SO_LINGER option is a combination. It sets a linger value, and also toggles linger behavior on and off. In previous releases SO_DONTLINGER was supported. In the 8.0 release, toggling SO_LINGER off is equivalent in function. For SO_LINGER, *optval* points to a struct *linger*, defined in */usr/include/sys/socket.h*. The linger structure contains an integer boolean flag to toggle behavior on/off, and an integer linger value.

Refer to the *getsockopt(2)* entry in the *HP-UX Reference Manual* for more information on *getsockopt*.

SO_REUSEADDR

This option is AF_INET socket-specific.

SO_REUSEADDR enables you to restart a daemon which was killed or terminated.

This option modifies the rules used by *bind* to validate local addresses, but it does not violate the uniqueness requirements of an association. SO_REUSEADDR modifies the *bind* rules only when a wildcard Internet Protocol (IP) address is used in combination with a particular protocol port. The host still checks at connection time to be sure any other sockets with the same local address and local port do not have the same remote address and remote port. *Connect* fails if the uniqueness requirement is violated.

The following example shows the SO_REUSEADDR option's use.

Suppose that a network daemon server is listening on a specific port: port 2000. If you executed `netstat an`, part of the output would resemble:

```
Active connections (including  servers)
Proto Recv-Q Send-Q  Local Address     Foreign Address   (state)
tcp      0      0 *.2000          *.*             LISTEN
```

Network Daemon Server Listening at Port 2000

When the network daemon accepts a connection request, the accepted socket will bind to port 2000 and to the address where the daemon is running (e.g. `192.6.250.100`).

If you then executed `netstat an`, the output would resemble:

```
Active connections (including servers)
Proto Recv-Q Send-Q  Local Address     Foreign Address   (state)
tcp      0      0 192.6.250.100.2000 192.6.250.101.4000 ESTABLISHED
tcp      0      0 *.2000          *.*             LISTEN
```

New Connection Established, Daemon Server Still Listening

Here the network daemon has established a connection to the client (`192.6.250.101.4000`) with a new server socket. The original network daemon server continues to listen for more connection requests.

If the listening network daemon process is killed, attempts to restart the daemon fail if SO_REUSEADDR is not set. The restart fails because the daemon attempts to bind to port 2000 and a wildcard IP address (e.g. `*.2000`). The wildcard address matches the address of the established connection (`192.6.250.100`), so the *bind* aborts to avoid duplicate socket naming.

When SO_REUSEADDR is set, *bind* ignores the wildcard match, so the network daemon can be restarted.

An example usage of this option is:

```
int optval = 1;
 setsockopt (s, SOL_SOCKET, SO_REUSEADDR, &optval, sizeof(optval));
bind (s, &sin, sizeof(sin));
```

SO_KEEPALIVE

This option is AF_INET socket-specific.

This option enables the periodic transmission of messages on a connected socket. This occurs at the transport level and does not require any work in your application programs.

If the peer socket does not respond to these messages, the connection is considered broken. The next time one of your processes attempts to use a connection that is considered broken, the process is notified (with a SIGPIPE signal if you are trying to send, or an end-of-file condition if you are trying to receive) that the connection is broken.

SO_DONTROUTE

This option is AF_INET socket-specific.

SO_DONTROUTE indicates that outgoing messages should bypass the standard routing facilities. Instead, messages are directed to the appropriate network interface according to the network portion of the destination address.

SO_SNDBUF

SO_SNDBUF changes the send socket buffer size. Increasing the send socket buffer size allows a user to send more data before the user's application will block, waiting for more buffer space.

Note Increasing buffer size to send larger portions of data before the application blocks **may** increase throughput, but the best method of tuning performance is to experiment with various buffer sizes.

You can increase a stream socket's buffer size at any time, but decrease it only prior to establishing a connection.

The maximum buffer size for stream sockets is 58254 bytes.

Example:

```
int result;
int buffsize = 10,000;
result = setsockopt(s, SOL_SOCKET, SO_SNDBUF, &buffsize, sizeof(buffsize));
```

SO_RCVBUF

SO_RCVBUF changes the receive socket buffer size.

You can increase a stream socket's buffer size at any time, but decrease it only prior to establishing a connection.

The maximum buffer size for stream sockets is 58254 bytes.

Example:

```
int result;
int buffsize = 10,000;
result = setsockopt(s, SOL_SOCKET, SO_RCVBUF, &buffsize, sizeof(buffsize));
```

Table 3-1. Summary Information for Changing Socket Buffer Size

Socket Type (Protocol)	When Buffer Size Increase Allowed	When Buffer Size Decrease Allowed	Maximum Buffer Size
stream (TCP)	at any time	only prior to establishing a connection	58254 bytes

SO_LINGER

SO_LINGER controls the actions taken when a *close* is executed on a socket that has unsent data.

This option can be cleared by toggling. The default is off.

The linger timeout interval is set with a parameter in the *setsockopt* call. The only useful values are zero and nonzero:

- If l_onoff is zero, close returns immediately, but any unsent data is transmitted (after close returns).
- If l_onoff is nonzero and l_linger is zero, close returns immediately, and any unsent data is discarded.
- If l_onoff is nonzero and l_linger is nonzero, clsoe does not return until all unsent data is transmitted (or the connection is closed by the remote system).

In the default case (SO_LINGER is off), *close* is not blocked. The socket itself, however, goes through graceful disconnect, and no data is lost.

Example:

```
int result;
struct linger linger;
linger.l_onoff = 1;  /*0 = off (l_linger ignored), nonzero = on      */
linger.l_linger =1;  /*0 = discard data, nonzero = wait for data sent */
result = setsockopt(s, SOL_SOCKET, SO_LINGER, &linger, sizeof(linger));
```

Table 3-2. Summary of Linger Options on Close

Socket Option	Option Set	Linger Interval	Graceful Close	Hard Close	Wait for Close	Does Not Wait for Close
SO_LINGER	off	don't care	x			x
SO_LINGER	on	zero		x		x
SO_LINGER	on	nonzero	x		x	

SO_USELOOPBACK

This option is not applicable to UNIX Domain sockets.

SO_USELOOPBACK directs the network layer (IP) of networking code to use the local loopback address when sending data from this socket. Use this option only when all data sent will also be received locally.

SO_OOBINLINE

This option is not applicable to UNIX Domain sockets.

This option enables receipt of out-of-band data inline. Normally, OOB data is extracted from the data stream and must be read with the MSG_OOB flag specified in the *recv()* call. When SO_OOBINLINE is specified, OOB data arriving at that socket remains inline and can be read without MSG_OOB specified.

In both cases, a normal *read()* or *recv()* which would read past the OOB mark will halt at the mark, instead leaving the OOB byte the next byte to be read.

SO_SNDLOWAT

This option allows the user to set or fetch the low water mark for the socket's send socket buffer.

At present, this option is not used. It is supported in anticipation of future use.

SO_RCVLOWAT

This option allows the user to set or fetch the low water mark for the socket's receive socket buffer.

At present, this option is not used. It is supported in anticipation of future use.

SO_SNDTIMEO

This option allows the user to set or fetch the timeout value for a socket's send socket buffer.

At present, this option is not used. It is supported in anticipation of future use.

RCVTIMEO

This option allows the user to set or fetch the timeout value for the socket's receive socket buffer.

At present, this option is not used. It is supported in anticipation of future use.

SO_TYPE

This option is used to return the socket type (e.g., stream, datagram, etc.). Use this option only with the *getsockopt* system call.

SO_ERROR

This option is used to get and clear any error that has occurred on the socket. Use this option only with the *getsockopt* system call.

SO_BROADCAST

This option is not supported for UNIX Domain sockets. Setting this option allows the user to send datagrams to a broadcast address. A broadcast address is defined as an Internet address whose local address portion is all 1s.

Synchronous I/O Multiplexing with Select

The *select* system call can be used with sockets to provide a synchronous multiplexing mechanism. The system call has several parameters which govern its behavior. If you specify a zero pointer for the timeout parameter, *select* will block until one or more of the specified socket descriptors is ready. If timeout is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete.

A *select* of a socket descriptor for **reading** is useful on:

- A connected socket, because it determines when data have arrived and are ready to be read without blocking; use the FIONREAD parameter to the *ioctl* system call to determine exactly how much data are available.
- A listening socket, because it determines when you can accept a connection without blocking.
- Any socket to detect if an error has occurred on the socket.

A *select* of a socket descriptor for **writing** is useful on:

- A connecting socket, because it determines when a connection is complete.
- A connected socket, because it determines when more data can be sent without blocking. This implies that at least one byte can be sent; there is no way, however, to determine exactly how many bytes can be sent.
- Any socket to detect if an error has occurred on the socket.

select for exceptional conditions will return true for Berkeley sockets if out-of-band data is available to be read. *Select* will always return true for sockets which are no longer capable of being used (e.g. if a *close* or *shutdown* system call has been executed against them).

select is used in the same way as in other applications. Refer to the *select(2)* entry in the *HP-UX Reference Manual* for information on how to use *select*. For an asynchronous alternative to *select*, see the next section, "Sending and Receiving Data Asynchronously."

Example:

The following example illustrates the *select* system call. Since it is possible for a process to have more than 32 open file descriptors, the bit masks used by *select* are interpreted as arrays of integers. The header file *sys/types.h* contains some useful macros to be used with the *select()* system call, some of which are reproduced below.

```
/*
 *   These macros are used with select().  select() uses bit masks of file
 *   descriptors in long integers.  These macros manipulate such bit fields
 *   (the file system macros use chars).  FD_SETSIZE may be defined by
 *   the user, but must be = u.u_highestfd + 1.  Since the absolute limit on
 *   the number of per process open files is 2048, FD_SETSIZE must
 *   be large enough to accommodate this many file descriptors.
 *   Unless the user has this many files opened, FD_SETSIZE should be
 *   redefined to a smaller number.
*/

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)))

do_select(s)
int s;                      /* socket to select on, initialized */
{
   struct fd_set read_mask, write_mask;  /* bit masks */
   int nfds;                             /* number to select on */
   int nfd;                              /* number found */

   for (;;) {                            /* for example... */
        FD_ZERO(&read_mask);             /* select will overwrite on return */
        FD_ZERO(&write_mask);
        FD_SET(s, &read_mask);           /* we care only about the socket */
        FD_SET(s, &write_mask);
        nfds = s+1;                      /* select descriptors 0 through s */
        nfd = select(nfds, &read_mask, &write_mask, (int *) 0,
                    (struct timeval *) 0);/* will block */
        if (nfd == -1) {
                perror(  select: unexpected condition  );
                exit(1);
        }
        if (FD_ISSET(s, &read_mask))
                do_read(s);              /* something to read on socket s */
                                         /* fall through as maybe more to do */
        if (FD_ISSET(s, &write_mask))
                do_write(s);            /* space to write on socket s */
    }
}
```

Sending and Receiving Data Asynchronously

Asynchronous sockets allow a user program to receive an SIGIO signal when the state of the socket changes. This state change can occur, for example, when new data arrives. Currently the user would have to issue a *select* system call in order to determine if data were available. If other processing is required of the user program, the need to call *select* can complicate an application by forcing the user to implement some form of polling, whereby all sockets are checked periodically. Asynchronous sockets allow the user to separate socket processing from other processing, eliminating polling altogether. *Select* may still be required to determine exactly why the signal is being delivered, or to which socket the signal applies.

Generation of the SIGIO signal is protocol dependent. It mimics the semantics of *select* in the sense that the signal is generated whenever *select* returns true. It is generally accepted that connectionless protocols deliver the signal whenever a new packet arrives. For connection oriented protocols, the signal is also delivered when connections are established or broken, as well as when additional outgoing buffer space becomes available. Be warned that these assertions are guidelines only; any signal handler should be robust enough to handle signals in unexpected situations.

The delivery of the SIGIO signal is dependent upon two things. First the socket state must be set as asynchronous; this is done using the FIOASYNC flag of the *ioctl* system call. Second, the process group (pgrp) associated with the socket must be set; this is done using the SIOCSPGRP flag of *ioctl*. The sign value of the pgrp can lead to various signals being delivered. Specifically, if the pgrp is negative, this implies that a signal should be delivered to the process whose PID is the absolute value of the pgrp. If the pgrp is positive, a signal should be delivered to the process group identified by the absolute value of the pgrp.

Any application that chooses to use asynchronous sockets must explicitly activate the described mechanism. The SIGIO signal is a "safe" signal in the sense that if a process is unprepared to handle it, the default action is to ignore it. Thus any existing applications are immune to spurious signal delivery.

Notification that out-of-band data has been received is also done asynchronously; see the section "Sending and Receiving Out-of-band Data" in this chapter for more details.

Examples:

The following example sets up an asynchronous SOCK_STREAM listen socket. This is typical of an application that needs to be notified when connection requests arrive.

```
int ls;                     /* SOCK_STREAM listen socket initialized */
int flag = 1;               /* for ioctl, to turn on async */
int iohndlr();              /* the function which handles the SIGIO */

signal (SIGIO, iohndlr);      /* set up the handler */

if (ioctl (ls, FIOASYNC, &flag) == -1) {
        perror ("can't set async on socket");
        exit(1);
}
flag = -getpid();           /* process group negative == deliver to process */
if (ioctl (ls, SIOCSPGRP, &flag) == -1) {
        perror ("can't set pgrp");
        exit(1);
}
          /* signal can come any time now */
```

The following example illustrates the use of process group notification. Note that the real utility of this feature is to allow multiple processes to receive the signal, which is not illustrated here. For example, the socket could be of type SOCK_DGRAM; a signal here can be interpreted as the arrival of a service-request packet. Multiple identical servers could be set up, and the first available one could receive and process the packet.

```
int flag = 1;           /* ioctl to turn on async */
int iohndlr();
signal (SIGIO, iohndlr);

setpgrp();              /* set my processes' process group */
if (ioctl (s, FIOASYNC, &flag) == -1) {
      perror ("can't set async on socket");
      exit(1);
}
flag = getpid();    /* process group + == deliver to every process in group */
if (ioctl (s, SIOCSPGRP, &flag) == -1) {
      perror ("can't set pgrp");
      exit(1);
}

         /* signal can come any time now */
```

For more information, see Appendix A, which contains a complete program showing the client and server code using asynchronous sockets.

Nonblocking I/O

Sockets are created in blocking mode I/O by default. You can specify that a socket be put in nonblocking mode by using the ioctl system call with the FIOSNBIO request.

An example usage of this call is:

```
#include <sys/ioctl.h>
     ...
     ioctl(s, FIOSNBIO, &arg);
```

arg is a pointer to *int*:

- When *int* equals 0, the socket is changed to blocking mode.
- When *int* equals 1, the socket is changed to nonblocking mode.

If a socket is in nonblocking mode, the following calls are affected:

accept	If no connection requests are present, *accept* returns immediately with the EWOULDBLOCK error.
connect	If the connection cannot be completed immediately, *connect* returns with the EINPROGRESS error.
recv	If no data are available to be received, *recv* returns the value –1 and the EWOULDBLOCK error. This is also true for read.
send	If there is no available buffer space for the data to be transmitted, *send* returns the value -1 and the EWOULDBLOCK error. This is also true for write.

The O_NDELAY flag for *fcntl(2)* is also supported. If you use this flag and there are no data available to be received on a *recv*, *recvfrom*, *recvmsg*, or *read* call, the call returns immediately with the value of 0. If you use the O_NONBLOCK flag, the call returns immediately with the value of -1 and the EAGAIN error. This is the same as returning an end-of-file condition. This is also true for *send*, *sendto*, *sendmsg*, and *write* if there is not enough buffer space to complete the send.

Note The O_NDELAY and O_NONBLOCK flags have precedence over the FIOSNBIO flag. Setting both the O_NDELAY and O_NONBLOCK flags is not allowed.

Using Shutdown

When your program is done reading or writing on a particular socket connection, you can use *shutdown* to bring down a part of the connection. (See the example programs for stream sockets.)

When one process uses *shutdown* on a socket descriptor, all other processes with the same socket descriptor are affected. *Shutdown* causes all or part of a full-duplex connection on the specified socket to be disabled. When *shutdown* is executed, the specified socket is marked unable to send or receive, according to the value of *how*:

- If *how* = 0, the specified socket can no longer receive data. The connection is not completely down until both sides have done a *shutdown* or a *close*.
- If *how* = 1, *shutdown* starts a graceful disconnect by attempting to send any unsent data before preventing further sending. *Shutdown* sends an end-of-file condition to the peer, indicating that there are no more data to be sent.

  Once both *shutdown(s, 0)* and *shutdown(s, 1)* have been executed on the same socket descriptor, the only valid operation on the socket at this point is a *close*.
- If *how* = 2, the specified socket can no longer send or receive data. The only valid operation on the socket is a *close*. This has the same effect as executing *shutdown(s, 0)* and *shutdown(s, 1)* on the same socket descriptor.

If you use *close* on a socket, *close* pays attention to the SO_LINGER option, but *shutdown(s, 2)* does not. With *close*, the socket descriptor is deallocated and the last process using the socket destroys it.

Shutdown and its parameters are described in the following table.

INCLUDE FILES:	none
SYSTEM CALL:	`shutdown(s,how)` `int s, how;`

Parameter	Description of Contents	INPUT Value
s	socket descriptor	socket descriptor of socket to be shut down
how	number that indicates the type of shutdown	0, 1 or 2

FUNCTION RESULT:	0 if shutdown is successful –1 if failure occurs
EXAMPLE SYSTEM CALL:	`shutdown (s, 1);`

When to Shut Down a Socket

Which Processes	When
server or client process	(optionally) after the process has sent all messages and wants to indicate that it is done sending

Refer to the *shutdown(2)* entry in the *HP-UX Reference Manual* for more information on *shutdown.*

Using Read and Write to Make Stream Sockets Transparent

An example application of *read* and *write* with stream sockets is to fork a command with a socket descriptor as *stdout.* The peer process can *read* input from the command. The command can be any command and does not have to know that *stdout* is a socket. It might use *printf,* which results in the use of *write.* Thus, the stream sockets are transparent.

Sending and Receiving Out-of-band Data

This option is not supported for UNIX Domain (AF_UNIX) sockets.

If an abnormal condition occurs when a process is in the middle of sending a long stream of data, it is useful to be able to alert the other process with an urgent message. The TCP stream socket implementation includes an out-of-band data facility. Out-of-band data uses a **logically** independent transmission channel associated with a pair of connected stream sockets. TCP supports the reliable delivery of only one out-of-band message at a time. The message can be a maximum of one byte long.

Out-of-band data arrive at the destination node in sequence and in stream, but are delivered independently of normal data. If the receiver has enabled the signalling of out-of-band data via the SIOCSPGRP socket *ioctl* (see *socket(7)* in the *HP-UX Reference Manual*), then a SIGURG is delivered when out-of-band data arrive. If the receiver is selecting for exceptional conditions on the receiving socket, it will return true to signal the arrival of out-of-band data. The receiving process can read the out-of-band message and take the appropriate action based on the message contents. A logical mark is placed in the normal data stream to indicate the point at which the out-of-band data were sent, so that data before the message can be handled differently (if necessary) from data following the message.

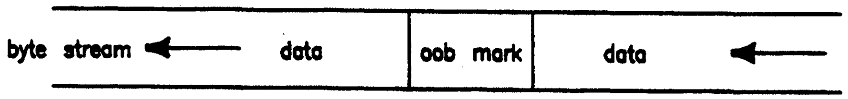

Figure 3-1. Data Stream with Out-of-Band Marker

For a program to know when out-of-band data are available to be received, you may arrange the program to catch the SIGURG signal as follows:

```
struct sigvec vec;
     int onurg();
     int pid, s;

     /*
     ** arrange for onurg() to be called when SIGURG is received:
     */
     vec.sv_handler = onurg;
     vec.sv_mask = 0;
     vec.sv_onstack = 0;
     if (sigvector(SIGURG, &vec, (struct sigvec *) 0) < 0) {
        perror("sigvector(SIGURG)");
     }
```

Onurg() is a routine that handles out-of-band data in the client program.

In addition, the socket's process group must be set, as shown below. The kernel will not send the signal to the process (or process group) unless this is done, even though the signal handler has been enabled.

```
/*
     ** arrange for the current process to receive SIGURG
     ** when the socket s has urgent data:
     */
     pid = getpid();
     if (ioctl(s, SIOCSPGRP, (char *) &pid) < 0) {
         perror("ioctl(SIOCSPGRP)");
     }
/*
     ** If a process needs to be notified, it should be
     ** pid = -getpgrp();
     */
```

Refer to the *socket(7)* entry in the *HP-UX Reference Manual* for more details.

If the server process is sending data to the client process, and a problem occurs, the server can send an out-of-band data byte by executing a *send* with the MSG_OOB flag set. This sends the out-of-band data and a SIGURG signal to the receiving process.

```
send(sd, &msg, 1, MSG_OOB)
```

When a SIGURG signal is received, *onurg* is called. *Onurg* receives the out-of-band data byte with the MSG_OOB flag set on a *recv* call.

It is possible that the out-of-band byte has not arrived when the SIGURG signal arrives. *recv* **never blocks** on a receive of out-of-band data, so the client may need to repeat the *recv* call until the out-of-band byte arrives. *Recv* will return EINVAL if the out-of-band data is not available.

Generally, out-of-band data byte is stored independently from normal data stream. If, however, the OOB_INLINE socket option has been turned on for this socket, the out-of-band data will remain inline and must be used without the MSG_OOB flag set on a *recv()* call.

You cannot read **past** the out-of-band pointer location in one *recv* call. If you request more data than the amount queued on the socket before the out-of-band pointer, then *recv* will return only the data up to the out-of-band pointer. However, once you read past the out-of-band pointer location with subsequent *recv* calls, the out-of-band byte can no longer be read.

Usually the out-of-band data message indicates that all data currently in the stream can be flushed. This involves moving the stream pointer with successive *recv* calls, to the location of the out-of-band data pointer.

The *ioctl* request SIOCATMARK informs you, as you receive data from the stream, when the stream pointer has reached the out-of-band pointer. If *ioctl* returns a 0, the next *recv* provides data sent by the server prior to transmission of the out-of-band data. *Ioctl* returns a 1 when the stream pointer reaches the out-of-band byte pointer. The next *recv* provides data sent by the server after the out-of-band message.

The following code segment illustrates how the SIOCATMARK request can be used in a SIGURG interrupt handler.

```
/*  s is the socket with urgent data  */

     onurg()
     {
         int atmark;
         char mark;
         char flush [100];

         while (1) {
             /*
             ** check whether we have read the stream
             ** up to the OOB mark yet
+            */
             if (ioctl(s, SIOCATMARK, &atmark) < 0) {
                 /* if the ioctl failed */
                 perror("ioctl(SIOCATMARK)");
                 return;
             }
             if (atmark) {
                 /* we have read the stream up to the OOB mark */
                 break;
             }
             /*
             ** read the stream data preceding the mark,
             ** only to throw it away
             */
             if (read(s, flush, sizeof(flush)) <= 0) {
                 /* if the read failed */
                 return;
             }
         }

/*
```

```
        ** receive the OOB byte
        */
        recv(s, &mark, 1, MSG_OOB);

        printf("received %c OOB\n", mark);
        return;
}
```

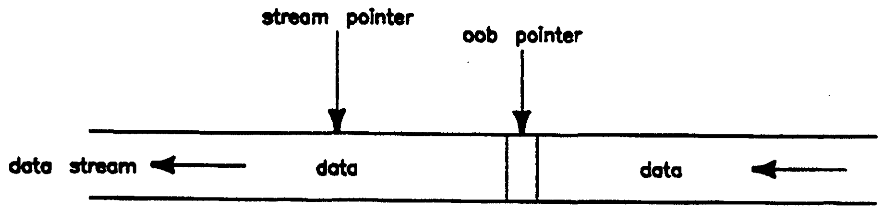

Figure 3-2. Before Flushing Stream

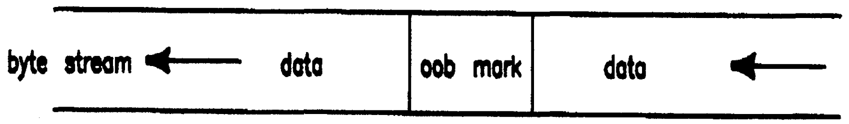

Figure 3-3. After Flushing Stream

Note This completes the discussion of stream sockets. If you do not plan to use datagram sockets, skip to the "Programming Hints" chapter.

4

BSD IPC Using Internet Datagram Sockets

As discussed in the "Protocols" section of chapter 1, Internet UDP datagram sockets provide bidirectional flow of data with record boundaries preserved. However, there is no guarantee that messages are reliably delivered. If a message is delivered, there is no guarantee that it is in sequence and unduplicated, but the data in the message are guaranteed to be intact.

Datagram sockets allow you to send and receive messages **without establishing a connection.** Each message includes a destination address. Processes involved in data transfer are not required to have a client-server relationship; the processes can be symmetrical.

Unlike stream sockets, datagram sockets allow you to send to many destinations from one socket, and receive from many sources with one socket. There is no two-process model, although a two-process model is the simplest case of a more general multiprocess model. The terms server and client are used in this section only in the application sense. There is no difference in the calls that must be made by the processes involved in the data transfer.

For example, you might have a name server process that receives host names from clients all over a network. That server process can send host name and Internet address combinations back to the clients. This can all be done with one UDP socket.

The simplest two-process case is used in this chapter to describe BSD IPC using datagram sockets.

The following table lists the steps required to exchange data between datagram sockets. Each step is described in more detail in the sections that follow the table.

Table 4-1. Setting Up for Data Transfer Using Datagram Sockets

Client Process Activity	System Call Used	Server Process Activity	System Call Used
create a socket	*socket()*	create a socket	*socket()*
bind a socket address	*bind()*	bind a socket address	*bind()*
send message	*sendto()* or *sendmsg()*		
		receive message	*recvfrom()* or *recvmsg()*
		send message	*sendto()* or *sendmsg()*
receive message	*recvfrom*() or *recvmsg()*		

The following sections discuss each of the activities mentioned in the previous table. The description of each activity specifies a system call and includes:

- what happens when the system call is used.
- when to make the system call.
- what the parameters do.
- how the call interacts with other BSD IPC system calls.
- where to find details on the system call.

The datagram socket program examples are at the end of these descriptive sections. You can refer to them as you work through the descriptions.

Preparing Address Variables

Before your client process can make a request of the server process, you must establish the correct variables and collect the information that you need about the server process and the service provided.

The server process needs to:

- Declare socket address variables.
- Assign a wildcard address.
- Get the port address of the service that you want to provide.

The client process needs to:

- Declare socket address variables.
- Get the remote server's Internet address.
- Get the port address for the service that you want to use.

These activities are described next. In addition, refer to the program example at the end of this chapter to see how these activities work together.

Declaring Socket Address Variables

You need to declare a variable of type struct *sockaddr_in* to use for the local socket address for both processes.

For example, the following declarations are used in the example client program:

```
struct sockaddr_in myaddr; /* for local socket address  */
struct sockaddr_in servaddr; /* for server socket address  */
```

Sockaddr_in is a special case of *sockaddr* and is used with the AF_INET addressing domain. Both types are shown in this chapter, but *sockaddr_in* makes it easier to manipulate the Internet and port addresses. Some of the BSD IPC system calls are declared using a pointer to *sockaddr*, but you can also use a pointer to *sockaddr_in*.

The *sockaddr_in* address structure consists of the following fields:

short *sin_family*	Specifies the address family and should always be set to AF_INET.
u_short *sin_port*	Specifies the port address. Assign this field when you bind the port address for the socket or when you get a port address for a specific service.
struct in_addr *sin_addr*	Specifies the Internet address. Assign this field when you get the Internet address for the remote host.

The server process must bind the port address of the service to its own socket and establish an address structure to store the clients' addresses when they are received with *recvfrom*.

The client process does not have to bind a port address for its local socket; the host binds one automatically if one is not already bound.

Refer to the *inet(7F)* entry in the *HP-UX Reference Pages* for more information on *sockaddr_in*.

Getting the Remote Host's Network Address

The client process can use *gethostbyname* to obtain the Internet address of the host and the length of that address (as the size of struct *inaddr*) from */etc/hosts, NIS,* or *BIND.*

Gethostbyname and its parameters are described in the following table.

INCLUDE FILES:	`#include <netdb.h>`

SYSTEM CALL:	`struct hostent *gethostbyname(name)`
	`char *name;`

Parameter	Description of Contents	INPUT Value
name	pointer to a valid node name (null-terminated string)	host name

FUNCTION RESULT:	pointer to struct *hostent* containing Internet address NULL pointer (0) if failure occurs

EXAMPLE SYSTEM CALL:

```
#include <netdb.h>
struct hostent *hp; /* point to host info for name server host */
...
servaddr.sin_family = AF_INET;
hp = gethostbyname (argv[1]);
servaddr.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
```

The *argv[1]* parameter is the host name specified in the client program command line.

Refer to the *gethostent(3N)* entry in the *HP-UX Reference Pages* for more information on *gethostbyname*.

Getting the Port Address for the Desired Service

When a client process needs to use a service that is offered by some server process, it must send a message to the server's socket. The client process must know the port address for that socket. If the service is not in */etc/services*, you must add it.

Getservbyname obtains the port address of the specified service from */etc/services*.

Getservbyname and its parameters are described in the following table.

INCLUDE FILES:

```
#include <netdb.h>
```

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
name	pointer to a valid service name	service name
proto	pointer to the protocol to be used	"udp" or 0 if UDP is the only protocol for the service

FUNCTION RESULT: pointer to struct servent containing port address
NULL pointer (0) if failure occurs

EXAMPLE SYSTEM CALL:

```
#include <netdb.h>
struct servent *sp; /* pointer to service info */
...
sp = getservbyname (''example'', ''udp'');
servaddr.sin_port = sp->s_port;
```

When to Get Server's Socket Address

Which Processes	When
server process	before binding
client process	before client requests the service from the host

Refer to the *getservent(3N)* entry in the *HP-UX Reference Pages* for more information on *getservbyname*.

Using a Wildcard Local Address

Wildcard addressing simplifies local address binding. When an address is assigned the value of INADDR_ANY, the host interprets the address as any valid address.

This means that the server process can receive on a wildcard address and does not have to look up its own Internet address. For example, to bind a specific port address to a socket, but leave the local Internet address unspecified, the following source code could be used:

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
  ...
struct sockaddr_in sin;
  ...
s = socket(AF_INET, SOCK_DGRAM, 0);
sin.sin_family = AF_INET;
sin.sin_addr.s_addr = INADDR_ANY;
sin.sin_port = MYPORT;
bind (s, &sin, sizeof(sin));
```

Writing the Server and Client Processes

This section discusses the calls your server and client processes must make.

Creating Sockets

Both processes must call *socket* to create communication endpoints.

Socket and its parameters are described in the following table.

INCLUDE FILES:	`#include <sys/types.h>`
	`#include <sys/socket.h>`

SYSTEM CALL:	`s = socket(af, type, protocol)`
	`int s, af, type, protocol;`

Parameter	Description of Contents	INPUT Value
af	address family	AF_INET
type	socket type	SOCK_DGRAM
protocol	underlying protocol to be used	0 (default) or value returned by *getprotobyname*

FUNCTION RESULT: socket number (HP-UX file descriptor)
–1 if failure occurs

EXAMPLE SYSTEM CALL:
```
ls = socket (AF_INET, SOCK_DGRAM, 0);
```

The socket number returned is the socket descriptor for the newly created socket. This number is an HP-UX file descriptor and can be used for reading, writing or any standard file system calls. A socket descriptor is treated like a file descriptor for an open file.

Note To use *write(2)* with a datagram socket, you must declare a default address. Refer to the "Specifying a Default Socket Address" section of the "Advanced Topics for Internet Datagram Sockets" chapter for more information.

When to Create Sockets

Which Processes	When
server or client process	before any other BSD IPC system calls

Refer to the *socket(2)* entry in the *HP-UX Reference Pages* for more information on *socket.*

Binding Socket Addresses to Datagram Sockets

After each process has created a socket, it must call *bind* to bind a socket address. Until an address is bound, other processes have no way to reference it.

The server process must bind a specific port address to its socket. Otherwise, a client process would not know what port to send requests to for the desired service.

The client process can let the local host bind its local port address. The client does not need to know its own port address, and if the server process needs to send a reply to the client's request, the server can find out the client's port address when it receives with *recvfrom.*

Set up the address structure with a local address (as described in the "Preparing Address Variables" section) before you make a bind call. Use the wildcard address so your processes do not have to look up their own Internet addresses.

bind and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
bind (s, addr, addrlen)
int s;
struct sockaddr *addr;
int addrlen;
```

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket to be bound
addr	socket address	pointer to address to be bound to s
addrlen	length of socket address	size of struct *sockaddr_in_address*

FUNCTION RESULT:	0 if bind is successful –1 if failure occurs
EXAMPLE SYSTEM CALL:	see below

```
struct sockaddr_in myaddr;
...
bind (s, &myaddr, sizeof(struct sockaddr_in));
```

When to Bind Socket Addresses

Which Processes	When
client and server process	after socket is created and before any other BSD IPC system calls

Refer to the *bind(2)* entry in the *HP-UX Reference Pages* for more information on *bind*.

Sending and Receiving Messages

The *sendto* and *recvfrom* (or *sendmsg* and *recvmsg*) system calls are usually used to transmit and receive messages. They are described in the next sections.

Sending Messages

Use *sendto* or *sendmsg* to send messages. *sendmsg* allows the send data to be gathered from several buffers.

If you have declared a default address (as described in the "Advanced Topics for Internet Datagram Sockets" chapter, "Specifying a Default Socket Address" section), you can use *send*, *sendto*, or *sendmsg* to send messages. If you use *sendto* or *sendmsg* in this special case, be sure you specify 0 as the address value, or an error will occur.

Send is described in the "Sending Data" section in the "BSD IPC Using Internet Stream Sockets" chapter of this manual and in the *send(2)* entry in the *HP-UX Reference Pages*.

Sendto and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket sending message
msg	pointer to data buffer	pointer to data to be sent
len	size of data buffer	size of msg
flags	settings for optional flags	0 (no options are currently supported)
to	address of recipient socket	pointer to the socket address that message should be sent to
tolen	size of to	length of address structure that to points to

FUNCTION RESULT:	Number of bytes actually sent –1 in the event of an error

EXAMPLE SYSTEM CALL:

```
count = sendto(s,argv[2],strlen(argv[2]),0,servaddr,sizeof(struct sockaddr_in));
```

The largest message size for this implementation is 9216 bytes.

You should not count on receiving error messages when using datagram sockets. The protocol is unreliable, meaning that messages may or may not reach their destination. However, if a message reaches its destination, the contents of the message are guaranteed to be intact.

If you need reliable message transfer, you must build it into your application programs or resend a message if the expected response does not occur.

When to Send Data

Which Processes	When
client or server process	after sockets are bound

Refer to the *send(2)* entry in the *HP-UX Reference Pages* for more information on *sendto* and *sendmsg*.

Receiving Messages

Use *recvfrom* or *recvmsg* to receive messages. *recvmsg* allows the read data to be scattered into buffers.

Recv can also be used if you do not need to know what socket sent the message. However, if you want to send a response to the message, you must know where it came from. Except for the extra information returned by *recvfrom* and *recvmsg*, the three calls are identical.

Recv is described in the "BSD IPC Using Internet Stream Sockets" chapter, "Receiving Data" section, and in the *recv(2)* entry in the *HP-UX Reference Pages.*

Recvfrom and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
count = recvfrom(s,buf,len,flags,from,fromlen)
int s;
char *buf;
int len, flags;
struct sockaddr *from;
int *fromlen;
```

Parameter	Contents	INPUT Value	OUTPUT Value
s	socket descriptor of local socket	socket descriptor of socket receiving message	unchanged
buf	pointer to data buffer	pointer to buffer that is to receive data	pointer to received data
len	maximum number of bytes that should be received	size of data buffer	unchanged
flags	settings for optional flags	0 or MSG_PEEK	unchanged
from	address of socket that sent message	pointer to address structure, not used for input	pointer to socket address of socket that sent the message
fromlen	pointer to the size of from	pointer to size of from	pointer to the actual size of address returned

FUNCTION RESULT: Number of bytes actually received
–1 if an error occurs

EXAMPLE SYSTEM CALL:

```
addrlen = sizeof(sockaddr_in);
...
count = recvfrom(s, buffer, BUFFERSIZE, 0, clientaddr, &addrlen);
```

recvfrom blocks until there is a message to be received.

No more than *len* bytes of data are returned. The entire message is read in one *recvfrom*, *recvmsg*, *recv* or *read* operation. If the message is too long for the allocated buffer, the excess data are discarded. Because only one message can be returned in a *recvfrom* call, if a second message is in the queue, it is not affected. Therefore, the best technique is to receive as much as possible on each call.

The host does not wait for *len* bytes to be available; if less than *len* bytes are available, that number of bytes are returned.

Flag Options

The *flag* options are:

- 0 for no options.
- MSG_PEEK for a nondestructive read.

Use the MSG_PEEK option to preview an incoming message. If this option is set on a *recvfrom*, any message returned remains in the data buffer as though it had not been read yet. The next *recvfrom* will return the **same message**.

When to Receive Data

Which Processes	When
client or server process	after sockets are bound

Refer to the *recv(2)* entry in the *HP-UX Reference Pages* for more information on *recvfrom* and *recvmsg*.

Closing a Socket

In most applications, you do not have to worry about cleaning up your sockets. When you exit your program and your process terminates, the sockets are closed for you.

If you need to close a socket while your program is still running, use the *close* HP-UX file system call.

You may have more than one process with the same socket descriptor if the process with the socket descriptor executes a *fork*. *Close* decrements the file descriptor count and the calling process can no longer use that file descriptor.

When the last *close* is executed on a socket, any unsent messages are sent and the socket is closed. Then the socket is destroyed and can no longer be used.

For syntax and details on *close*, refer to the *close(2)* entry in the *HP-UX Reference Manual.*

Example Using Datagram Sockets

Note These programs are provided as examples only of datagram socket usage and are not Hewlett-Packard supported products.

These program examples demonstrate how to set up and use datagram sockets. These sample programs can be found in the */usr/netdemo/socket* directory. The client program is intended to run in conjunction with the server program.

This example implements a simple name server. The server process receives requests from the client process. It determines the Internet address of the specified host and sends that address to the client process. If the specified host's Internet address is unknown, the server process returns an address of all 1s.

The client process requests the Internet address of a host and receives the results from the server process.

Note Before you run the example programs, make the following entry in the two hosts' */etc/services* files:

```
example 22375/udp
```

The source code for these two programs follows.

```
/*                      S E R V . U D P
 *
 *      This is an example program that demonstrates the use of
 *      datagram sockets as an BSD IPC mechanism.  This contains the server,
 *      and is intended to operate in conjunction with the client
 *      program found in client.udp.  Together, these two programs
 *      demonstrate many of the features of sockets, as well as good
 *      conventions for using these features.  NOTE:  This example
 *      is valid only if the /etc/hosts file is being used to lookup host names.
 *
 *      This program provides a service called ''example''.  It is an
 *      example of a simple name server.  In order for
 *      it to function, an entry for it needs to exist in the
 *      /etc/services file.  The port address for this service can be
 *      any port number that is likely to be unused, such as 22375,
 *      for example.  The host on which the client will be running
 *      must also have the same entry (same port number) in its
 *      /etc/services file.
 *
 */

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

int s;                          /* socket descriptor */

#define BUFFERSIZE      1024    /* maximum size of packets to be received */
int cc;                         /* contains the number of bytes read */
char buffer[BUFFERSIZE];        /* buffer for packets to be read into */

struct hostent *hp;             /* pointer to host info for requested host */
struct servent *sp;             /* pointer to service information */

struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in clientaddr_in;       /* for client's socket address */
struct in_addr reqaddr;         /* for requested host's address */

#define ADDRNOTFOUND    0xffffffff      /* return address for unfound host */

/*
 *                      M A I N
 *
 *      This routine starts the server.  It forks, leaving the child
 *      to do all the work, so it does not have to be run in the
 *      background.  It sets up the socket, and for each incoming
 *      request, it returns an answer.  Each request consists of a
 *      host name for which the requester desires to know the
 *      Internet address.  The server will look up the name in its
 *      /etc/hosts file, and return the Internet address to the
 *      client.  An Internet address value of all ones will be returned
 *     if the host name is not found.  NOTE:  This example is valid only
 *     if the /etc/hosts file is being used to lookup host names.
 *
 */

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

                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&clientaddr_in, 0, sizeof(struct sockaddr_in));

                /* Set up address structure for the socket. */
        myaddr_in.sin_family = AF_INET;
                /* The server should receive on the wildcard address,
                 * rather than its own Internet address.  This is
                 * generally good practice for servers, because on
                 * systems which are connected to more than one
                 * network at once will be able to have one server
                 * listening on all networks at once.  Even when the
                 * host is connected to only one network, this is good
                 * practice, because it makes the server program more
                 * portable.
                 */
        myaddr_in.sin_addr.s_addr = INADDR_ANY;
                /* Find the information for the ''example'' server
                 * in order to get the needed port number.
                 */
        sp = getservbyname (''example'', ''udp'');
        if (sp == NULL) {
                printf(''%s: host not found'',
                                argv[0]);
                exit(1);
        }
        myaddr_in.sin_port = sp->s_port;

                /* Create the socket. */
        s = socket (AF_INET, SOCK_DGRAM, 0);
        if (s == -1) {
                perror(argv[0]);
                printf(''%s: unable to create socket\n'', argv[0]);
                exit(1);
        }

                /* Bind the server's address to the socket. */
        if (bind(s, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                printf(''%s: unable to bind address\n'', argv[0]);
                exit(1);
        }
        /* Now, all the initialization of the server is
         * complete, and any user errors will have already
         * been detected.  Now we can fork the daemon and
         * return to the user.  We need to do a setpgrp
         * so that the daemon will no longer be associated
         * with the user's control terminal.  This is done
         * before the fork, so that the child will not be
         * a process group leader.  Otherwise, if the child
         * were to open a terminal, it would become associated
         * with that terminal as its control terminal.  It is
         * always best for the parent to do the setpgrp.
         */
setpgrp();

switch (fork()) {
case -1:                /* Unable to fork, for some reason. */
        perror(argv[0]);
        printf(''%s: unable to fork daemon\n'', argv[0]);
        exit(1);

case 0:                 /* The child process (daemon) comes here. */
                /* Close stdin, stdout, and stderr so that they will
                 * not be kept open.  From now on, the daemon will
                 * not report any error messages.  This daemon
                 * will loop forever, waiting for requests and
                 * responding to them.
                 */
        fclose(stdin);
        fclose(stdout);
        fclose(stderr);
                /* This will open the /etc/hosts file and keep
                 * it open.  This will make accesses to it faster.
                 * If the host has been configured, however, to use the NIS
                 * server or the name server (BIND), then it is desirable
                 * not to call sethostent(1), because then a STREAM
                 * socket is used instead of datagrams for each call
                 * to gethosbyname().
                 */
        sethostent(1);
        for(;;) {
                        /* Note that addrlen is passed as a pointer
                         * so that the recvfrom call can return the
                         * size of the returned address.
                         */
                addrlen = sizeof(struct sockaddr_in);


                        /* This call will block until a new
                         * request arrives.  Then, it will
                         * return the address of the client,
                         * and a buffer containing its request.
                         * BUFFERSIZE - 1 bytes are read so that
```

```
                                * room is left at the end of the buffer
                                * for a null character.
                                */
                        cc = recvfrom(s, buffer, BUFFERSIZE - 1, 0,
                                                &clientaddr_in, &addrlen);
                        if ( cc == -1) exit(1);
                               /* Make sure the message received is
                                * null terminated.
                                */
                        buffer[cc]='\0';
                               /* Treat the message as a string containing
                                * a hostname.  Search for the name in
                                * /etc/hosts.
                                */
                        hp = gethostbyname (buffer);
                        if (hp == NULL) {
                                       /* Name was not found.  Return a
                                        * special value signifying the
                                        * error.
                                        */
                                reqaddr.s_addr = ADDRNOTFOUND;
                        } else {
                                       /* Copy address of host into the
                                        * return buffer.
                                        */
                                reqaddr.s_addr =
                                       ((struct in_addr *)(hp->h_addr))->s_addr;
                        }
                               /* Send the response back to the
                                * requesting client.  The address
                                * is sent in network byte order. Note that
                                * all errors are ignored.  The client
                                * will retry if it does not receive
                                * the response.
                                */
                        sendto (s, &reqaddr, sizeof(struct in_addr),
                                        0, &clientaddr_in, addrlen);
                }

        default:                /* Parent process comes here. */
                exit(0);
        }
}

 /*
 *                      C L I E N T . U D P
 *
 *      This is an example program that demonstrates the use of datagram
 *      sockets as an BSD IPC mechanism.  This contains the client, and is
 *      intended to operate in conjunction with the server program found
 *      in serv.udp.  Together, these two programs demonstrate many of
 *      the features of sockets, as well as good conventions for using
 *      these features.
 *
 *      This program requests a service called ''example''.  In order for
 *      it to function, an entry for it needs to exist in the
 *      /etc/services file.  The port address for this service can be
 *      any port number that is likely to be unused, such as 22375, for
 *      example.  The host on which the server will be running must also
 *      have the same entry (same port number) in its /etc/services file.
 *
 *      The ''example'' service is an example of a simple name server
 *      application.  The host that is to provide this service is
 *      required to be in the /etc/hosts file.  Also, the host providing
 *      this service presumably knows the Internet addresses of many
 *      hosts which the local host does not.  Therefore, this program
 *      will request the Internet address of a target host by name from
 *      the serving host.  The serving host will return the requested
 *      Internet address as a response, and will return an address of
 *      all ones if it does not recognize the host name.
 *
 */

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

extern int errno;

int s;                          /* socket descriptor */

struct hostent *hp;             /* pointer to host info for nameserver host */
struct servent *sp;             /* pointer to service information */

struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in servaddr_in; /* for server socket address */
struct in_addr reqaddr;         /* for returned Internet address */

#define ADDRNOTFOUND    0xffffffff      /* value returned for unknown host */
#define RETRIES 5               /* number of times to retry before giving up */

/*
 *                      H A N D L E R
 *
 *      This routine is the signal handler for the alarm signal.
 *      It simply re-installs itself as the handler and returns.
 */
handler()
{
        signal(SIGALRM, handler);
}
```

```
/*
 *                      M A I N
 *
 *      This routine is the client which requests service from the remote
 *      ''example server''.  It will send a message to the remote nameserver
 *      requesting the Internet address corresponding to a given hostname.
 *      The server will look up the name, and return its Internet address.
 *      The returned address will be written to stdout.
 *
 *      The name of the system to which the requests will be sent is given
 *      as the first parameter to the command.  The second parameter should
 *      be the name of the target host for which the Internet address
 *      is sought.
 */
main(argc, argv)
int argc;
char *argv[];
{
        int i;
        int retry = RETRIES;            /* holds the retry count */
        char *inet_ntoa();

        if (argc != 3) {
                fprintf(stderr, ''Usage:  %s <nameserver> <target>\n'', argv[0]);
                exit(1);
        }

                /* clear out address structures */
        memset ((char *)&myaddr_in, 0, sizeof(struct sockaddr_in));
        memset ((char *)&servaddr_in, 0, sizeof(struct sockaddr_in));

                /* Set up the server address. */
        servaddr_in.sin_family = AF_INET;
                /* Get the host information for the server's hostname that the
                 * user passed in.
                 */
        hp = gethostbyname (argv[1]);
        if (hp == NULL) {
                fprintf(stderr, ''%s: %s not found in /etc/hosts\n'',
                                argv[0], argv[1]);
                exit(1);
        }
        servaddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;
                /* Find the information for the ''example'' server
                 * in order to get the needed port number.
                 */
        sp = getservbyname (''example'', ''udp'');
        if (sp == NULL) {
                fprintf(stderr, ''%s: example not found in /etc/services\n'',
                                argv[0]);
                exit(1);
        }
        servaddr_in.sin_port = sp->s_port;
```

```
                /* Create the socket. */
        s = socket (AF_INET, SOCK_DGRAM, 0);
        if (s == -1) {
                perror(argv[0]);
                fprintf(stderr, ''%s: unable to create socket\n'', argv[0]);
                exit(1);
        }
                /* Bind socket to some local address so that the
                 * server can send the reply back.  A port number
                 * of zero will be used so that the system will
                 * assign any available port number.  An address
                 * of INADDR_ANY will be used so we do not have to
                 * look up the Internet address of the local host.
                 */
        myaddr_in.sin_family = AF_INET;
        myaddr_in.sin_port = 0;
        myaddr_in.sin_addr.s_addr = INADDR_ANY;
        if (bind(s, &myaddr_in, sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                fprintf(stderr, ''%s: unable to bind socket\n'', argv[0]);
                exit(1);
        }
                /* Set up alarm signal handler. */
        signal(SIGALRM, handler);

                /* Send the request to the nameserver. */
again:  if (sendto (s, argv[2], strlen(argv[2]), 0, &servaddr_in,
                                sizeof(struct sockaddr_in)) == -1) {
                perror(argv[0]);
                fprintf(stderr, ''%s: unable to send request\n'', argv[0]);
                exit(1);
        }
                /* Set up a timeout so I don't hang in case the packet
                 * gets lost.  After all, UDP does not guarantee
                 * delivery.
                 */
        alarm(5);
                /* Wait for the reply to come in.  We assume that
                 * no messages will come from any other source,
                 * so that we do not need to do a recvfrom nor
                 * check the responder's address.
                 */
        if (recv (s, &reqaddr, sizeof(struct in_addr), 0) == -1) {
                if (errno == EINTR) {
                                /* Alarm went off and aborted the receive.
                                 * Need to retry the request if we have
                                 * not already exceeded the retry limit.
                                 */
                        if ( retry) {
                                goto again;
                        } else {
                                printf(''Unable to get response from'');
                                printf('' %s after %d attempts.\n'',
                                                argv[1], RETRIES);
                                exit(1);
                        }
                } else {
                        perror(argv[0]);
                        fprintf(stderr, ''%s: unable to receive response\n'',
                                                                argv[0]);
                        exit(1);
                }
        }
        alarm(0);
                /* Print out response. */
        if (reqaddr.s_addr == ADDRNOTFOUND) {
                printf(''Host %s unknown by nameserver %s.\n'', argv[2],
                                                                argv[1]);
                exit(1);
        } else {
                printf(''Address for %s is %s.\n'', argv[2],
                        inet_ntoa(reqaddr));
        }
}
```

5

Advanced Topics for Internet Datagram Sockets

This chapter explains the following:

- SO_BROADCAST socket option.
- Specifying a default socket address.
- Synchronous I/O multiplexing with *select*.
- Sending and receiving data asynchronously.
- Nonblocking I/O.
- Using broadcast addresses.

SO_BROADCAST Socket Option

This option is AF_INET socket-specific.

SO_BROADCASTADDR establishes permission to send broadcast datagrams from the socket.

Specifying a Default Socket Address

It is possible (but not required) to specify a default address for a remote datagram socket.

This allows you to send messages without specifying the remote address each time. In fact, if you use *sendto* or *sendmsg*, an error occurs if you enter any value other than 0 for the socket address after the default address has been recorded. You can use *send* or *write* instead of *sendto* or *sendmsg* once you have specified the default address.

Use *recv* for receiving messages. Although *recvfrom* can be used, it is not necessary, because you already know that the message came from the default remote socket.

(Messages from sockets other than the default socket are discarded without notice.) *read(2)* can also be used, but does not allow you to use the MSG_PEEK flag.

Specify the default address with the *connect* system call. *connect* recognizes two special default addresses, INADDR_ANY and INADDR_BROADCAST. Using INADDR_ANY connects your socket to the IP address of your local host's primary LAN interface (for loopback connections). Using INADDR_BROADCAST connects your socket to the subnet broadcast address for your primary LAN interface; it allows you to *send* out broadcast packets that interface without specifying the subnet broadcast address.

When a datagram socket descriptor is specified in a *connect* call, *connect* associates the specified socket with a particular remote socket address. *connect* returns immediately because it only records the peer's socket address. After *connect* records the default address, any message sent from that socket is automatically addressed to the peer process and only messages from that peer are delivered to the socket.

connect may be called any number of times to change the associated destination address.

Note This call does not behave the same as a *connect* for stream sockets. There is no connection, just a default destination. The remote host that you specify as the default may or may not use *connect* to specify your local host as its default remote host. The default remote host is **not** notified if your local socket is destroyed.

---

connect and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
connect(s, addr, addrlen)
int s;
struct sockaddr *addr;
int addrlen;
```

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket requesting a default peer address
addr	pointer to the socket address	pointer to socket address of the socket to be the peer
addrlen	length of address	length of address pointed to by addr

FUNCTION RESULT: 0 if connect is successful
–1 if failure occurs

When to Specify a Default Socket Address

Which Processes	When
client or server process	after sockets are bound

Synchronous I/O Multiplexing with Select

The *select* system call can be used with sockets to provide a synchronous multiplexing mechanism. The system call has several parameters which govern its behavior. If you specify a zero pointer for the timeout parameter **timout**, *select* will block until one or more of the specified socket descriptors is ready. If **timout** is a non-zero pointer, it specifies a maximum interval to wait for the selection to complete.

select is useful for datagram socket descriptors to determine when data have arrived and are ready to be read without blocking; use the FIONREAD parameter to the *ioctl* system call to determine exactly how much data are available.

select for exceptional conditions will return true for Berkeley sockets if out-of-band data is available. *Select* will always return true for sockets which are no longer capable of being used (e.g. if a *close* or *shutdown* system call has been executed against them).

select is used in the same way as in other applications. Refer to the *select(2)* entry in the *HP-UX Reference Manual* for information on how to use *select*. For an example of a *select* system call, refer to the "I/O Multiplexing with Select" section in chapter 4.

Sending and Receiving Data Asynchronously

Asynchronous sockets allow a user program to receive an SIGIO signal when the state of the socket changes. This state change can occur, for example, when new data arrive. More information on SIGIO can be found in the "Advanced Topics for Internet Stream Sockets" chapter, "Sending and Receiving Data Asynchronously" section of this manual.

Nonblocking I/O

Sockets are created in blocking mode I/O by default. You can specify that a socket be put in nonblocking mode by using the *ioctl* system call with the FIOSNBIO request.

An example usage of this call is:

```
#include <sys/ioctl.h>
...
ioctl(s, FIOSNBIO, &arg);
```

arg is a pointer to *int*:

- When *int* equals 0, the socket is changed to blocking mode.
- When *int* equals 1, the socket is changed to nonblocking mode.

If a socket is in nonblocking mode, the following calls are affected:

recvfrom	If no messages are available to be received, *recvfrom* returns the value -1 and the EWOULDBLOCK error. This is also true for *recv* and *read*.
sendto	If there is no available message space for the message to be transmitted, *sendto* returns the value -1 and the EWOULDBLOCK error.

The O_NDELAY flag for *fcntl(2)* is also supported. If you use this flag and there is no message available to be received on a *recv, recvfrom*, or *read* call, the call returns immediately with the value of 0. If you use the O_NONBLOCK flag, the call returns immediately with the value of -1 and the EAGAIN error. This is the same as returning an end-of-file condition. This is also true for *send*, *sendto*, and *write* if there is not enough buffer space to complete the *send*.

Note The O_NDELAY and O_NONBLOCK flags have precedence over the FIOSNBIO flag. Setting both the O_DELAY and O_NONBLOCK flags is not allowed.

Using Broadcast Addresses

In place of a unique Internet address or the wildcard address, you can also specify a broadcast address. A broadcast address is an Internet address with a local address portion of all 1s.

If you use broadcast addressing, be careful not to overload your network.

6

BSD IPC Using UNIX Domain Stream Sockets

This section describes the steps involved in creating a UNIX Domain stream socket BSD IPC connection between two processes executing on the same node.

UNIX Domain (AF_UNIX) stream sockets provide bidirectional, reliable, unduplicated flow of data without record boundaries. They offer significant performance increases when compared with the use of local Internet (AF_INET) sockets, due primarily to lower code execution overhead.

The following table lists the steps involved in creating and terminating a UNIX Domain BSD IPC connection using stream sockets. Each step is described in more detail in the sections that follow the table.

Table 6-1. Building a BSD IPC Connection Using UNIX Domain Stream Sockets

Client Process Activity	System Call Used	Server Process Activity	System Call Used
create a socket	*socket()*	create a socket	*socket()*
		bind a socket address	*bind()*
		listen for incoming connection requests	*listen()*
request a connect-tion	*connect()*		
		accept connection	*accept()*
send data	*write()* or *send()*		
		receive data	*read()* or *recv()*
		send data	*write()* or *send()*
receive data	*read()* or *recv()*		
disconnect socket (optional)	*shutdown()* or *close()*	disconnect socket (optional)	*shutdown()* or *close()*

The following sections explain each of the activities mentioned in the previous table. The description of each activity specifies a system call and includes:

- what happens when the system call is used.
- when to make the call.
- what the parameters do.
- how the call interacts with other BSD IPC system calls.
- where to find details on the system call.

The UNIX Domain stream socket program examples are at the end of these descriptive sections. You can refer to the example code as you work through the descriptions.

Preparing Address Variables

Before you begin to create a connection, establish the correct variables and collect the information that you need to request a connection.

Your server process needs to:

- Declare socket address variables.
- Get the pathname (character string) for the service you want to provide.

Your client process needs to:

- Declare socket address variables.
- Get the pathname (character string) for the service you want to use.

These activities are described next. Refer to the program example at the end of this chapter to see how these activities work together.

Declaring Socket Address Variables

You need to declare a variable of type struct *sockaddr_un* to use for socket addresses.

For example, the following declarations are used in the example client program:

```
struct sockaddr_un myaddr; /* for local socket address  */
struct sockaddr_un peeraddr; /* for peer socket address  */
```

Sockaddr_un is a special case of *sockaddr* and is used with the AF_UNIX address domain. The *sockaddr_un* address structure consists of the following fields:

short *sun_family*	Specifies the address family and should always be set to AF_UNIX
u_char *sun_path[92]*	Specifies the pathname to which the socket is bound or will be bound (e.g. */tmp/mysocket*).

The server process only needs an address for its own socket. Your client process will not need an address for its own socket.

Writing the Server Process

This section discusses the calls your server process must make to connect with and serve a client process.

Creating a Socket

The server process must call *socket* to create a communication endpoint.

Socket and its parameters are described in the following table.

INCLUDE FILES:	`#include <sys/types.h>`
	`#include <sys/socket.h>`

SYSTEM CALL:	`s = socket(af, type, protocol)`
	`int af, type, protocol;`

Parameter	Description of Contents	INPUT Value
af	address family	AF_UNIX
type	socket type	SOCK_STREAM
protocol	underlying protocol to be used	0 (default)

FUNCTION RESULT:	socket number (HP-UX file descriptor) –1 if failure occurs
EXAMPLE SYSTEM CALL:	`s = socket (AF_UNIX, SOCK_STREAM, 0);`

The socket number returned is the socket descriptor for the newly created socket. This number is an HP-UX file descriptor and can be used for reading, writing or any standard file system calls after a BSD IPC connection is established. A socket descriptor is treated like a file descriptor for an open file.

When to Create Sockets

Which Processes	When
server process	before any other BSD IPC system calls

Refer to the *socket(2)* entry in the *HP-UX Reference Manual* for more information on *socket.*

Binding a Socket Address to the Server Process's Socket

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

The server process must bind a specific pathname to this socket, which is used for listening. Otherwise, a client process would not know what pathname to connect to for the desired service.

Set up the address structure with a local address (as described in the "Preparing Address Variables" section) before you make a *bind* call. *Bind* and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
bind (s, addr, addrlen)
int s;
struct sockaddr_un *addr;
int addrlen;
```

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket to be bound
addr	socket address	pointer to address to be bound to s
addrlen	length of socket address	size of struct sockaddr_un

FUNCTION RESULT:	0 if bind is successful –1 if failure occurs

EXAMPLE SYSTEM CALL:

```
struct sockaddr_un myaddr;
...
bind (ls, &myaddr, sizeof(struct sockaddr_un));
```

When to Bind Socket Addresses

Which Processes	When
server process	after socket is created and before any other BSD IPC system calls

Refer to the *bind(2)* entry in the *HP-UX Reference Manual* for more information on *bind*.

Setting the Server Up to Wait for Connection Requests

Once your server process has an address bound to it, it must call *listen* to set up a queue that accepts incoming connection requests. The server process then monitors the queue for requests (using *select(2)* or *accept*, which is described in "Accepting a Connection"). The server process cannot respond to a connection request until it has executed *listen*.

Listen and its parameters are described in the following table.

INCLUDE FILES: none

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	server socket's descriptor
backlog	maximum number of connection requests in the queue at any time	size of queue (between 1 and 20)

FUNCTION RESULT:	0 if listen is successful –1 if failure occurs
EXAMPLE SYSTEM CALL:	`listen (ls, 5);`

Backlog is the number of unaccepted incoming connections allowed at a given time. Further incoming connection requests are rejected.

When to Set Server Up to Listen

Which Processes	When
server process	after socket is created and bound and before the server can respond to connection requests

Refer to the *listen(2)* entry in the *HP-UX Reference Manual* for more information on *listen.*

Accepting a Connection

The server process can accept any connection requests that enter its queue after it executes *listen. Accept* 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 pathname 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. (For information on nonblocking I/O, see the "Nonblocking I/O" section of the "Advanced Topics for Stream Sockets" chapter.)

Accept and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = accept(ls,addr,addrlen)
int s;
int ls;
struct sockaddr_un *addr;
int *addrlen;
```

Parameter	Contents	INPUT Value	OUTPUT Value
s	socket descriptor of local socket	socket descriptor of server socket	unchanged
addr	socket address	pointer to address structure where address will be put	pointer to socket address of client socket that server's new socket is connected to
addrlen	length of address	pointer to the size of struct sockaddr_un	pointer to the actual length of address returned in addr

FUNCTION RESULT: socket descriptor of new socket if *accept* is successful
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
struct sockaddr_un peeraddr;
...
addrlen = sizeof(sockaddr_un);
s = accept (ls, &peeraddr, &addrlen);
```

There is no way for the server process to indicate which requests it can accept. It must accept all requests or none.

When to Accept a Connection

Which Processes	When
server process	after executing the listen call

Refer to the *accept(2)* entry in the *HP-UX Reference Manual* for more information on *accept*.

Writing the Client Process

This section discusses the calls your client process must make to connect with and be served by a server process.

Creating a Socket

The client process must call *socket* to create a communication endpoint.

Socket and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = socket(af, type, protocol)
int af, type, protocol;
```

Parameter	Description of Contents	INPUT Value
af	address family	AF_UNIX
type	socket type	SOCK_STREAM
protocol	underlying protocol to be used	0 (default)

FUNCTION RESULT: socket number (HP-UX file descriptor)
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
s = socket (AF_UNIX, SOCK_STREAM, 0);
```

The socket number returned is the socket descriptor for the newly created socket. This number is an HP-UX file descriptor and can be used for reading, writing or any standard file system calls after a BSD IPC connection is established. A socket descriptor is treated like a file descriptor for an open file.

When to Create Sockets

Which Processes	When
client process	before requesting a connection

Refer to the *socket(2)* entry in the *HP-UX Reference Manual* for more information on *socket*.

Requesting a Connection

Once the server process is listening for connection requests, the client process can request a connection with the *connect* call.

Connect and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
connect(s, addr, addrlen)
int s;
struct sockaddr_un *addr;
int addrlen;
```

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket requesting a connection
addr	pointer to the socket address	pointer to the socket address of the socket to which client wants to connect
addrlen	length of addr	size of address structure pointed to by addr

FUNCTION RESULT: 0 if connect is successful
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
struct sockaddr_un peeraddr;
...
connect (s, &peeraddr, sizeof(struct sockaddr_un));
```

connect initiates a connection. When the connection is ready, the client process completes its *connect* call and the server process can complete its *accept* call.

Note The client process does not get feedback that the server process has completed the *accept* call. As soon as the *connect* call returns, the client process can send data.

When to Request a Connection

Which Processes	When
client process	after socket is created and after server socket has a listening socket

Refer to the *connect(2)* entry in the *HP-UX Reference Manual* for more information on *connect.*

Sending and Receiving Data

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. Because the stream socket descriptors correspond to HP-UX file descriptors, you can use the *read* and *write* calls (in addition to *send* and *recv*) to pass data through a socket-terminated channel.

If you are considering the use of the *read* and *write* system calls instead of the *send* and *recv* calls described below, you should consider the following:

Advantage:	If you use *read* and *write* instead of *send* and *recv*, you can use a socket for *stdin* or *stdout.*
Disadvantage:	If you use *read* and write instead of *send* and *recv*, you cannot use the options specified with the *send* or *recv* *flags* parameter.

See the table called "Other System Calls," listed in the "Programming Hints" chapter for more information on which of these system calls are best for your application.

Sending Data

Send and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket sending data
msg	pointer to data buffer	pointer to data to be sent
len	size of data buffer	size of msg
flags	settings for optional flags	0

FUNCTION RESULT: number of bytes actually sent
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
count = send (s, buf, 10, 0);
```

Send blocks until the specified number of bytes have been queued to be sent, unless you are using nonblocking I/O. (For information on nonblocking I/O, see the "Nonblocking I/O" section of the "Advanced Topics for Stream Sockets" chapter.)

When to Send Data

Which Processes	When
server or client process	after connection is established

Refer to the *send(2)* entry in the *HP-UX Reference Manual* for more information on *send.*

Receiving Data

recv and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	Description of Contents	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket receiving data
buf	pointer to data buffer	pointer to buffer that is to receive data
len	maximum number of bytes that should be received	size of data buffer
flags	settings for optional flags	0

FUNCTION RESULT: number of bytes actually received
–1 if failure occurs

EXAMPLE SYSTEM CALL:

```
count = recv(s, buf, 10, 0);
```

recv blocks until there is at least 1 byte of data to be received, unless you are using nonblocking I/O. (For information on nonblocking I/O, see the "Nonblocking I/O" section of the "Advanced Topics for Stream Sockets" chapter.) The host does not wait for *len* bytes to be available; if less than *len* bytes are available, that number of bytes are 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*.

Flag Options

There are no *flag* options for UNIX Domain (AF_UNIX) sockets. The only supported value for this field is 0.

When to Receive Data

Which Processes	When
server or client process	after connection is established

Refer to the *recv(2)* entry in the *HP-UX Reference Manual* for more information on *recv*.

Closing a Socket

In most applications, you do not have to worry about cleaning up your sockets. When you exit your program and your process terminates, the sockets are closed for you.

If you need to close a socket while your program is still running, use the *close* system call. For example, you may have a daemon process that uses *fork* to create the server process. The daemon process creates the BSD IPC connection and then passes the socket descriptor to the server. You then have more than one process with the same socket descriptor. The daemon process should do a *close* of the socket descriptor to avoid keeping the socket open once the server is through with it. Because the server performs the work, the daemon does not use the socket after the *fork*.

Close decrements the file descriptor reference count and the calling process can no longer use that file descriptor.

When the last *close* is executed on a socket descriptor, any unsent data are sent before the socket is closed. Any unreceived data are lost.

Example Using UNIX Domain Stream Sockets

Note These programs are provided as examples only of UNIX Domain stream socket usage and are not Hewlett-Packard supported products.

These programming examples demonstrate how to set up and use UNIX Domain stream sockets. These sample programs can be found in the */usr/netdemo/af_unix* directory. The client program is intended to run in conjunction with the server program.

This example shows how to create UNIX Domain stream sockets and how to set up address structures for the sockets. In this example the client process sends 2000 bytes of data to the server (five times). The server process can receive data from any other process and will echo the data back to the sender.

```
/*
 *      Sample Program : AF_UNIX stream sockets, server process
 *
 *      CATCH - RECEIVE DATA FROM THE PITCHER
 *
 *      Pitch and catch set up a simple unix domain stream socket
 *      client-server connection. The client (pitch) then sends data to
 *      server (catch), throughput is calculated, and the result is
 *      printed to the client's stdout.
 */
#include <stdio.h>
#include <time.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCKNAME        "/tmp/p_n_c"
#define BUFSIZE         32*1024-1
int     timeout();
int     s;              /* server socket */

char buffer[BUFSIZE];
struct bullet {
     int bytes;
     int throughput;
     int magic;
} bullet = { 0, 0, 0 };

send_data(fd, buf, buflen)
     char *buf;
```

```
{
     int cc;

     while (buflen > 0) {
          cc = send(fd, buf, buflen, 0);

          if (cc == -1) {
               perror("send");
               exit(0);
          }

          buf += cc;
          buflen -= cc;
     }
}

recv_data(fd, buf, buflen)
     char *buf;
{
     int cc;

     while (buflen > 0) {
          cc = recv(fd, buf, buflen, 0);

          if (cc == -1) {
               perror("recv");
               exit(0);
          }

          buf += cc;
          buflen -= cc;
     }
}

main(argc, argv)
     int argc;
     char *argv[];
{
     int bufsize, bytes, cc, i, total, pid, counter_pid;
     float msec;
     struct timeval tp1, tp2;
     int ns, recvsize, secs, usec;
     struct timezone tzp;
     struct sockaddr_un sa;

/*
 * The SIGPIPE signal will be received if the peer has gone away and an attempt
 * is made to write data to the peer.  Ignoring this signal causes
 * the write operation to receive an EPIPE error.  Thus, the user is
 * informed about what happened.
*/
     signal(SIGPIPE, SIG_IGN);
     signal(SIGCLD, SIG_IGN);
```

```
    signal(SIGINT, timeout);

    setbuf(stdout, 0);
    setbuf(stderr, 0);
    if (argc > 1) {
         argv++;
         counter_pid = atoi(*argv++);
    } else
         counter_pid = 0;
/*
 * Set up the socket variables - address family, socket name.
 * They'll be used later to bind() the name to the server socket.
 */
    sa.sun_family = AF_UNIX;
    strncpy(sa.sun_path, SOCKNAME,
                   (sizeof(struct sockaddr_un) - sizeof(short)));
/*
 * Create the server socket
 */
    if ((s = socket( AF_UNIX, SOCK_STREAM, 0)) == -1) {
         perror("catch - socket failed");
         exit(0);
    }
    bufsize = BUFSIZE;
/*
 * Use setsockopt() to change the socket buffer size to improve throughput
 * for large data transfers
 */
    if ((setsockopt(s, SOL_SOCKET, SO_RCVBUF, &bufsize, sizeof(bufsize)))
         == -1) {
              perror("catch - setsockopt failed");
              exit(0);
    }
/*
 * Bind the server socket to its name
 */
    if ((bind(s, &sa, sizeof(struct sockaddr_un))) == -1) {
         perror("catch - bind failed");
         exit(0);
    }
/*
 * Call listen() to enable reception of connection requests
 * (listen() will silently change the given backlog, 0, to be 1 instead)
 */
    if ((listen(s, 0)) == -1) {
         perror("catch - listen failed");
         exit(0);
    }

next_conn:
    i = sizeof(struct sockaddr_un);
/*
 * Call accept() to accept the connection request. This call will block
 * until a connection request arrives.
 */
     if ((ns = accept(s, &sa, &i)) == -1) {
          if (errno == EINTR)
               goto next_conn;
          perror("catch - accept failed");
          exit(0);
     }
     if ((pid = fork()) != 0) {
          close(ns);
          goto next_conn;
     }
/*
     close(s);
*/
/*
 * Receive the bullet to synchronize with the other side
 */
     recv_data(ns, &bullet, sizeof(struct bullet));

     if (bullet.magic != 12345) {
          printf("catch: bad magic %d\n", bullet.magic);
          exit(0);
     }
     bytes = bullet.bytes;
     recvsize = (bytes>BUFSIZE)?BUFSIZE:bytes;
/*
 * Send the bullet back to complete synchronization
 */
     send_data(ns, &bullet, sizeof(struct bullet));

     cc = 0;
     if (counter_pid)
          kill(counter_pid, SIGUSR1);
     if (gettimeofday(&tp1, &tzp) == -1) {
          perror("catch time of day failed");
          exit(0);
     }
/*
 * Receive data from the client
 */
     total = 0;
     i = bytes;
     while (i > 0) {
          cc = recvsize < i ? recvsize : i;

          recv_data(ns, buffer, cc);
          total += cc;
          i -= cc;
     }
/*
 * Calculate throughput
 */
```

```
     if (gettimeofday(&tp2, &tzp) == -1) {
          perror("catch time of day failed");
          exit(0);
     }
     if (counter_pid)
          kill(counter_pid, SIGUSR2);
     secs = tp2.tv_sec - tp1.tv_sec;
     usec = tp2.tv_usec - tp1.tv_usec;
     if (usec < 0) {
          secs ;
          usec += 1000000;
     }
     msec = 1000*(float)secs;
     msec += (float)usec/1000;
     bullet.throughput = bytes/msec;
/*
 * Send back the bullet with throughput info, then close the
 * server socket
 */
     if ((cc = send(ns, &bullet, sizeof(struct bullet), 0)) == -1) {
          perror("catch - send end bullet failed");
          exit(0);
     }
     close(ns);
}

timeout()
{
        printf( "alarm went off -- stopping the catch process\n" );
        fprintf(stderr, "stopping the catch process\n");
        unlink(SOCKNAME);
        close(s);
        exit(6);
}

/*
*       Sample Program : AF_UNIX stream sockets, client process
*
*       PITCH - SEND DATA TO THE CATCHER
*
*       Pitch and catch set up a simple unix domain stream socket
*       client-server connection. The client (pitch) then sends data to
*       the server (catch), throughput is calculated, and the result is
*       printed to the client's stdout.
*/
#include <stdio.h>
#include <time.h>
#include <netdb.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
```

```
#define SOCKNAME    "/tmp/p_n_c"

#define BUFSIZE          32*1024-1
char buffer[BUFSIZE];

struct bullet {
     int bytes;
     int throughput;
     int magic;
} bullet = { 0, 0, 12345 };

send_data(fd, buf, buflen)
     char *buf;
{
     int cc;

     while (buflen > 0) {
          cc = send(fd, buf, buflen, 0);

          if (cc == -1) {
               perror("send");
               exit(0);
          }

          buf += cc;
          buflen -= cc;
     }
}

recv_data(fd, buf, buflen)
     char *buf;
{
     int cc;

     while (buflen > 0) {
          cc = recv(fd, buf, buflen, 0);



          if (cc == -1) {
               perror("recv");
               exit(0);
          }
          buf += cc;
          buflen -= cc;
     }
}

main( argc, argv)
     int argc;
     char *argv[];
{
     int bufsize, bytes, cc, i, total, pid;
     float msec;
     struct timeval tp1, tp2;
     int s, sendsize, secs, usec;
     struct timezone tzp;
     struct sockaddr_un sa;

/*
 * The SIGPIPE signal will be received if the peer has gone away and
 * an attempt is made to write data to the peer.  Ignoring the signal causes
 * the write operation to receive an EPIPE error.  Thus, the user is
 * informed about what happened.
*/

signal(SIGPIPE, SIG_IGN);
     setbuf(stdout, 0);
     setbuf(stderr, 0);
     if (argc < 2) {
          printf("usage: pitch Kbytes [pid]\n");
          exit(0);
     }
     argv++;

/*
 * Set up the socket variables (address family; name of server socket)
 * (they'll be used later for the connect() call)
 */
     sa.sun_family = AF_UNIX;
     strncpy(sa.sun_path, SOCKNAME,
                (sizeof(struct sockaddr_un) - sizeof(short)));
     bullet.bytes = bytes = 1024*atoi(*argv++);
     if (argc > 2)
          pid = atoi(*argv++);
     else
          pid = 0;
     sendsize = (bytes < BUFSIZE) ? bytes : BUFSIZE;
/*
 * Create the client socket
 */
     if ((s = socket( AF_UNIX, SOCK_STREAM, 0)) == -1) {
          perror("pitch - socket failed");
          exit(0);
     }
     bufsize = BUFSIZE;
/*
 * Change the default buffer size to improve throughput for
 * large data transfers
 */
     if ((setsockopt(s, SOL_SOCKET, SO_SNDBUF, &bufsize, sizeof(bufsize)))
          == -1) {
               perror("pitch - setsockopt failed");
               exit(0);
     }
```

```
/*
 * Connect to the server
 */
     if ((connect(s, &sa, sizeof(struct sockaddr_un))) == - 1) {
          perror("pitch - connect failed");
          exit(0);
     }
/*
 * send and receive the bullet to synchronize both sides
 */
     send_data(s, &bullet, sizeof(struct bullet));
     recv_data(s, &bullet, sizeof(struct bullet));

     cc = 0;
     if (pid)
          kill(pid,SIGUSR1);
     if (gettimeofday(&tp1, &tzp) == -1) {
          perror("pitch time of day failed");
          exit(0);
     }
     i = bytes;
     total = 0;
/*
 * Send the data
 */
     while (i > 0) {
          cc = sendsize < i ? sendsize : i;
          send_data(s, buffer, cc);
          i -= cc;
          total += cc;
     }
/*
 * Receive the bullet to calculate throughput
 */
     recv_data(s, &bullet, sizeof(struct bullet));

     if (gettimeofday(&tp2, &tzp) == -1) {
          perror("pitch time of day failed");
          exit(0);
     }
     if (pid)
          kill(pid, SIGUSR2);
/*
 * Close the socket
 */
     close(s);
     secs = tp2.tv_sec - tp1.tv_sec;
     usec = tp2.tv_usec - tp1.tv_usec;
     if (usec < 0) {
          secs;
          usec += 1000000;
     }
     msec = 1000*(float)secs;
```

```
        msec += (float)usec/1000;
        printf("PITCH: %d Kbytes/sec\n", (int)(bytes/msec));
        printf("CATCH: %d Kbytes/sec\n", bullet.throughput);
        printf("AVG:   %d Kbytes/sec\n", ((int)(bytes/msec)+bullet.throughput)/2);
}
```

7

BSD IPC Using UNIX Domain Datagram Sockets

This section describes communication between processes using UNIX Domain datagram sockets. The UNIX Domain only allows communication between processes executing on the same machine. In contrast to pipes, it does not require the communicating processes to have common ancestry. For more information on the UNIX Domain protocol, refer to the *unix(7p)* entry in the *HP-UX Reference Pages.*

UNIX domain (AF_UNIX) datagram sockets provide bidirectional, reliable, unduplicated flow of data while preserving record boundaries. Domain sockets significantly improve performance when compared to local IP loopback, due primarily to the lower code execution overhead and the fact that data is looped back at the protocol layer rather than at the driver layer.

AF_UNIX datagram sockets allow you to send and receive messages **without establishing a connection.** Each message includes a destination address. Processes involved in data transfer are not required to have a client-server relationship; the processes can be symmetrical.

AF_UNIX datagram sockets allow you to send to many destinations from one socket, and receive from many sources with one socket. There is no two-process model, although a two-process model is the simplest case of a more general multi-process model. The terms server and client are used in this section only in the application sense. For example, you might have a server process that receives requests from several clients on the same machine. This server process can send replies back to the various clients. This can all be done with one AF_UNIX datagram socket for the server.

The simplest two-process model is used in this section to describe AF_UNIX datagram sockets.

The following table lists the steps required to exchange data between AF_UNIX datagram sockets.

Table 7-1. Setting Up for Data Transfer Using AF_UNIX Datagram Sockets

Client Process Activity	System Call Used	Server Process Activity	System Call Used
create a socket	*socket()*	create a socket	*socket()*
bind a socket	*bind()*	bind a socket	*bind()*
send message	*sendto()* or *sendmsg()*		
		receive message	*recvfrom()* or *recvmsg()*
		send message	*sendto()* or *sendmsg()*
receive message	*recvfrom()* or *recvmsg()*		

The following sections discuss each of the activities mentioned in the table above. The description of each activity specifies a system and includes:

- What happens when the system call is used.
- When to make the system call.
- What the parameters do.
- Where to find details on the system call.

The domain datagram sockets programming examples are at the end of these descriptive sections. You can refer to them as you work through the descriptions.

Preparing Address Variables

Before your client process can make a request of the server process, you must establish the correct variables and collect the information you need about the server process.

Your server process needs to:

- Declare socket address variables.
- Get the pathname (character string) for the service you want to provide.

Your client process needs to :

- Declare socket address variables.
- Get the pathname (character string) for the service you want to use.

Next, we will describe these activities in detail. You can also refer to the program example at the end of this chapter to see how it is done.

Declaring Socket Address Variables

You need to declare a variable of type struct *sockaddr_un* to use for the socket address for both processes.

For example, the following declarations are used in the example server program:

```
   struct sockaddr_un    servaddr;     /* server socket address */
```

Sockaddr_un is a special case of *sockaddr* and is used with AF_UNIX address domain. The *sockaddr_un* address structure is defined in *sys/un.h* and consists of the following fields:

short sun_family	Specifies the address family and should always be set to AF_UNIX
u_char sun_path[92]	Specifies the pathname to which the socket is bound or will be bound (eg : */tmp/myserver*)

The server process only needs one address for its socket. Any process that knows the address of the server process can then send messages to it. Thus, your client process needs to know the address of the server socket. The client process will not need an address for its own socket, unless other processes need to refer to the client process.

Writing the Server and Client Processes

This section discusses the calls your server and client processes must make.

Creating Sockets

Both processes must call *socket* to create communication endpoints.

Socket and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
s = socket(af, type, protocol)
int   af, type, protocol;
```

Parameter	Description	INPUT Value
af	address family	AF_UNIX
type	socket type	SOCK_DGRAM
protocol	underlying protocol to be used	0 (default)

FUNCTION RESULT: socket number (HP-UX file descriptor) if successful
-1 if socket call fails

EXAMPLE SYSTEM CALL:

```
#include <sys/type.h>
#include <sys/socket.h>
...
s = socket(AF_UNIX, SOCK_DGRAM, 0)
```

When to Create Sockets

Which Process	When
server or client process	before any other BSD IPC system calls

Refer to the *socket(2)* entry in the *HP-UX Reference Pages* for more information on *socket*.

Binding Socket Addresses to UNIX Domain Datagram Sockets

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

The server process must bind a specific pathname to its socket. Set up the address structure with a local address (as described in the section on "Preparing Address Variables") before the server makes a call to *bind*. You can also refer to the program example at the end of this chapter to see how it is done.

The *bind* system call creates the *inode* file. If the *inode* file is not deallocated after bound sockets are closed, the names will continue to exist and cause directories to fill up with the unused files. To avoid directories filling up with these unused files, you can remove the files by calling *unlink* or remove them manually with the *rm* command.

Bind and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
bind(s, addr, addrlen);
int    s;
struct  sockaddr_un    *addr;
int    addrlen;
```

Parameter	Description	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket to be bound
addr	socket address	pointer to address to be bound to s
addrlen	length of socket address	size of struct *sockaddr_un* address

FUNCTION RESULT: 0 if bind is successful
-1 if bind fails

EXAMPLE SYSTEM CALL:

```
#include <sys/type.h>
#include <sys/socket.h>
#include <sys/un.h>
#define  SOCKET_PATH /tmp/myserver
struct  sockaddr_un    servaddr;
```

```
...
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, SOCKET_PATH);
unlink(SOCKET_PATH);

bind(s, &servaddr, sizeof(struct sockaddr_un));
```

When to Bind Socket Addresses

Which Process	When
server	after socket is created and before any other BSD IPC system calls

Refer to the *bind(2)* entry in the *HP-UX Reference Pages* for more information on *bind*.

Sending and Receiving Messages

The *sendto* and *recvfrom* (or *sendmsg* and *recvmsg*) system calls are usually used to transmit and receive messages with datagram sockets. They are described in the next sections.

Sending Messages

Use *sendto* or *sendmsg* to send messages. *sendmsg* is similar to *sendto*, except *sendmsg* allows the send data to be gathered from several buffers.

Sendto and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

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

Parameter	Description	INPUT Value
s	socket descriptor of local socket	socket descriptor of socket that is sending the message
msg	pointer to data buffer	pointer to data to be sent
len	size of data buffer	size of msg
flags	settings for optional flags	0 (no options are currently supported)
to	address of recipient socket	pointer to the socket address that message should be sent to
tolen	size of to	length of address structure that *to* points to

FUNCTION RESULT: number of bytes actually sent if *sendto* succeeds
-1 if sendto call fails

EXAMPLE SYSTEM CALL:

```
struct  sockaddr_un    servaddr;
...
count = sendto(s, argv[2], strlen(argv[2]), 0, &servaddr,
sizeof(struct sockaddr_un);
```

When to Send Data

Which Process	When
server or client process	after server has bound to an address

Refer to the *send(2)* entry in the *HP-UX Reference Pages* for more information on *sendto* and *sendmsg.*

Receiving Messages

Use *recvfrom* or *recvmsg* to receive messages. *recvmsg* is similar to *recvfrom*, except *recvmsg* allows the read data to be scattered into buffers.

Recv can also be used if you do not need to know what socket sent the message. However, if you want to send a response to the message, you must know where it came from. Except for the extra information returned by *recvfrom*, the two calls are identical.

Recvfrom and its parameters are described in the following table.

INCLUDE FILES:

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

SYSTEM CALL:

```
count = recvfrom(s, msg, len, flags, from, fromlen)
int    s;
char   *msg;
int    len, flags;
struct  sockaddr_un *from;
int    *fromlen;
```

Parameter	**Description**	**INPUT Value**	**OUTPUT VALUE**
s	socket descriptor of local socket	socket descriptor of socket receiving the message	unchanged
msg	pointer to data buffer	pointer to buffer that is to receive data	pointer to received data
len	maximum number of bytes that should be received	size of data buffer	unchanged
flags	settings for optional flags	0 (no options are supported)	unchanged
from	address of socket that sent message	pointer to address structure, not used for input	pointer to socket address of socket that sent the message
fromlen	pointer to the size of from	pointer to size of from	pointer to the actual size of address returned

FUNCTION RESULT: number of bytes actually received if *recvfrom* succeeds
-1 if *recvfrom* call fails

EXAMPLE SYSTEM CALL:

```
struct  sockaddr_un    fromaddr;
int    fromlen;
...
count = recvfrom(s, msg, sizeof(msg), 0, &fromaddr, &fromlen);
```

Recvfrom blocks until there is a message to be received.

No more than *len* bytes of data are returned. The entire message is read in one *recvfrom*, *recvmsg*, *recv*, or *read* operation. If the message is too long for the receive buffer, the excess data are discarded. Because only one message can be returned in a *recvfrom* call, if a second message is in the queue, it is not affected. Therefore, the best technique is to receive as much as possible on each call.

Refer to the *recv(2)* entry in the *HP-UX Reference Pages* for more information on *recvfrom* and *recvmsg*.

Closing a Socket

In most applications, you do not have to close the sockets. When you exit your program and your process terminates, the sockets are closed for you.

If you need to close a socket while your program is still running, use the *close* system call.

You may have more than one process with the same socket descriptor if the process with the socket descriptor executes a *fork*. *Close* decrements the file descriptor count and the calling process can no longer use that file descriptor.

When the last *close* is executed on a socket, any unsent messages are sent and the socket is closed. Any unreceived data are lost.

Example Using UNIX Domain Datagram Sockets

Note These programs are provided as examples only of UNIX Domain datagram socket usage and are not Hewlett-Packard supported products.

These programming examples demonstrate how to set up and use UNIX Domain datagram sockets. These sample programs can be found in the */usr/netdemo/af_unix* directory. The client program is intended to run in conjunction with the server program.

This example shows how to create UNIX Domain datagram sockets and how to set up address structures for the sockets. In this example the client process sends 2000 bytes of data to the server (five times). The server process can receive data from any other process and will echo the data back to the sender.

The source code for these two programs follows.

```
/*
 *      AF_UNIX datagram server process
 *
 *      This is an example program that demonstrates the use of AF_UNIX
 *      datagram sockets as a BSD IPC mechanism.  This program contains the
 *      server and is intended to operate in conjunction with the
 *      client program.
 *
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>

#define SOCKET_PATH      /tmp/myserver
#define bzero(ptr, len)  memset((ptr), NULL, (len))

int     timeout();

main()
{
        int     sock;
        int     slen, rlen, expect;
        unsigned char   sdata[5000];
        struct  sockaddr_un servaddr;           /* address of server */
        struct  sockaddr_un from;
```

```
int     fromlen;

/*      Escape hatch so blocking calls don't wait forever       */

signal(SIGALRM,timeout);
alarm((unsigned long) 120);

/*      Create a UNIX datagram socket for server        */

if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        perror("server: socket");
        exit(1);
}

/*      Set up address structure for server socket      */

bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, SOCKET_PATH);

if (bind(sock, &servaddr, sizeof(servaddr)) < 0) {
        close(sock);
        perror("server: bind");
        exit(2);
}

/*      Receive data from anyone and echo back data to the sender
*       Note that fromlen is passed as a pointer so the recvfrom
*       call can return the size of the returned address.
*/
expect = 5 * 2000;
while (expect > 0) {
        fromlen = sizeof(from);
        rlen = recvfrom(sock, sdata, 2000, 0, &from, &fromlen);
        if (rlen == -1) {
                perror("server : recv\n");
                exit(3);
        } else {
                expect -= rlen;
                printf("server : recv'd %d bytes\n",rlen);
                slen = sendto(sock, sdata, rlen, 0, &from, fromlen);
                if (slen <0) {
                        perror ("server : sendto\n");
                        exit (4);
                }
        }
}
/*      Use unlink to remove the file (inode) so that the name
*       will be available for the next run.
*/
unlink(SOCKET_PATH);
close(sock);
printf("Server done\n");
```

```
        exit(0);
}

timeout()               /* escape hatch so blocking calls don't wait forever */
{
        printf( alarm received   stopping server\n  );
        fprintf(stderr,  stopping the server process\n );
        exit(5);
}

/*
*       AF_UNIX datagram client process
*
*       This is an example program that demonstrates the use of AF_UNIX
*       datagram sockets as a BSD IPC mechanism.  This contains the
*       client, and is intended to operate in conjunction with the
*       server program.
*
*/
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/un.h>
#include <stdio.h>
#include <signal.h>
#include <netdb.h>

#define SOCKET_PATH      /tmp/myserver
#defineSOCKET_PATHCLNT /tmp/my_af_unix_client
#define bzero(ptr, len)  memset((ptr), NULL, (len))
int     timeout();

main()
{
        int     sock;
        int     j, slen, rlen;
        unsigned char   sdata[2000];            /* send data */
        unsigned char   rdata[2000];            /* receive data */
        struct  sockaddr_un servaddr;           /* address of server */
        struct  sockaddr_un clntaddr;           /* address of client */
        struct  sockaddr_un from;
        int     fromlen;

        /*      Stop the program if not done in 2 minutes */

        signal(SIGALRM, timeout);
        alarm((unsigned long) 120);

        /*      Fork the server process to receive data from client */

        printf("Client : Forking server\n");
        if (fork() == 0 ) {
                execl("./server",  server , 0 );
        printf("Cannot exec ./server.\n");
        exit(1);
}

/*      Initialize the send data        */

for (j = 0; j < sizeof(sdata); j++)
        sdata[j] = (char) j;

/*      Create a UNIX datagram socket for client        */

if ((sock = socket(AF_UNIX, SOCK_DGRAM, 0)) < 0) {
        perror("client: socket");
        exit(2);
}

/*      Client will bind to an address so the server will get
 *      an address in its recvfrom call and use it to
 *      send data back to the client.
 */

bzero(&clntaddr, sizeof(clntaddr));
clntaddr.sun_family = AF_UNIX;
strcpy(clntaddr.sun_path, SOCKET_PATHCLNT);

if (bind(sock, &clntaddr, sizeof(clntaddr)) < 0) {
        close(sock);
        perror("client: bind");
        exit(3);
}

/*      Set up address structure for server socket */

bzero(&servaddr, sizeof(servaddr));
servaddr.sun_family = AF_UNIX;
strcpy(servaddr.sun_path, SOCKET_PATH);

for (j = 0; j < 5; j++) {
        sleep(1);
        slen = sendto(sock, sdata, 2000, 0,
                (struct sockaddr *) &servaddr, sizeof(servaddr));
        if (slen<0) {
                perror("client: sendto");
                exit(4);
        }
        else {
                printf("client : sent %d bytes\n", slen);
                fromlen = sizeof(from);
                rlen = recvfrom(sock, rdata, 2000, 0, &from, &fromlen);
                if (rlen == -1) {
                        perror("client: recvfrom\n");
                        exit(5);
                } else
```

```
                                printf("client : received %d bytes\n", rlen);
                }
        }
        /*      Use unlink to remove the file (inode) so that the name
         *      will be available for the next run.
         */
        sleep(1);
        unlink(SOCKET_PATHCLNT);
        close(sock);
        printf("Client done\n");
        exit(0);
}

timeout()               /* escape hatch so blocking calls don't wait forever */
{
        printf(  alarm went off   stopping client\n  );
        fprintf(stderr,  stopping the client process\n );
        exit(6);
}
```

8

Programming Hints

This chapter contains information for:

- Troubleshooting.
- Port addresses.
- Using diagnostic utilities as troubleshooting tools.
- Adding a server process to the Internet daemon.
- Summary tables for system and library calls.

Note Refer to the "Portability Issues" appendix for information about the differences between 4.3 BSD and the HP-UX implementation of BSD IPC.

Troubleshooting

The first step to take is to avoid many problems by using good programming and debugging techniques. Your programs should check for a returned error after each system call and print any that occur. For example, the following program lines print an error message for *read*:

```
cc=read(sock,buffer,1000);
if (cc<0) {
  perror ("reading message")
  exit(1)
}
```

Refer to the *HP-UX Reference Manual* for information about *perror(3C)*. Also refer to the *HP-UX Reference Manual* for information about errors returned by the BSD IPC system calls such as *read*.

You can also compile your program with the debugging option (**-g**) and use one of the debuggers (e.g. cdb or xdb) to help debug the programs.

Port Addresses

The following port values are reserved for the super-user: 1 - 1023, 1260, 1536, 1542 and 4672. These ports are for:

Port Addresses	Used By
1 - 1023	ARPA/Berkeley services
1260	NS daemon *rlbdaemon*
1536	NS daemon *nftdaemon*
1542	NS service Remote Process Management (Series 500 only)
4672	NFS Services

It is possible that you could assign one of these ports and cause a service to fail. For example, if the *nftdaemon* is not running, and you assign its port, when you try to start the *nftdaemon*, it fails.

See the */etc/services* file for the list of reserved ports.

Using Diagnostic Utilities as Troubleshooting Tools

You can use the following diagnostic utilities to help debug your programs. It is helpful if you have multiple access to the system so you can obtain information about the program while it is running.

ping	Use *ping* to verify the physical connection with the destination node.
netstat	Use *netstat* to display sockets and associations to help you troubleshoot problems in your application programs. Use *netstat* to determine if your program has successfully created a connection. If you are using stream sockets (TCP protocol), *netstat* can provide the TCP state of the connection. To check the status of a connection at any point in the program, use the *sleep* (seconds) statement in your program to pause the program. While the program is paused, execute *netstat -a* from another terminal.
Network Tracing	*Network Tracing* can be used to trace packets. For the trace information to be useful, you must have a working knowledge of network protocols.
Network Event Logging	*Network Event Logging* is an error logging mechanism. Use it in conjunction with other diagnostic tools.

These utilities are described in detail in the *Installing and Administering LAN/9000 Software* manual.

Adding a Server Process to the Internet Daemon

This section contains example BSD IPC programs that use the Internet daemon, called *inetd*. For more information on *inetd*, refer to the *Installing and Administering ARPA Services* manual and the *inetd(1M)* entry in the *HP-UX Reference* manual.

You can invoke the example server programs from *inetd* if you have **super-user** capabilities and you make the following configuration modifications:

- Add the following lines to the */etc/inetd.conf* file:

```
example  stream  tcp  nowait  root  <path>/server.tcp  server.tcp
example  dgram   udp  wait    root  <path>/server.udp  server.udp
```

where `<path>` is the path to the files on **your** host. (For detailed information on this file, refer to the *Installing and Administering ARPA Services* manual or to the *inetd.conf(4)* entry in the *HP-UX Reference Pages*.)

- Add the following lines to the */etc/services* file:

```
example 22375/tcp
example 22375/udp
```

- If *inetd* is already running, execute the following command so that *inetd* recognizes the changes:

```
/etc/inetd -c
```

These example programs do the same thing as the previous example servers do, but they are designed to be called from *inetd*. They do not have daemon loops or listen for incoming connection requests, because *inetd* does that. The source code for the two example servers follows.

```
/*
 *                       S E R V E R . T C P
 *
 *      This is a variation of the example program called serv.tcp.
 *      This one performs the same function, except that it is
 *      designed to be called from /etc/inetd.  Hence, this version
 *      does not contain a daemon loop, and does not listen for incoming
 *      connections on the socket.  /etc/inetd does these functions.  This
 *      server simply assumes that the socket to receive the messages
 *      from and send the responses to is file descriptor 0 when
 *      the program is started.  It also assumes that the client's
 *      connection is already established to the socket.  For the sake
 *      of simplicity, the activity logging functions of serv.tcp
 *      have also been removed.
 *
 */

/*
 *                             M A I N
 *
 *      This is the actual server routine that the /etc/inetd forks to
 *      handle each individual connection.  Its purpose is to receive
 *      the request packets from the remote client, process them,
 *      and return the results to the client.
 *
 */
main()
{
        char buf[10];           /* This example uses 10 byte messages. */
        int len, len1;

                /* Go into a loop, receiving requests from the remote
                 * client.  After the client has sent the last request,
                 * it will do a shutdown for sending, which will cause
                 * an end-of-file condition to appear on this end of the
                 * connection.  After all of the client's requests have
                 * been received, the next recv call will return zero
                 * bytes, signalling an end-of-file condition.  This is
                 * how the server will know that no more requests will
                 * follow, and the loop will be exited.
                 */
        while (len = recv(0, buf, 10, 0)) {
                if (len == -1) {
                        exit (1); /* error from recv */
                }
                        /* The reason this while loop exists is that there
                         * is a remote possibility of the above recv returning
                         * less than 10 bytes.  This is because a recv returns
                         * as soon as there is some data, and will not wait for
                         * all of the requested data to arrive.  Since 10 bytes
                         * is relatively small compared to the allowed TCP
                         * packet sizes, a partial receive is unlikely.  If
                         * this example had used 2048 bytes requests instead,
                 * a partial receive would be far more likely.
                 * This loop will keep receiving until all 10 bytes
                 * have been received, thus guaranteeing that the
                 * next recv at the top of the loop will start at the
                 * beginning of the next request.
                 */
            while (len < 10) {
                    len1 = recv(0, &buf[len], 10-len, 0);
                    if (len1 == -1) {
                            exit (1);
                    }
                    len += len1;
            }
                     /* This sleep simulates the processing of the
                      * request that a real server might do.
                      */
            sleep(1);
                      /* Send a response back to the client.  */
            if (send(0, buf, 10, 0) != 10) {
                    exit (1);
            }
        }
}
            /* The loop has terminated, because there are no
             * more requests to be serviced.
        exit (0);
}
```

```
/*
 *                      S E R V E R . U D P
 *
 *      This is a variation of the example program called serv.udp.
 *      This one performs the same function, except that it is
 *      designed to be called from /etc/inetd.  Hence, this version
 *      does not contain a daemon loop, and does not wait for requests
 *      to arrive on a socket.  /etc/inetd does these functions.  This
 *      server simply assumes that the socket to receive the message
 *      from and send the response to is file descriptor 0 when
 *      the program is started.  It also assumes that the client's
 *      request is already ready to be received from the socket.
 *
 */


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

#define BUFFERSIZE      1024    /* maximum size of packets to be received */
int cc;                         /* contains the number of bytes read */
char buffer[BUFFERSIZE];        /* buffer for packets to be read into */

struct hostent *hp;             /* pointer to host info for requested host */

struct sockaddr_in clientaddr_in;       /* for client's socket address */
struct in_addr reqaddr;         /* for requested host's address */

#define ADDRNOTFOUND    0xffffffff      /* return address for unfound host */

/*
 *                      M A I N
 *
 *      This routine receives the request and returns an answer.
 *      Each request consists of a
 *      host name for which the requester desires to know the
 *      internet address.  The server will look up the name in its
 *      /etc/hosts file, and return the internet address to the
 *      client.  An internet address value of all ones will be returned
 *      if the host name is not found.
 *
 */
main()
{
        int  addrlen;


                /* clear out address structure */
        memset ((char *)&clientaddr_in, 0, sizeof(struct sockaddr_in));

                /* Note that addrlen is passed as a pointer
```

```
                 * so that the recvfrom call can return the
                 * size of the returned address.
                 */
        addrlen = sizeof(struct sockaddr_in);
                /* This call will
                 * return the address of the client,
                 * and a buffer containing its request.
                 * BUFFERSIZE - 1 bytes are read so that
                 * room is left at the end of the buffer
                 * for a null character.
                 */
        cc = recvfrom(0, buffer, BUFFERSIZE - 1, 0 &clientaddr_in, &addrlen);
        if ( cc == -1) exit(1);
                /* Make sure the message received in
                 * null terminated.
                 */
        buffer[cc]='\0';
                /* Treat the message as a string containing
                 * a hostname.  Search for the name
                 * in /etc/hosts.
                 */
        hp = gethostbyname (buffer);
        if (hp == NULL) {
                        /* Name was not found.  Return a
                         * special value signifying the error.
                        /*
                reqaddr.s_addr = ADDRNOTFOUND;
        } else {
                        /* Copy address of host into the
                         * return buffer.
                        /*
                reqaddr.s_addr =
                       ((struct in_addr *)(hp->h_addr))->s_addr;
        }
                /* send the response back to the requesting client.  The address
                 * is sent in network byte order.  Note that
                 * all errors are ignored.  The client
                 * will retry if it does not receive
                 * the response.
                */
        sendto (0, &reqaddr, sizeof(struct in_addr), 0,
                                &clientaddr_in, addrlen);
        exit(0);
}
```

Summary Tables for System and Library Calls

The following table contains a summary of the BSD IPC system calls.

Table 8-1. BSD IPC System Calls

System Call	Description
socket	Creates a socket, or communication endpoint for the calling process.
bind	Assigns a socket address to the socket specified by the calling process.
listen	Sets up a queue for incoming connection requests. (Stream sockets only.)
connect	For stream sockets, requests and creates a connection between the remote socket (specified by address) and the socket (specified by descriptor) of the calling process. For datagram sockets, permanently specifies the remote peer socket.
accept	Receives a connection between the socket of the calling process and the socket specified in the associated connect call. (Stream sockets only.)
send, sendto, sendmsg	Sends data from the specified socket.
recv, recvfrom, recvmsg	Receives data at the specified socket.
shutdown	Disconnects the specified socket.
getsockname	Gets the socket address of the specified socket.

Table 8-1. BSD IPC System Calls (con't)

System Call	Description
getsockopt, setsockopt	Gets, or sets, the options associated with a socket.
getpeername	Gets the name of the peer socket connected to the specified socket.

The following table contains a summary of the other system calls that can be used with BSD IPC.

Table 8-2. Other System Calls

System Call	Description
read	Can be used to read data at stream or datagram sockets just like *recv* or *recvfrom*, without the benefit of the *recv* flags. Read offers implementation independence; the descriptor can be for a file, a socket or any other object.
write	Can be used to write data from stream sockets (and datagram sockets if you declare a default remote socket address) just like send. Write offers implementation independence; the descriptor can be for a file, a socket or any other object.
close	Deallocates socket descriptors. The last close can be used to destroy a socket. Close does a graceful disconnect or a hard close, depending on the LINGER option. Refer to the sections on "Closing a Socket."
select	Can be used to improve efficiency for a process that accesses multiple sockets or other I/O devices simultaneously. Refer to the sections on "Synchronous I/O Multiplexing with Select."
ioctl	Can be used for finding the number of receivable bytes with FIONREAD and for setting the nonblocking I/O flag with FIOSBNBIO. Can also be used for setting a socket to receive asynchronous signals with FIOASYNC.
fcntl	Can be used for duplicating a socket descriptor and for setting the O_NDELAY or O_NONBLOCK flag.

BSD IPC attempts to isolate host-specific information from applications by providing library calls that return the necessary information.

The following table contains a summary of the library calls used with BSD IPC. The library calls are in the common "c" library named *libc.a*. Therefore, there is no need to specify any library name on the *cc* command line to use these library calls — *libc.a* is used automatically.

Table 8-3. Library Calls

Library Call	Description
htonl *htons* *ntohl* *ntohs*	convert values between host and network byte order (for portability to DEC VAX hosts)
inet_addr *inet_lnaof* *inet_makeaddr* *inet_netof* *inet_network*	Internet address manipulation routines
setservent *endservent* *getservbyname* *getservbyport* *getservent*	get or set service entry
setprotoent *endprotoent* *getprotobyname* *getprotobynumber* *getprotoent*	get or set protocol entry

Table 8-3. Library Calls (con't)

Library Call	Description
setnetent *endnetent* *getnetbyaddr* *getnetbyname* *getnetent*	get or set network entry
sethostent *endhostent* *gethostbyaddr* *gethostbyname* *gethostent*	get or set host entry

A

Example Programs

These sample programs can be found in the */usr/netdemo/socket* directory.

```
= /*
 *                      A S Y N C . S E R V
 *
 *      This program demonstrates the use of Asynchronous datagram
 *      and stream sockets.  It contains the server, and is
 *      intended to operate in conjunction with the client program
 *      found in async.clnt.  Together, these programs
 *      illustrate a very simple application of asynchronous
 *      sockets, and therefore lack the robustness of typical
 *      situations.  A program capable of handling all SIGIO
 *      interrupts requires substantial programmer investment, and is
 *      beyond the scope of this example.
 *
 *      This program provides two services called ''sigex_udp''
 *      and ''sigex_tcp'', for datagram and streams, respectively.  In
 *      order for it to function, entries need to exist in the
 *      /etc/services file.  The port address for these services can be
 *      any port numbers that are likely to be unused, such as 22373
 *      and 22374, for example.  The host on which the client will
 *      be running must also have the same entries (same port numbers)
 *      in its /etc/services file.
 *
 * Algorithm for Async.serv:
 *
 *      Set up:
 *              Catch SIGIO signal
 *              Address family
 *              Local Internet address = wildcard
 *      Datagram socket setup:
 *              Get the port address of the desired service
 *              Create the datagram socket
 *              Bind socket and make it asynchronous (set_up_async())
 *      Streams socket setup:
 *              Get the port address of the desired service
 *              Create the streams socket
 *              Bind socket and make it asynchronous (set_up_async())
 *              Create a listen queue for the socket
 *      Loop Forever or Until SIGIO interrupt
 *
```

```
* Algorithm for Async.serv's interrupt handler :
*

*       Set up:
*               Define Macros
*               Notify operator of interrupt
*               Set readmask for our datagram and streams sockets
*               Set select call timeout to zero
*       Select on datagram and streams sockets
*       If datagram socket selected
*               Read and print out data
*       Endif
*       If streams socket selected
*               Accept the connection request
*               Read and print out data
*               Exit
*       Endif
*

***************************************************************/


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

int         ds;                     /* datagram socket descriptor */
int         ss;                     /* streams socket descriptor  */

struct hostent *hp;       /* ptr to host info for remote host */
struct servent *sp;       /* ptr to service information */
struct sockaddr_in myaddr;       /* local socket address */
struct sockaddr_in peeraddr_in;  /* remote socket address */

/***************************************************************


*                         M A I N
*
***************************************************************/
main(argc, argv)
int     argc;
char    *argv[];
{

        struct sigvec vec;
        int     io_handler();    /* SIGIO interrupt handler */
```

```
/* Set up asynchronous notification of socket event */
vec.sv_handler = (int *) io_handler;
vec.sv_mask = 0;
vec.sv_flags = 0;
if ( sigvector(SIGIO, &vec, (struct sigvec *) 0) == -1)
        perror('' sigvector(SIGIO)'');

/* Set the Address Family */
myaddr.sin_family = AF_INET;

/*  Use a wildcard for the local Internet address */
myaddr.sin_addr.s_addr = INADDR_ANY;

/* Get the port address of my service & insert
        *  it in data struct.
        */
sp = getservbyname(''sigex_udp'', ''udp'');
if (sp == NULL) {
        printf('' sigex_udp service not found in /etc/services\n'');
        exit(1);
}
myaddr.sin_port = sp->s_port;

/* Create the datagram socket */
ds = socket(myaddr.sin_family, SOCK_DGRAM, 0);
if (ds == -1) {
        perror(argv[0]);
        printf(''%s: unable to create datagram socket\n'', argv[0]);
        exit(1);
}

/*  Make this socket asynchronous  */
set_up_async(ds);


/* Get the streams socket port address information */
sp = getservbyname(''sigex_tcp'', ''tcp'');
if (sp == NULL) {
        printf(''%s: sigex_tcp service not found in /etc/services\n'');
        exit(1);
}
myaddr.sin_port = sp->s_port;

/* Create the stream socket */
ss = socket(myaddr.sin_family, SOCK_STREAM, 0);
if (ss == -1) {
        perror(argv[0]);
        printf(''%s: unable to create datagram socket\n'', argv[0]);
        exit(1);
}

/*  Make this socket asynchronous  */
set_up_async(ss);
```

```
        /* Create a listen queue for the stream socket */
        /* Listen call doesn't apply to datagram sockets */
        if (listen(ss, 5) == -1 ) {
                perror(argv[0]);
                printf(''%s: unable to listen \n'', argv[0]);
                exit(1);
        }



        /*
                *  The following loop simulates any other processing
                *  that the server might do here.  The SIGIO interrupt
                *  will break this loop and execution will go to the
                *  interrupt handler.
                */
        printf('' Server entering a tight loop...\n\n'');
        for (; ; ) {
        }

} /* end main */


/**********************************************************************
*       SET_UP_ASYNC(S)
*       This routine will bind the sockets and activate asynchronous
*       notification of a socket event.
**********************************************************************/
set_up_async(s)
int         s;
{
        int      flag = 1;

        /* Bind the listen address to the socket */
        if (bind(s, &myaddr, sizeof(myaddr)) == -1) {
                perror('' unable to bind address\n'');
                exit(1);
        }

        /*  Set the socket state for Asynchronous  */
        if (ioctl(s, FIOASYNC, &flag) == -1) {
                perror('' can't set async on socket '');
                exit(1);
        }

        /* Arrange for the current process to receive
                *  SIGIO when the state of the socket changes.
                *  Process group negative == deliver to process.
                */
        flag = -getpid();
        if (ioctl(s, SIOCSPGRP, &flag) == -1) {
                perror(''can't get the process group.'');
        }
} /* end set_up_async */



/****************************************************************
*        IO_HANDLER()
*         Execution jumps here upon receipt of the SIGIO interrupt.
****************************************************************/
io_handler()
#include <sys/param.h>  /* standard parameter definitions */
#include ys/types.h

/*
 * These macros are used with select().  select() uses bit masks of file
 * descriptors in long integers. These macros manipulate such bit fields
 * (the file system macros use chars).  FD_SETSIZE may be defined by the user
 * but must be = u.u_highestfd + 1.  Since the absolute limit on the number
 * of per process open files is 2048, FD_SETSIZE must be defined to be
 * large enough to accommodate this many file descriptors.  Unless the user
 * has this many file opened, he should redefine FD_SETSIZE to a smaller number.
*/

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 DONT_CARE       (char *) 0
#define BUFLEN  100

{
        struct fd_set readmask;
        int     numfds;
        char    buf[BUFLEN];
        int     count;
        int     s;
        struct timeval {
                unsigned long   tv_sec;         /* seconds */
                long    tv_usec;        /* and microseconds */
        } timeout;

        memset (buf, 0, BUFLEN);

        /*  Notify operator of SIGIO interrupt */
        printf('' SIGIO interrupt received!\n\n'');
        /*  setup the masks */
        FD_ZERO(&readmask);
        FD_SET(ds, &readmask);
        FD_SET(ss, &readmask);

        /*  set the timeout value  */
        timeout.tv_sec = 0;
        timeout.tv_usec = 0;

        /*  select on socket descriptors */
        if ((numfds = select(ss + 1, &readmask, DONT_CARE,
            DONT_CARE, &timeout))  < 0) {
                perror(''select failed '');
                exit(1);
        }
        if (numfds == 0) {
                printf('' unexpected condition - investigate.\n'');
                exit(1);
        }
        if (FD_ISSET(ds, &readmask)) {
                /* a packet has come in, read it */
                count = recv(ds, buf, BUFLEN, 0);
                buf[count] = '\0';
                printf('' received a datagram packet of data: %s\n\n'', buf);
        }
        if (FD_ISSET(ss, &readmask)) {
                /* another program requests connection */
                s = accept(ss, &peeraddr_in, sizeof(struct sockaddr_in ));
                if (s == -1 ) {
                        perror('' accept call failed '');
                        exit(1);
                }
                printf('' accepted a connection request\n'');

                /*  Note that the following recv call will block
                        *   until data becomes available.  A real server
                        *   would probably include code to handle the
                        *   recv call asynchronously, thereby avoiding
                        *   this blockQ                    */
                count = recv (s, buf, BUFLEN, 0);
                if (count = -1) {
                        perror('' receive error'') ;
                        exit(1);
                }
                buf[count] = '\0';
                printf('' received a streams packet of data: %s\n\n'', buf);
                exit(0);
        }
} /* end io_handler */
```

```
/*
 *                       A S Y N C . C LN T
 *
 *        This is an example program that demonstrates the use
 *        of Asynchronous datagram and stream sockets.  This
 *        contains the client, and is intended to operate in
 *        conjunction with the server program found in async.serv.
 *        Together, these programs illustrate a very simple
 *        application of asynchronous sockets and therefore lacks
 *        the robustness of typical situations.  A program capable of
 *        handling all SIGIO interrupts requires substantial
 *        programmer investment and is beyond the scope of this
 *        example.
 *
 *        This program provides two services called ''sigex_udp''
 *        and ''sigex_tcp'', for datagram and streams, respectively.  In
 *        order for it to function, entries need to exist in the
 *        /etc/services file.  The port address for these services can be
 *        any port numbers that are likely to be unused, such as 22373
 *        and 22374, for example.  The host on which the client will
 *        be running must also have the same entries (same port numbers)
 *        in its /etc/services file.
 *
 *  ALGORITHM for Async.clnt:
 *
 *        Set up:
 *                Runstring
 *                Address Family
 *                Get the remote host's Internet address
 *        Datagram socket setup:
 *                Create the datagram socket
 *                Get the port address of desired service
 *        Send datagram data
 *        Sleep for 5 seconds
 *        Streams socket setup:
 *                Create the streams socket
 *                Get the port address of desired service
 *                Request a connection
 *        Send streams data
 *
 *
*********************************************************************/

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

int        ds;                              /* datagram socket descriptor */
int        ss;                              /* streams socket descriptor */

struct hostent *hp;              /* pointer to host info for remote host */
struct servent *sp;            /* pointer to service information */

struct sockaddr_in myaddr_in;   /* for local socket address */
struct sockaddr_in peeraddr_in; /* for peer socket address */




/*******************************************************************

 *                        M A I N
 *
 *******************************************************************/
main(argc, argv)
int     argc;
char    *argv[];
{
        int     addrlen;

        if (argc != 4) {
                fprintf(stderr,
                    ''Usage:  %s <remote host> <datagram data> <streams data> \n\n'',
argv[0]);
                exit(1);
        }

        /* Set up the peer address to which we will connect. */
        peeraddr_in.sin_family = AF_INET;

        /* Get the host information for the hostname that the
                 * user passed in.
                 */
        hp = gethostbyname (argv[1]);
        if (hp == NULL) {
                fprintf(stderr, ''%s: %s not found in /etc/hosts\n'',
                    argv[0], argv[1]);
                exit(1);
        }
        peeraddr_in.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))->s_addr;

        /*      Create the local datagram socket */
        ds = socket (AF_INET, SOCK_DGRAM, 0);
        if (ds == -1) {
                perror(argv[0]);
                fprintf(stderr, ''%s: unable to create datagram socket\n'', argv[0]);
                exit(1);
        }
```

```
/* Find the information for the ''sigex_udp '' server
         * in order to get the needed port number.
         */
sp = getservbyname (''sigex_udp'', ''udp'');
if (sp == NULL) {
        fprintf(stderr, ''%s: sigex_udp not found in /etc/services\n'',
            argv[0]);
        exit(1);
}
peeraddr_in.sin_port = sp->s_port;


/*  Send data over the datagram socket. This will
        *   cause the server process to recieve a SIGIO
        *   interrupt.  The server will use select to determine
        *   which of its sockets interrupted and then do the
        *   appropriate action.
        */
if (sendto(ds, argv[2], strlen(argv[2]), 0, &peeraddr_in,
    sizeof(struct sockaddr_in )) == -1)  {
        fprintf(stderr, ''%s: Datagram Sendto failed. '', argv[0]);
        perror(argv[0]);
        exit(1);
}
printf('' %s : Sent a datagram packet. \n'', argv[0]);

/*  This sleep simulates any processing that a real client
        *   might be do here.
        */
sleep(5);

/* Create the local streams socket. */
ss = socket (AF_INET, SOCK_STREAM, 0);
if (ss == -1) {
        perror(argv[0]);
        fprintf(stderr, ''%s: unable to create socket\n'', argv[0]);
        exit(1);
}

/*  Setup the address information for the remote
        *   streams socket.
        */
sp = getservbyname (''sigex_tcp'', ''tcp'');
if (sp == NULL) {
        fprintf(stderr, ''%s: sigex_tcp not found in /etc/services\n'',
            argv[0]);
        exit(1);
}
peeraddr_in.sin_port = sp->s_port;

/* Try to connect to the remote server at the address
        *    which was just built into peeraddr.
        *    This will cause another SIGIO interrupt of the
```

```
	 *    server.  When the server identifies the interrupting
	 *    socket (select(2)), it will accept the connection
	 *    request without blocking.
	 */
	if (connect(ss, &peeraddr_in, sizeof(struct sockaddr_in )) == -1) {
		perror(argv[0]);
		fprintf(stderr, ''%s: unable to connect to remote\n'', argv[0]);
		exit(1);
	}

	/*   Send data to the server over the stream socket.
	 *    This will cause yet another SIGIO interrupt of the
	 *    server.  In this case, the interrupt will be ignored
	 *    because the server is already waiting for data
	 *    (see async.serv).
	 */
	if (sendto(ss, argv[3], strlen(argv[3]), 0, &peeraddr_in,
	    sizeof(struct sockaddr_in )) == -1)  {
		fprintf(stderr, ''%s: Streams Sendto failed. '', argv[0]);
		perror(argv[0]);
		exit(1);
	}
	printf('' %s : Sent a streams packet. \n'', argv[0]);



	/* Print message indicating completion of task. */
	printf(''%s:Finished!\n'', argv[0]);
} /*  end main */
```

B

Portability Issues

This appendix describes implementation differences between 4.3 BSD IPC and HP-UX IPC. It contains porting issues for:

- IPC functions and library calls.
- Other functions and library calls typically used by IPC programs.

Because HP-UX IPC is based on 4.3 BSD IPC (it is a subset of 4.3 BSD), programs should port easily between HP-UX and 4.3 BSD systems. If you need to have portable applications, keep the information described in this appendix in mind when you write your IPC programs.

Porting Issues for IPC Functions and Library Calls

The following is a list of differences in IPC functions and library calls to watch out for if you want to port your IPC applications between HP-UX and 4.3 BSD systems.

Shutdown

When *shutdown* has been used on a datagram socket on an HP-UX system, the local port number bound to that socket remains unavailable for use until that socket has been destroyed by *close*.

Some other systems free that port number for use immediately after the *shutdown*. In general, sockets should be destroyed by *close* (or by terminating the process) when they are no longer needed. This allows you to avoid unnecessary delay in deallocating local port numbers.

Address Conversion Functions for DEC VAX Hosts

The functions *htonl*, *htons*, *ntonl* and *ntons* are not required on HP-UX systems. They are included for porting to a DEC VAX host. You can use these functions in your HP-UX programs for portability; they are defined as null macros on HP-UX systems, and are found in *netinet/in.h*.

FIONREAD Return Values

For HP-UX systems, the FIONREAD *ioctl* request on a datagram socket returns a number that may be larger than the number of bytes actually readable. Previously, HP-UX systems returned the maximum number of bytes that a subsequent *recv* would be able to return.

Listen's Backlog Parameter

HP-UX treats the *listen(2) backlog* value as the actual size of the queue for pending connections. Some implementations set their queue size to 3/2 * B+1, where B is the *backlog* value.

Pending Connections

There is no guarantee which pending connection on a listening socket is returned by *accept*. HP-UX systems return the newest pending connection. Applications should be written such that they do not depend upon connections being returned by *accept* on a first-come, first-served basis.

Porting Issues for Other Functions and Library Calls Typically Used by IPC

The following is a list of differences in functions and library calls to watch out for when you port your IPC applications between HP-UX and 4.3 BSD systems.

Ioctl and Fcntl Calls

4.3 BSD terminal *ioctl* calls are incompatible with the HP-UX implementation. These calls are typically used in virtual terminal applications. The HP-UX implementation uses UNIX System V compatible calls.

Pty Location

Look for the *pty* masters in */dev/ptym/ptyp?* and for the *pty* slaves in */dev/pty/ttyp?*. An alternative location to check is */dev*.

Utmp

The 4.3 BSD */etc/utmp* file format is incompatible with the HP-UX implementation. The HP-UX implementation uses UNIX System V compatible calls. Refer to the *utmp(4)* entry in the *HP-UX Reference Manual* for details.

Library Equivalencies

Certain commonly used library calls in 4.3 BSD are not present in HP-UX systems, but they do have HP-UX equivalents. To make code porting easier, use the following equivalent library calls. You can do this by putting them in an include file, or by adding the define statements (listed in the following table) to your code.

Table B-1. Definition of Library Equivalents

	4.2 BSD Library	**HP-UX Library**
#define	index(a,b)	strchr(a,b)
#define	rindex(a,b)	strrchr(a,b)
#define	bcmp(a,b,c)	memcmp(a,b,c)
#define	bcopy(a,b,c)	memcpy(b,a,c)
#define	bzero(a,b)	memset(a,0,b)
#define	getwd(a)	getcwd(a,MAXPATHLEN)

Note Include *string.h* before using *strchr* and *strrchr*. Include *sys/param.h* before using *getcwd*.

Signal Calls

Normal HP-UX *signal* calls are different from 4.3 BSD signals. See the *sigvector(2)* entry in the *HP-UX Reference Manual* for information on signal implementation. Note the following signal mapping.

Sprintf Return Value

For 4.3 BSD, *sprintf* returns a pointer to a string. For HP-UX systems, *sprintf* returns a count of the number of characters in the buffer.

Glossary

A

Address family: The address format used to interpret addresses specified in socket operations. The internet address family (AF_INET) is supported.

Address: An Interprocess Communication term that refers to the means of labeling a socket so that it is distinguishable from other sockets, and routes to that socket are able to be determined.

Advanced Research Projects Agency: A U.S. government research agency that was instrumental in developing and using the original ARPA Services on the ARPANET.

Alias: A term used to refer to alternate names for networks, hosts and protocols. This is also an internetwork mailing term that refers an alternate name for a recipient or list of recipients (a mailing list).

ARPA: See "Advanced Research Projects Agency."

ARPA/Berkeley Services: The set of services originally developed for use on the ARPANET (i.e., *telnet(1)*) or distributed with the Berkeley Software Distribution of UNIX, version 4.2 (i.e., *rlogin(1)*).

ARPANET: The Advanced Research Projects Agency Network.

Association: An Interprocess Communication connection (e.g., a socket) is defined by an association. An association contains the (protocol, local address, local port, remote address, remote port)-tuple. **Associations must be unique**; duplicate associations on the same system may not exist.

Asynchronous Sockets: Sockets set up via *ioctl* with the FIOASYNC option to be notified with a SIGIO signal whenever a change on the socket occurs. Primarily used for sending and receiving data without blocking.

B

Berkeley Software Distribution: A version of UNIX software released by the University of California at Berkeley.

Binding: Establishing the address of a socket which allows other sockets to connect to it or to send data to it.

BSD: See "Berkeley Software Distribution."

C

Channel: A communication path created by establishing a connection between sockets.

Client: A process that is requesting some service from another process.

Client host: The host on which a client process is running.

Communication domain: A set of properties that describes the characteristics of processes communicating through sockets. Only the Internet domain is supported.

Connection: A communications path to send and receive data. A connection is uniquely identified by the pair of sockets at either end of the connection. See also, "Association."

D

Daemon: A software process that runs continuously and provides services on request.

DARPA: See "Defense Advanced Research Projects Agency."

Datagram sockets: A socket that maintains record boundaries and treats data as individual messages rather than a stream of bytes. Messages may be sent to and received from many other datagram sockets. Datagram sockets do not support the concept of a connection. Messages could be lost or duplicated and may not arrive in the same sequence sent. Datagram sockets use the User Datagram Protocol.

Defense Advanced Research Projects Agency: The military arm of the Advanced Research Projects Agency. DARPA is instrumental in defining standards for ARPA services.

Domain: A set of allowable names or values. See also, "Communication domain."

F

File Transfer Protocol: The file transfer protocol that is traditionally used in ARPA networks. The *ftp* command uses the FTP protocol.

Forwarding: The process of forwarding a mail message to another destination (i.e., another user name, host name or network).

4.2 BSD: See "Berkeley Software Distribution."

Frame: See "Packet."

FTP: See "File Transfer Protocol."

G

Gateway: A node that connects two or more networks together and routes packets between those networks.

H

Host: A node that has primary functions other than switching data for the network.

I

International Standards Organization: Called "ISO," this organization created a network model that identifies the seven commonly-used protocol levels for networking.

Internet: All ARPA networks that are registered with the Network Information Center.

Internet address: A four-byte quantity that is distinct from a link-level address and is the network address of a computer node. This address identifies both which network is on the Internet and which host is on the network.

Internetwork: A term used to mean "among different physical networks."

Interprocess Communication: A facility that allows a process to communicate with another process on the same host or on a remote host. IPC provides system calls that access sockets. This facility is distinct from Bell System V IPC. See also, "Sockets."

IPC: See "Interprocess Communication."

ISO: See "International Standards Organization."

L

Link-level address: A six-byte quantity that is distinct from the internet address and is the unique address of the LAN interface card on each LAN.

M

Message: In IPC, the data sent in one UDP packet. When using sendmail a message is the information unit transferred by mail.

N

Node: A computer system that is attached to or is part of a computer network.

Node manager: The person who is responsible for managing the networking services on a specific node or host.

O

Official host name: The first host name in each entry in the */etc/hosts* file. The official host name cannot be an alias.

P

Packet: A data unit that is transmitted between processes. Also called a "frame."

Peer: An Interprocess Communication socket at the other end of a connection.

Port: An address within a host that is used to differentiate between multiple sockets with the same internet address.

Protocol: A set of conventions for transferring information between computers on a network (e.g., UDP or TCP).

R

Remote host: A computer that is accessible through the network or via a gateway.

Reserved port: A port number between 1 and 1023 that is only for super-user use.

S

Server: A process or host that performs operations that local or remote client hosts request.

Service: A facility that uses Interprocess Communication to perform remote functions for a user (e.g., *rlogin(1)* or *telnet(1)*).

Socket: Addressable entities that are at either end of an Interprocess Communication connection. A socket is identified by a socket descriptor. A program can write data to and read data from a socket, just as it writes and reads data to and from files.

Socket address: The internet address, port address and address family of a socket. The port and internet address combination allows the network to locate a socket.

Socket descriptor: An HP-UX file descriptor accessed for reading, writing or any standard file system calls after an Interprocess Communication connection is established. All Interprocess Communication system calls use socket descriptors as arguments.

Stream socket: A socket that, when connected to another stream socket, passes data as a byte stream (with no record boundaries). Data is guaranteed to arrive in the sequence sent. Stream sockets use the TCP protocol.

T

TCP: See "Transmission Control Protocol."

Telnet: A virtual terminal protocol traditionally used on ARPA networks that allows a user to log into a remote host. The *telnet* command uses the Telnet protocol.

Transmission Control Protocol: A protocol that provides the underlying communication support for AF_INET stream sockets. TCP is used to implement reliable, sequenced, flow-controlled two-way communication based on a stream of bytes similar to pipes.

U

UDP: See "User Datagram Protocol."

UNIX Domain Address: A character string containing the UNIX pathname to a UNIX Domain socket.

UNIX Domain Protocol: A protocol providing fast communication between processes executing on the same node and using the AF_UNIX socket address family.

User Datagram Protocol: A protocol that provides the underlying communication support for datagram sockets. UDP is an unreliable protocol. A process receiving messages on a datagram socket could find that messages are duplicated, out-of-sequence or missing. Messages retain their record boundaries and are sent as individually addressed packets. There is no concept of a connection between the communicating sockets.

V

Virtual Terminal Protocol: A protocol that provides terminal access to interactive services on remote hosts (e.g., *telnet(1)*).

Index

Printing History

New editions are complete revisions of the manual. Update packages, which are issued between editions, contain additional and replacement pages to be merged into the manual by the customer. The dates on the title page change only when a new edition or a new update is published. No information is incorporated into a reprinting unless it appears as a prior update; the edition does not change when an update is incorporated.

Note that many product updates and fixes do not require manual changes and, conversely, manual corrections may be done without accompanying product changes. Therefore, do not expect a one-to-one correspondence between product updates and manual updates.

Edition 1	February 1991
Edition 2	July 1992

PACKARD

Customer Order No.
98194-60531

Manufacturing No.
98194-90031
Mfg. number is for HP internal use on

98194-90031