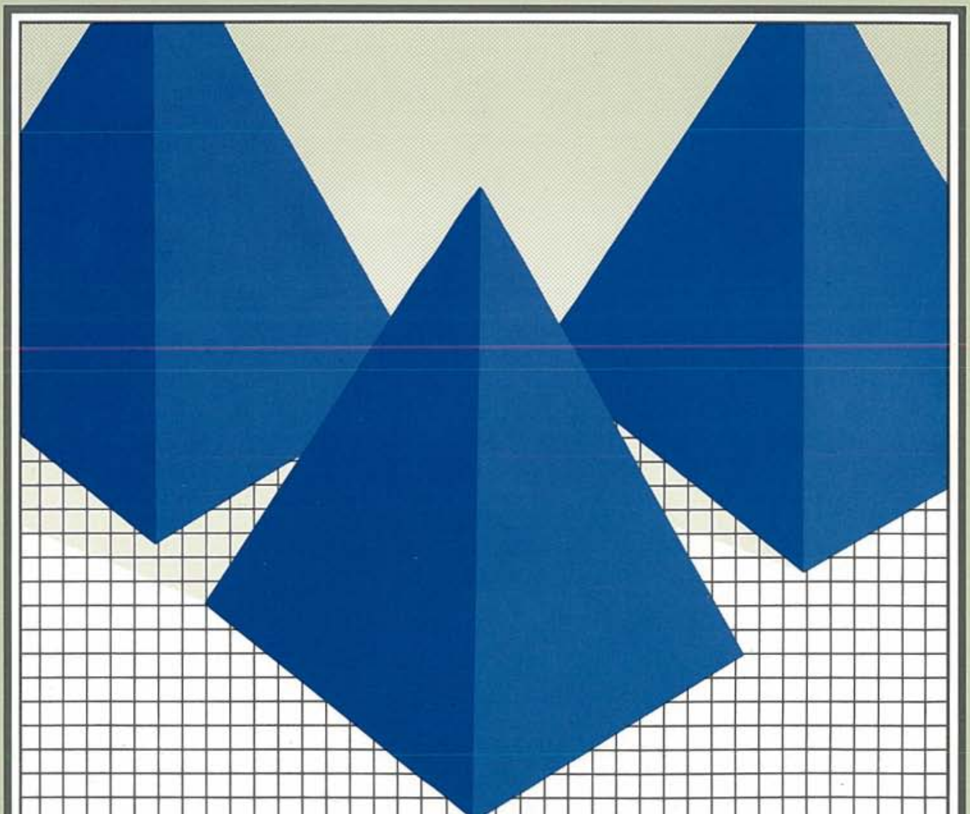
HEWLETT-PACKARD

## HP 9000 Series 300 Computers

# Using ARPA Services



# Using ARPA Services

HP 9000 Series 300

HEWLETT PACKARD

Manual Part Number: 50952-90001

Printed in U.S.A., January 1989

# Notice

UNIX® is a U.S. registered trademark of AT&T in the U.S.A. and in other countries.

NFS is a trademark of Sun Microsystems, Inc.

**Hewlett-Packard Co.**
**3404 E. Harmony Rd.**
**Fort Collins, CO 80525 U.S.A.**

# Printing History

December 1986 . . . Edition 1.

December 1987 . . . Edition 2.

January 1989 . . . Edition 3.

# Table of Contents

## Chapter 4: Listing Hosts with Ruptime



## Chapter 5: Listing Users with Rwho



## Chapter 6: Logging into a Host with Telnet

## Chapter 8: Transferring Files with Ftp

## Chapter 10: Executing Commands with Remsh



## Chapter 11: Interprocess Communication

**Appendix A: Portability Issues**



**Glossary**

**Index**

# 1

# Documentation Overview

**Note**

Before you read this manual, read the *Networking Overview: NS-ARPA and NFS Services* booklet for an introduction to important terms and concepts. The booklet positions HP 9000 Series 300 networking products relative to each other and lists and describes the components of each product. The booklet also contains specific network connectivity diagrams and a detailed documentation map.

## Manual Overview

### Who Should Read This Manual

This manual is written primarily for people who have:

- some experience with the HP-UX environment; and
- access to and familiarity with the *HP-UX Reference* manuals.

## What Is in This Manual

The list below briefly describes the contents of each chapter in this manual.

### Chapter 1: Introduction

The remainder of this chapter describes what you need to get started and provides a list of reference manuals.

### Chapter 2: Services Overview

To aid you in finding the service that best suits your needs, this chapter lists and briefly describes the ARPA/Berkeley Services according to their function. The "Services Overview" chapter also briefly describes the Interprocess Communication package and how to obtain general information about your system once the NS-ARPA Services/300 product has been installed.

### Chapter 3: Sending Mail

This chapter briefly describes the internetwork mail routing facility provided with the ARPA/Berkeley Services. Since this facility is automatically installed when the product is installed, but is **not** executable, your node manager may choose whether to make this facility executable on your local host. Plan to read this chapter only if your node manager has made this facility executable on your local host.

### Chapter 4: Listing Hosts with Ruptime

This chapter explains how to list the names and condition of network hosts. The chapter also explains how to sort the list based on various items in the list.

### Chapter 5: Listing Users with Rwho

This chapter explains how to list information about users logged into network hosts.

### Chapter 6: Logging Into a Host with Telnet

This chapter explains how to use *telnet* to log into a remote host.

### Chapter 7: Logging Into a Host with Rlogin

This chapter explains how to use *rlogin* to log into a remote host. It also describes how to give other network users *rlogin* access to your local account.

### Chapter 8: Transferring Files with Ftp

This chapter explains how to use *ftp* to transfer files between your local host and a remote host and to perform remote file management operations.

### Chapter 9: Transferring Files with Rcp

This chapter explains how to use *rcp* to transfer files and directories among network hosts. The chapter also describes how to give other network users *rcp* access to your local account.

### Chapter 10: Executing Commands with Remsh

This chapter explains how to use *remsh* to execute a command on a remote host. The chapter also describes how to give other network users *remsh* access to your local account.

### Chapter 11: Interprocess Communication

This chapter is for programmers who intend to use Interprocess Communication (IPC) based on 4.2 BSD programming development tools. The chapter describes how to use datagram and stream sockets. An overview of IPC and example programs are also included.

### Appendix A: Portability Issues

This appendix explains portability issues and differences between HP's implementation of 4.2 BSD Interprocess Communication (sockets) and the Berkeley sockets.

### Glossary

The glossary lists and defines terms used in this manual.

### Index

The index provides a quick page-reference for subjects contained within this manual.

## Conventions in This Manual

| Notation | Description |
|---|---|
| **boldface** | **Boldfacing** is used for emphasis. |
| `computer_font` | Words in syntax statements that are not in italics must be entered exactly as shown. Punctuation characters other than brackets, braces and ellipses must also be entered exactly as shown. |
| *italics* | Words in syntax statements that are in italics denote a parameter that you must supply. |
| **Return** | This font is used to indicate a key on the computer's keyboard. |
| **CTRL-D** | This convention is used to indicate a combination of keys to press simultaneously for a desired function. |
| ... | An ellipsis in a syntax statement indicates that a previous element may be repeated. In addition, vertical and horizontal ellipses are used in examples to indicate that portions of the example have been omitted. |
| `-l` | This is a command argument that appears frequently in the manual. We mention it here so you will not confuse the letter "l" with the number "1." |

## Reference Manual Guide

For more information on the following subjects, refer to the publications listed in the right column.

| For information on: | Read: |
|---|---|
| HP-UX Operating System (HP 9000) | *Introducing UNIX System V*<br>*HP-UX Concepts and Tutorials*<br>*Beginner's Guide* series for HP-UX<br>*HP-UX Reference* manuals |
| ARPA/Berkeley Manual Reference Pages | *ARPA/Berkeley Services Reference Pages* |
| C Programming Language | *The C Programming Language*, Brian W. Kernighan, Dennis M. Ritchie; Prentice-Hall, Inc. |
| | *C Programming Guide*, Jack Purdum, Que Corporation, Indianapolis |
| Networking | *Computer Networks*, Andrew S. Tanenbaum; Prentice-Hall, Inc. |
| | *Networking Overview: NS-ARPA and NFS Services and X.25* |
| NS Part of the NS-ARPA Services/300 Product | *Using Network Services* |
| NS-ARPA Services/300 Installation, Configuration, Maintenance, and Troubleshooting | *Installing and Maintaining NS-ARPA Services* |

# 2

# Services Overview

## Introduction

The ARPA Services part of the NS-ARPA Services/300 product enables your HP 9000 Series 300 to transfer files, log into remote hosts, execute commands remotely, and send mail to and receive mail from remote hosts that are either on your network or accessible by your network.

The ARPA Services part of the NS-ARPA Services/300 product is a subset of networking services originally developed by the University of California at Berkeley (UCB) for the Advanced Research Projects Agency (ARPA) and UCB. The services originally developed for ARPA are called "ARPA Services." The services originally developed for UCB are called "Berkeley Services."

UCB developed the services based on the Berkeley Software Distribution of UNIX[1], version 4.2 (4.2 BSD).

4.2 BSD programming development tools for interprocess communication are also provided with the ARPA Services part of the NS-ARPA Services/300 product.

This chapter briefly explains the ARPA/Berkeley Services. To guide you to the service you need for a desired task, the services are listed by functionality. For tutorial information about individual services, read this manual. For specific details about individual services, read the reference page for the service.

---

(1) UNIX is a U.S. registered trademark of AT&T in the U.S.A. and other countries.

This chapter also lists and describes:

- the sources from which you can obtain additional information about your local host or network; and
- the 4.2 BSD-based ARPA Services/300 Interprocess Communication package.

## Getting Started

Before you begin, make sure that:

- your node manager has installed the NS-ARPA Services/300 product on your local host and has brought up the network;
- 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 Series 300 can communicate with.

---

**Note**

The computer you are working on is referred to as your **local host**, and all other computers (hosts) on the network are remote in relation to your local host.

---

# The ARPA/Berkeley Services

HP's implementation of the ARPA/Berkeley Services is actually a combination of services originating from the Advanced Research Projects Agency (ARPA) and from the University of California at Berkeley (UCB).

The services originating from the ARPA environment are used to communicate in HP-UX, UNIX and non-UNIX environments.

Services originating from UCB are used for HP-UX or UNIX operations only.

The services from each environment are shown in the figure below.

**ARPA/Berkeley Services**

| ARPA Services | Berkeley Services |
|---|---|
| File Transfer Protocol (*ftp*) | Remote copy (*rcp*) |
| Telnet (*telnet*) | Remote login (*rlogin*) |
| Simple Mail Transfer Protocol (SMTP) | Remote execution (*rexec*) |
| | Remote shell (*remsh*) |
| | Remote uptime (*ruptime*) |
| | Remote who (*rwho*) |
| | Internetwork Mail Routing (*sendmail*) |
| | Interprocess Communication ("IPC" or "Berkeley sockets") |

# Services Listed by Function

The number in parentheses next to the service name, e.g., *rlogin(1)*, corresponds to the section in the *ARPA/Berkeley Services Reference Pages* that documents the service.

## Sending Mail to a Remote Host

*sendmail(1M)* originates from UCB and, when installed, works with your network's mailers to perform internetwork mail routing among HP-UX, UNIX and non-UNIX hosts on the network. When used in the command line, *sendmail* does not provide a friendly user interface. *Sendmail* supports mail aliasing and forwarding and uses ARPA's standard Simple Mail Transfer Protocol (SMTP).

## Listing Information about a Remote Host

*ruptime(1)* is a Berkeley Service. It is used to list information about HP-UX or UNIX hosts on the network that are running the *rwho* daemon, *rwhod*. The information that *ruptime* displays includes host names, whether the hosts are up or down, the number of active users on the remote host and three numeric fields containing the 1-, 5- and 15-minute load averages for the number of processes in the remote host's run queue.

*rwho(1)* is a Berkeley Service. It is used to list information about HP-UX or UNIX hosts on the network that are running the *rwho* daemon, *rwhod*. The information that *rwho* displays includes the user names of those who are actively logged into remote or local hosts on the network, the remote or local hosts' names, the users' terminal lines, the users' login times, and the amount of time each user has been idle.

## Logging into a Remote Host

*Rlogin* and *telnet* allow you to log into a remote host on the network if you have an account on the remote host.

*telnet(1)* originates from the ARPA environment and is used to log into a remote, HP-UX, UNIX or non-UNIX host. You **must** use *telnet* if the remote host is a non-UNIX host (i.e., VAX/VMS) and you must supply a login name and password to log into the remote host.

*rlogin(1)* originates from UCB and is used to log in from a local HP-UX or UNIX host to a remote HP-UX or UNIX host without being prompted for a login name and password.

## Transferring Files to or from a Remote Host

*ftp(1)* originates from the ARPA environment and allows you to transfer files among HP-UX, UNIX and non-UNIX hosts on the network. For example, if you want to transfer a UNIX file to a remote host using an MS-DOS format, you would use *ftp*. *Ftp* is the file transfer program which uses the ARPA standard File Transfer Protocol (FTP).

*rcp(1)* originates from UCB and allows you to transfer files between only HP-UX or UNIX hosts on the network.

## Executing Commands on a Remote Host

*remsh(1)* originates from UCB and allows you to execute commands on a remote HP-UX or UNIX host on the network. *Remsh* is the same command as *rsh* from the Berkeley Software Distribution of UNIX, version 4.2 (4.2 BSD). No login names or passwords are used although your account permissions are verified on the remote host before you can remotely execute a command.

*rexec(3X)* originates from UCB and is a library routine used to execute commands on a remote HP-UX or UNIX host on the network. A customized program must be written to use *rexec*. The advantage of using *rexec* is that it can be used in programs and passwords can be specified.

# Obtaining General Information

This section describes the sources from which you can obtain additional information.

| | |
|---|---|
| *hosts(4)* | The */etc/hosts* file contains host names and internet addresses of remote hosts. You may not be limited to communicating only with those hosts listed in */etc/hosts*. Check with your node manager to verify which hosts are available for communication with your local host. |
| *hosts.equiv(4)* | The */etc/hosts.equiv* file lists remote hosts on the network that are "equivalent" to your local host. |
| *intro(3N)* | The *intro* HP-UX reference page lists and briefly describes the Internet network library functions. |
| *netrc(4)* | The *netrc(4)* reference page describes the *.netrc* file used by *rexec* and *ftp* to determine login names and passwords to remote hosts. |
| *networks(4)* | The */etc/networks* file lists the official network name, number and aliases for networks that your local host recognizes. |

| | |
|---|---|
| *$HOME/.rhosts* | The *$HOME/.rhosts* file may be created by each user on the local host to specify remote user names that are equivalent to the local user. The local host then permits equivalent remote users to access the local user's account without requiring a password. *$HOME/.rhosts* is described on the *hosts.equiv(4)* reference page. |
| *protocols(4)* | The */etc/protocols* file contains the official protocol names, protocol numbers and protocol aliases recognized by your local host. |
| *services(4)* | The */etc/services* file contains the official service names, port numbers, protocol names used by the service and service name aliases that correspond to the service. |

# Interprocess Communication

The NS-ARPA Services/300 product provides programmers with an Interprocess Communication (IPC) package that allows processes to communicate with other local and remote processes through system calls. HP's IPC implementation is based on the IPC in 4.2 BSD.

Two transport protocols are available:

- Transmission Control Protocol (TCP), which is the transport protocol for stream sockets, and
- User Datagram Protocol (UDP), which is the transport protocol for datagram sockets.

For details on IPC, refer to the "Interprocess Communication" chapter in this manual.

# 3

# Sending Mail

## Introduction

*Sendmail(1M)*, the internetwork mailing facility supplied with the ARPA/Berkeley Services, acts as a central post office that determines the internetwork routing needed for mail delivery to local or remote users. It routes messages to local users, files and programs. *Sendmail* also enables your local host to send mail to and receive mail from other hosts on a local area network or through a gateway. In addition, message aliasing and forwarding can also be specified.

Because *sendmail* is typically used in environments where internetwork communications are frequent or heavy, your node manager **may** have installed *sendmail* on your system. Ask your node manager if *sendmail* has been installed on your system before you continue reading this section.

---

**Note**

For details about *sendmail* and message aliasing and forwarding, refer to the *Installing and Maintaining NS-ARPA Services* manual.

---

# Using Sendmail

## Executing Sendmail

You can execute *sendmail* in two ways:

- Whenever a standard mailer is accessed, *sendmail* is automatically invoked. Standard HP-UX mailing programs are *mail* and *mailx*.
- You can use the *sendmail* command with arguments on the command line. Because this method does **not** provide a friendly user interface, it is typically used only in programs.

## Mailing to Files

If you want to send a message to a local file, you must specify the filename as an absolute path (i.e., you must begin the filename with a slash "/").

If the file does not exist, you must own and have search (execute) permission in the directory in which the file is to be created.

If the file already exists, is not executable and is writable by all users, the message will be appended to the file.

---

**Note**

*Sendmail* does not write to executable files.

---

# Sendmail Operations Overview

*Sendmail* performs its task in two phases: it collects messages and then routes them. While collecting and routing messages, message-address interpretation is controlled by a production system that manages both network-style addressing (e.g., user@host) and UUCP-style addressing (e.g., host!user). This production system is defined by the contents of the *sendmail* configuration file.

## Collecting Messages

If *sendmail* is invoked via a standard mail program or by using the *sendmail* command on the command line, it collects the message from *stdin* and the argument list.

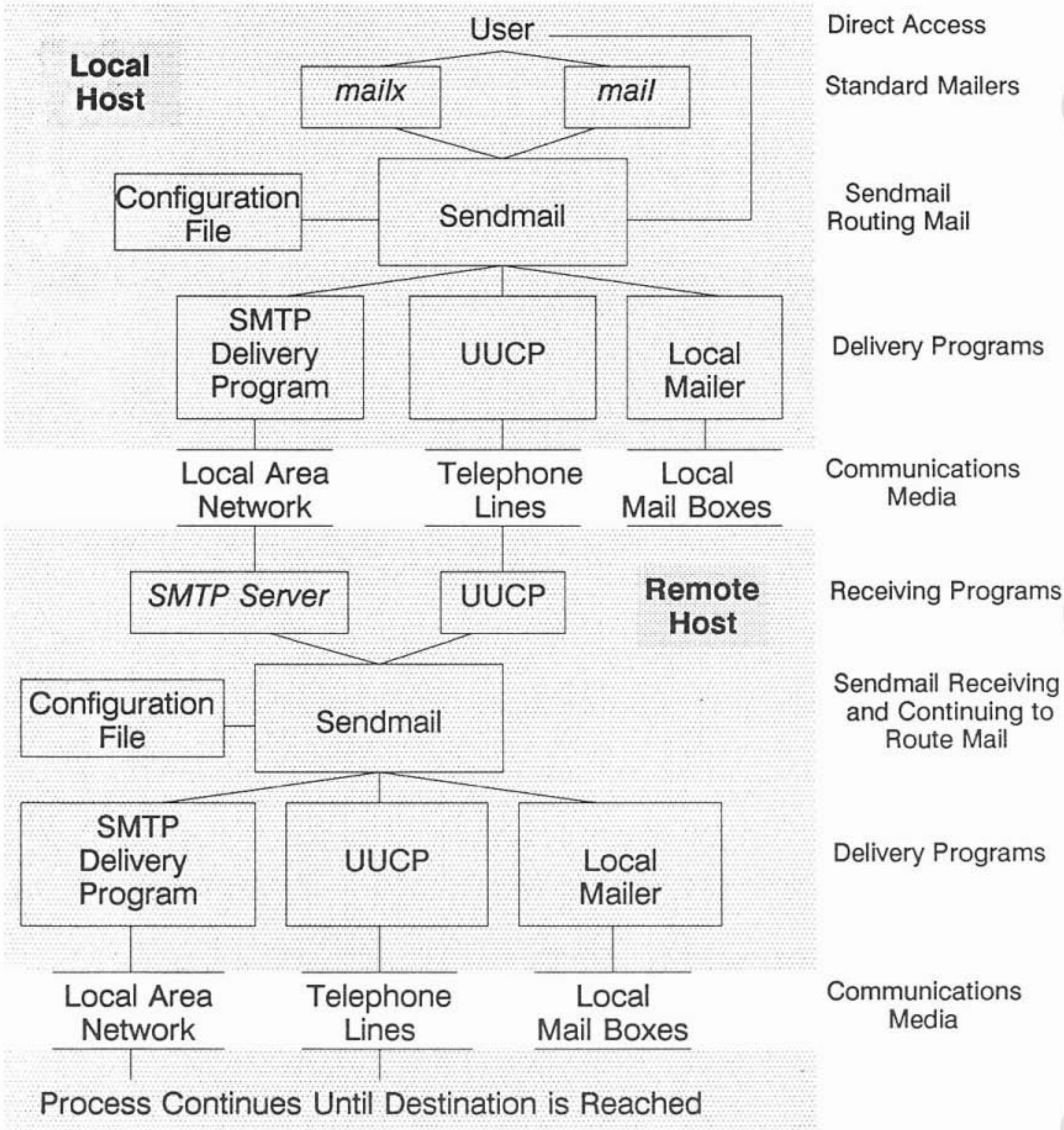
## Routing the Messages

Once *sendmail* collects the message, it routes the information. To route the message, *sendmail*:

- rewrites the mail addresses of the recipients to conform to the standards of the target network;
- if necessary, adds lines to the message header so that the information is available for a recipient to use in a reply; and
- passes the mail to one of several specialized delivery agents for delivery.

*Sendmail* is also executed by the program that receives mail from the network. When incoming mail arrives, a receiving program passes the mail to *sendmail* for routing in the same way that a mailer invokes *sendmail*.

The figure below outlines the flow of messages through *sendmail*.




**Flow of Mail Through Sendmail**

### Routing to Remote Hosts

If a recipient's host is on the LAN, *sendmail* uses the SMTP delivery module to send the message to the remote host on the network.

If the recipient has a UUCP address, *sendmail* calls the *uux* program to deliver the message on the remote system.

### Routing to Local Destinations

If the recipient is a local user, *sendmail* calls the local mailer to deposit the message in the recipient's mailbox. The HP-UX local mailer is *rmail*.

If the recipient is a local file, *sendmail* writes the message to the file. This is the only case in which *sendmail* directly **delivers** a message to a destination.

### Routing Error Messages

During mail transfer processes, *sendmail* creates a transcript of each mail transaction to send to the originator if the message is permanently undeliverable. This transcript contains any error messages that occur during the attempted mail delivery.

If an error status indicates that the delivery failed but might be successful if re-tried, *sendmail* stores the message on a queue for later delivery. *Sendmail* attempts to send the message again when it next processes the queue.

## More Mail System Information

If *sendmail* is installed, the commands listed below or the corresponding HP-UX reference pages provide additional information about the *sendmail* program or your mail system.

| | |
|---|---|
| *mailq* | is described on the *sendmail(1M)* reference page and prints a list of mail messages that are in the mail queue. |
| *mailstats(1)* | prints the mail traffic statistics. |
| *praliases(1)* | displays any aliases that your system recognizes. |
| *uupath(1)* | expands UUCP-style and network-style addresses. |

# 4

# Listing Hosts with Ruptime

*Ruptime* is a Berkeley Service that lists information about HP-UX or UNIX hosts on the network. The status information that *ruptime* displays includes:

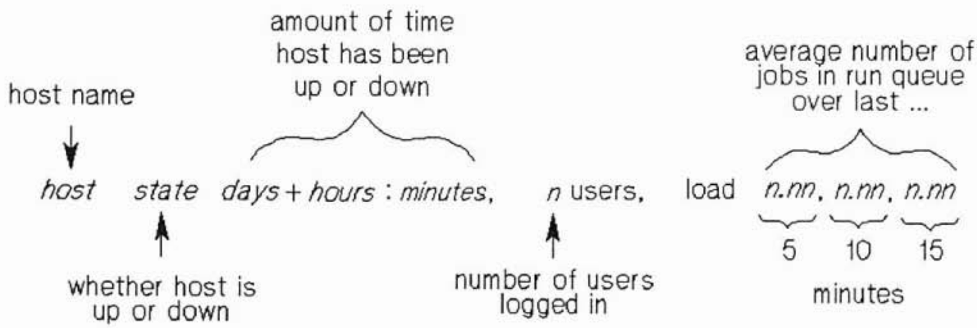
- network host names,
- whether each network host is up or down,
- the number of active users on each network host, and
- the average number of jobs in each network host's run queue over the last one, five, and fifteen minutes.

This information is useful in determining:

- which network hosts you can perform work on,
- which network hosts are most heavily or least heavily loaded, and
- how responsive each network host is likely to be over the network.

# Using Ruptime

For each network host, *ruptime* displays a status line with the following format:




**Ruptime Status Line**

---

**Note**

*Ruptime* does not count users who have not used the system for an hour or more. You can have *ruptime* count these idle users by invoking the command with the **–a** option as shown in the "Displaying Ruptime Status Lines" section.

---

Ruptime can display status lines (with or without idle users) sorted by:

- host name in alphabetical order,
- host name in reverse alphabetical order,
- decreasing uptime,
- increasing uptime,
- decreasing number of users,
- increasing number of users,
- decreasing load average, and
- increasing load average.

The following sections tell how to display *ruptime* status lines sorted in the ways described above.

# Displaying Ruptime Status Lines

## Sorted by Host Name in Alphabetical Order

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcc        up  1+17:40,     4 users,   load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,   load 0.10, 0.38, 0.49
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcc        up  1+17:40,     6 users,   load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,   load 0.10, 0.38, 0.49
```

## Sorted by Host Name in Reverse Alphabetical Order

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -r
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcd      up 14+06:49,    3 users,  load 0.10, 0.38, 0.49

hpabcc      up  1+17:40,    4 users,  load 0.18, 0.13, 0.09

hpabcb    down      1:13

hpabca    down 14+08:34
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -r
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcd      up 14+06:49,    3 users,  load 0.10, 0.38, 0.49

hpabcc      up  1+17:40,    6 users,  load 0.18, 0.13, 0.09

hpabcb    down      1:13

hpabca    down 14+08:34
```

## Sorted by Decreasing Uptime

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -t
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcd       up 14+06:49,     3 users,  load 0.10, 0.38, 0.49

hpabcc       up  1+17:40,     4 users,  load 0.18, 0.13, 0.09

hpabcb     down      1:13

hpabca     down 14+08:34
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -t
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcd       up 14+06:49,     3 users,  load 0.10, 0.38, 0.49

hpabcc       up  1+17:40,     6 users,  load 0.18, 0.13, 0.09

hpabcb     down      1:13

hpabca     down 14+08:34
```

## Sorted by Increasing Uptime

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -r -t
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down      1:13
hpabcc        up  1+17:40,     4 users,  load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -r -t
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down      1:13
hpabcc        up  1+17:40,     6 users,  load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
```

## Sorted by Decreasing Number of Users

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -u
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcc        up  1+17:40,     4 users,  load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
hpabcb      down      1:13
hpabca      down 14+08:34
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -u
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcc        up  1+17:40,     6 users,  load 0.18, 0.13, 0.09
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
hpabcb      down      1:13
hpabca      down 14+08:34
```

## Sorted by Increasing Number of Users

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -r -u
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcd        up 14+06:49,    3 users,  load 0.10, 0.38, 0.49
hpabcc        up  1+17:40,    4 users,  load 0.18, 0.13, 0.09
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -r -u
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcd        up 14+06:49,    3 users,  load 0.10, 0.38, 0.49
hpabcc        up  1+17:40,    6 users,  load 0.18, 0.13, 0.09
```

## Sorted by Decreasing Load Average

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -l
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcc      up  1+17:40,    4 users,  load 0.18, 0.13, 0.09
hpabcd      up 14+06:49,    3 users,  load 0.10, 0.38, 0.49
hpabcb    down      1:13
hpabca    down 14+08:34
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -l
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabcc      up  1+17:40,    6 users,  load 0.18, 0.13, 0.09
hpabcd      up 14+06:49,    3 users,  load 0.10, 0.38, 0.49
hpabcb    down      1:13
hpabca    down 14+08:34
```

## Sorted by Increasing Load Average

### Excluding Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -r -l
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
hpabcc        up  1+17:40,     4 users,  load 0.18, 0.13, 0.09
```

### Including Idle Users

**At your HP-UX prompt, enter:**

```
ruptime -a -r -l
```

**Result:**

*Ruptime* displays a list similar to the following:

```
hpabca      down 14+08:34
hpabcb      down     1:13
hpabcd        up 14+06:49,     3 users,  load 0.10, 0.38, 0.49
hpabcc        up  1+17:40,     6 users,  load 0.18, 0.13, 0.09
```

# 5

# Listing Users with Rwho

*Rwho* is a Berkeley Service that lists information about HP-UX or UNIX hosts on the network. The information that *rwho* displays includes:

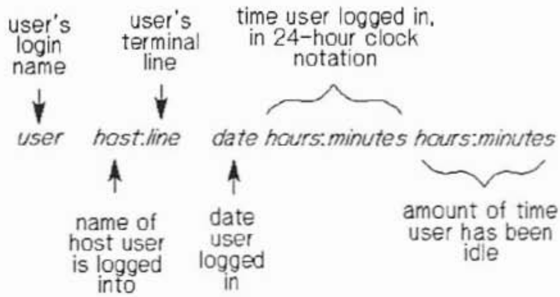
- the login name of each user who is logged into a host on the network,
- the name of the host each user is logged into,
- each user's terminal line,
- the date and time each user logged in, and
- the amount of time (if any) each user has been idle (has not used the system for one minute or more).

This information is useful in determining:

- who is logged into the hosts on the network and
- who is likely to be at their terminal or workstation.

# Using Rwho

For each user logged into a network host, *rwho* displays an information line with the following format:




**Rwho Status Line**

*Rwho* reports information in the following way:

| If | Then rwho |
|---|---|
| a user has not used the system for one to fifty-nine minutes, | reports the amount of idle time for that user. |
| a user has not used the system for one hour or more, | omits the user from its list, unless you invoke *rwho* with the **-a** option. |
| a host has not broadcast an *rwho* status report to the network lately, | assumes that the host is down and does not list any users on that host. |

**Note**

*Rwho*'s list of users can become excessively long when the number of users on network hosts becomes large.

With *rwho*, you can list either:

- users on network hosts who are active or who have been idle for less than one hour or
- all users logged into network hosts, regardless of the amount of time any of them have been idle.

## Listing Active and Likely Active Users of Network Hosts

**At your HP-UX prompt, enter:**

```
rwho
```

**Result:**

*Rwho* displays a list similar to the following:

```
acb     hpabcd:ttyp3    Jun  2 08:32 :19
bjt     hpabcf:tty3p3   Jun  2 09:35     ←— Active User
chas    hpabcd:tty3p3   Jun  2 07:47 :27
cjc     hpabcd:tty1p2   Jun  2 07:55     ←— Active User
dae     hpabcf:ttyp2    Jun  2 08:28 :57
```

## Listing All Users of Network Hosts

**At your HP-UX prompt, enter:**

```
rwho -a
```

**Result:**

*Rwho* displays a list similar to the following:

```
acb    hpabcd:ttyp3     Jun  2 08:32   :19
bjt    hpabcf:tty3p3    Jun  2 09:35  <---  Active User
chas   hpabcd:tty3p3    Jun  2 07:47   :27
cjc    hpabcd:tty1p2    Jun  2 07:55  <---  Active User
dae    hpabcf:ttyp2     Jun  2 08:28   :57
gen    hpabcd:ttyp4     Jun  2 08:45  5:59
kgj    hpabcd:ttyp0     Jun  2 08:09  1:02
scb    hpabce:tty3p1    Jun  2 12:12  3:24
```

# 6

# Logging into a Host with Telnet

*Telnet* is an ARPA Service that you use to log into a remote HP-UX, UNIX, or non-UNIX host. *Telnet* is the virtual terminal program that uses the ARPA standard TELNET protocol.

## Using Telnet

To use *telnet* you:

- invoke *telnet*,
- change the *telnet* escape character if necessary,
- connect to a remote host, and then
- log into that host.

After that, you can use *telnet* to do work on the remote host as if your terminal or workstation were physically connected to that host.

## 1. Invoke Telnet

**At your HP-UX prompt, enter:**

```
telnet
```

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

### Telnet's Two States

*Telnet* has two states: input state and command state. The `telnet>` prompt means that *telnet* is in its command state. *Telnet*'s command state allows you to execute individual *telnet* commands to get help, get status information, change characteristics of your *telnet* session or exit from *telnet*.

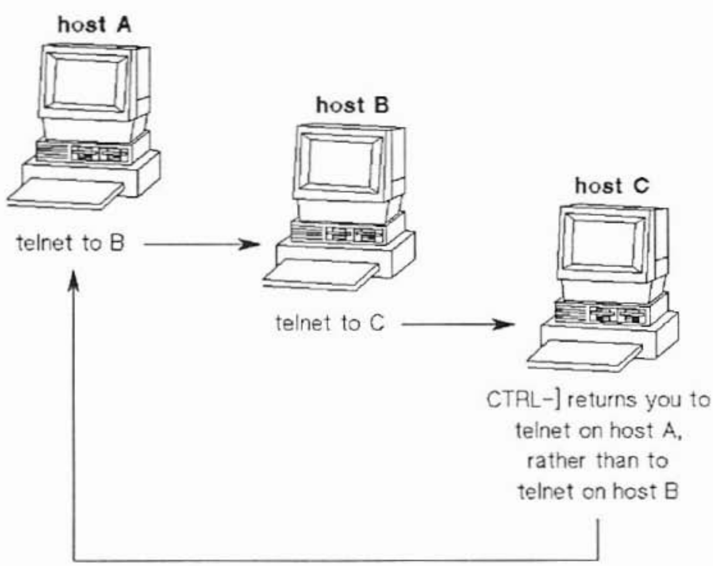
Connecting to and logging into a remote host puts *telnet* into its input state. Everything you type, with the exception of the *telnet* escape character, goes to the remote host. When *telnet* is in its input state, you can do work on the remote host as if it were your local host.

When *telnet* is in its input state, entering the *telnet* escape character puts *telnet* back into its command state. After you execute a *telnet* command, *telnet* returns to its input state.

## 2. Change the Telnet Escape Character If Necessary

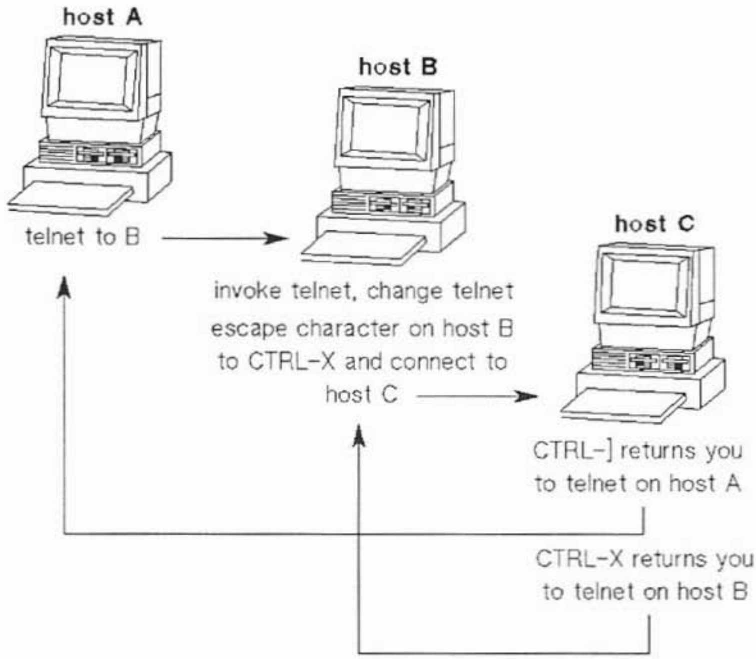
When you connect to a remote host, *telnet* presets its escape character to **CTRL–]** (sometimes shown as ^]). You need to set the *telnet* escape character to something else if:

- the character **CTRL-]** performs a particular function for a program within which you are running *telnet*,
- that character **CTRL-]** performs a particular function for a program you plan to run from within *telnet*, or
- you want to be able to return to an intermediate remote host when you nest a series of *telnet* commands over a chain of remote hosts. This last case is illustrated below:




**Nested Telnet Commands**

If you want to return to *telnet* on host B, you must change the escape character for *telnet* on host B when you invoke *telnet* to connect to host C:




**Nested Telnet Commands with Different Escape Characters**

You can change the *telnet* escape character with *telnet*'s *escape* command.

## Caution: What Not to Change the Telnet Escape Character To

Do **not** change the *telnet* escape character to:

- a character with a particular function in a program you may run from within *telnet* or
- a character that may conflict with your terminal configuration. HP-UX associates certain characters with specific functions. These characters are listed in the following table.

| Function | Predefined Character in HP-UX |
| --- | --- |
| You can change these characters, so they may be different in your terminal configuration: | |
| End of File | **CTRL-D** (EOT) |
| Interrupt | **CTRL-C** (DEL or Rubout) |
| Quit | **CTRL-** (FS) |
| Erase | **CTRL-H** (#) |
| Kill | **CTRL-U** (@) |
| End of Line | **CTRL-@** (NUL) |
| Shell Layers Switch | **CTRL-Z** |
| You can **not** change these characters in your terminal configuration: | |
| New Line/Line Feed | **CTRL-J** (LF) |
| Stop or XOFF (Transmit Off) | **CTRL-S** (DC3) |
| Start or XON (Transmit On) | **CTRL-Q** (DC1) |
| Enquire | **CTRL-E** (ENQ) |
| Acknowledge | **CTRL-F** (ACK) |

To find out what characters are used in your terminal configuration, at your HP-UX prompt, enter:

```
stty -a
```

## Changing the Telnet Escape Character

**At the telnet > prompt, enter:**

`escape` *new_telnet_escape_character*

where *new_telnet_escape_character* is the character you want to change the *telnet* escape character to.

---

**Note**

If *new_telnet_escape_character* is a control character, you must enter it as ^*character* where ^ represents **CTRL**. (For example, you would enter **CTRL–X** as ^X.)

---

**Example Entry:**

```
escape ^[
```

**Result:**

*Telnet* changes its escape character to the character you specify and displays:

`Escape character is '`*new_telnet_escape_character*`'.`

If you are **not** connected to a remote host, *telnet* then redisplays its `telnet>` prompt.

If you are connected to a remote host, *telnet* returns you to the remote host.

**Note**

When connected to a remote host, press **Return** to redisplay the remote host's prompt.

## 3. Connect to a Remote Host

**At the telnet > prompt, enter:**

`open` *remote_host*

where *remote_host* is the name or alias of a host listed in */etc/hosts*.

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address    official_name    alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

To connect to a host **not** listed in */etc/hosts*, you can give the host's internet address where *remote_host* appears above. The internet address must be in dot notation (for example, 192.6.21.9).

**Example Entry:**

```
open hpabsa
```

**Result:**

*Telnet* connects you to the remote host and prompts for a remote login name, displaying:

```
Trying...

Connected to remote_host.

Escape character is '^]'.

remote_host_identification_message

remote_host_login_prompt
```

## 4. Log into the Remote Host

---

**Note**

You must be able to supply *telnet* with a valid login name and password (if required) on the remote host.

---

**Log into the remote host by supplying a valid remote login name and password, if required. The login name and password you supply may be yours or someone else's.**

**Result:**

If the login name and password you supplied are valid on the remote host, the remote host logs you in and displays its login message and its prompt.

# Giving Telnet Commands When Telnet Is in Its Input State

To give a command to *telnet* from its input state, you need to enter the *telnet* escape character. This character tells *telnet* that what follows is a *telnet* command, **not** information you are sending to the remote host.

The preset *telnet* escape character is **CTRL–]** (sometimes shown as ^]). (You may have changed this to something else, as explained earlier in this chapter.)

When you enter the *telnet* escape character, *telnet* responds with its `telnet>` prompt. The `telnet>` prompt lets you know that *telnet* is ready to accept a command. (*Telnet* is in its command state.)

You can execute only one *telnet* command at a time; *telnet* returns to its input state after each command completes.

The remaining sections of this chapter discuss what *telnet* commands are available in addition to *escape* and *open*.

# Checking the Behavior of Carriage Returns from a Remote Host

When some remote hosts send a carriage return to your local host, your local host may need to change the carriage return into a carriage return-line feed combination. *Telnet*'s *crmod* command allows you to enable or disable this behavior.

The following behavior indicates that *telnet*'s carriage return mode setting is wrong for the type of remote host you are communicating with:

- If pressing **Return** produces double-spaced lines (indicating an extra line feed), you need to **disable** carriage return mode.
- If pressing **Return** moves the cursor to the beginning of the same line so that the same line keeps getting overwritten (indicating no line feed), you need to **enable** carriage return mode.

## Changing the Carriage Return Mode Setting

1. **If you are not already at the telnet > prompt, enter the *telnet* escape character.**

   **Result:**

   *Telnet* displays its prompt:

   ```
   telnet>
   ```

2. **At the telnet > prompt, enter:**

   ```
   crmod
   ```

**Result:**

If carriage return mode was on, *telnet* turns it off and displays:

```
Wont map carriage return on output.
```

If carriage return mode was off, *telnet* turns it on and displays:

```
Will map carriage return on output.
```

If you are connected to a remote host, *telnet* returns you to the remote host.

---

**Note**

To redisplay the remote host's prompt, press **Return**.

---

If you are **not** connected to a remote host, *telnet* redisplays its `telnet>` prompt.

## Disconnecting from a Remote Host and/or Exiting Telnet

You can disconnect from a remote host but remain in *telnet* if you connected to the remote host with *telnet*'s *open* command. This is useful if you want to connect to other remote hosts during the same *telnet* session.

If you want to exit from *telnet* and return to HP-UX on your local host, there are two ways to do so, depending on which state (input or command) *telnet* is in. Exiting from *telnet* disconnects from a remote host if a connection exists.

## Disconnecting from a Remote Host and Remaining in Telnet

---

**Note**

This is possible only if you connected to the remote host with *telnet*'s *open* command.

---

**1. If you are not at the telnet > prompt, enter the *telnet* escape character.**

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

**2. At the telnet > prompt, enter:**

```
close
```

**Result:**

If you connected to the remote host with *telnet*'s *open* command, *telnet* disconnects from the remote host and returns the `telnet>` prompt, displaying:

```
Connection closed.
```

```
telnet>
```

If you connected to the remote host when you invoked *telnet* (as explained later in this chapter), *telnet* disconnects from the remote host and returns you to HP-UX on your local host. (You exit from *telnet*.) *Telnet* displays:

```
Connection closed.
```

*local_HP-UX_prompt*

If no connection exists to a remote host, the *close* command has no effect. *Telnet* just redisplays its prompt:

```
telnet>
```

## Exiting from Telnet When Telnet Is in Its Input State

**Log out of the remote host as you normally would (such as with CTRL-D).**

**Result:**

*Telnet* disconnects from the remote host and returns you to HP-UX on your local host, displaying:

```
Connection closed by foreign host.
```

*local_HP-UX_prompt*

## Exiting from Telnet When Telnet Is in Its Command State

**At the telnet > prompt, enter:**

```
quit
```

**Result:**

*Telnet* disconnects from the remote host and returns you to HP-UX on your local host, displaying:

```
Connection closed.
```

*local_HP-UX_prompt*

# Obtaining Help

You can obtain summary information about *telnet* commands with *telnet*'s *?* command. You can either list the *telnet* commands or get information about a specific *telnet* command.

## Listing the Telnet Commands

**1. If you are not at the telnet> prompt, enter the *telnet* escape character.**

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

**2. At the telnet> prompt, enter:**

```
?
```

**Result:**

*Telnet* lists its commands, displaying:

```
Commands may be abbreviated.  Commands are:

open            connect to a site
close           close current connection
quit            exit telnet
escape          set escape character
status          print status information
options         toggle viewing of options processing
crmod           toggle mapping of received carriage returns
debug           toggle debugging
!               shell escape
?               print help information

telnet>
```

---

**Note**

If you were connected to a remote host and want to redisplay its prompt, press **Return** twice.

---

## Getting Information about a Specific Telnet Command

**1. If you are not already at the telnet > prompt, enter the *telnet* escape character.**

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

**2. At the telnet > prompt, enter:**

? *telnet_command*

**Example Entry:**

```
? open
```

**Result:**

*Telnet* displays:

*brief_description_of_command*

```
telnet>
```

---

**Note**

If you were connected to a remote host and want to redisplay its prompt, press **Return** twice.

---

# Temporarily Returning to HP-UX on Your Local Host

From within *telnet*, you can temporarily invoke a local HP-UX shell. This is a new shell, descended from the one started when you logged into your local host. This allows you to work on your local host and then return to *telnet*.

You can either:

- execute a single HP-UX command on your local host and automatically return to *telnet* or
- work on your local host for as long as you need to before you return to *telnet*.

## Executing a Single HP-UX Command on Your Local Host

**1. If you are not already at the telnet > prompt, enter the *telnet* escape character.**

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

**2. At the telnet > prompt, enter:**

! *HP-UX_command*

where *HP-UX_command* is an HP-UX command line.

**Example Entries:**

```
! pwd
```

```
! hostname
```

**Result:**

A local HP-UX shell executes the command and returns you to the remote host, displaying:

```
[Returning to remote]
```

---

**Note**

To redisplay the remote host's prompt, press **Return**.

---

## Working for an Extended Time on Your Local Host

1. **If you are not already at the telnet > prompt, enter the *telnet* escape character.**

   **Result:**

   *Telnet* displays its prompt:

   ```
   telnet>
   ```

2. **At the telnet > prompt, enter:**

   ```
   !
   ```

**Result:**

*Telnet* gives you a local HP-UX shell to work in, displaying:

*local_HP-UX_prompt*

**3. Enter HP-UX commands.**

**Result:**

The local shell executes each command you enter and then redisplays the local HP-UX prompt.

**4. Exit the local HP-UX shell as you normally would (such as with CTRL-D).**

**Result:**

The local shell returns you to the remote host, displaying:

```
[Returning to remote]
```

---

**Note**

To redisplay the remote host's prompt, press **Return**.

---

# Obtaining Telnet Status

You can display:

- whether or not a connection to a remote host exists and the name of the host to which a connection exists (if any),
- the current status of *echo* and *mode* (if a connection exists), and
- the current *telnet* escape character.

1. **If you are not already at the telnet> prompt, enter the *telnet* escape character.**

   **Result:**

   *Telnet* displays its prompt:

   ```
   telnet>
   ```

2. **At the telnet> prompt, enter:**

   ```
   status
   ```

   **Result:**

   *Telnet* displays its status information. If you are connected to a remote host, *telnet* returns you to the remote host.

   ---

   **Note**

   To redisplay the remote host's prompt, press **Return**.

   ---

   If you are not connected to a remote host, *telnet* redisplays its `telnet>` prompt.

# Changing Where User Input Is Echoed

You can choose whether user input is echoed **locally** or **remotely**. In **local** echo, user input is echoed to the terminal by the local *telnet* before being transmitted to the remote host. In **remote** echo, the remote host echoes user input. By default, *telnet* starts a connection in local echo, and requests that the TELNET server do remote echo. If the server refuses the request, you will see an error message. You can check the status of *echo* with the *telnet status* command.

Local echo produces less network traffic than remote echo, because the server need not transmit user input back to the local system, and will transmit only the output of the remote application. When communication between the local and remote systems is slow, local echo will appear to provide better system response. However, note that remote applications that expect to handle echoing of user input themselves, such as *csh(1)*, *ksh(1)*, and *vi(1)*, will not work correctly with local echo.

**1.** If you are not at the telnet> prompt, enter the *telnet* escape character.

**Result:**

*Telnet* displays its prompt:

```
telnet>
```

**2.** At the telnet> prompt, enter:

```
echo local
```

or

```
echo remote
```

**Result:**

*Telnet* now echoes input locally or remotely, depending on which you specified.

# Changing User Input Mode

You can set *telnet*'s user input mode to **character** or **line**. In **character** mode, *telnet* sends each character to the remote host as it is typed. In **line** mode, *telnet* gathers user input into lines and transmits each line to the remote host when the user types a carriage return, linefeed, or EOF. By default, *telnet* uses character mode. Note that setting line mode also sets local echo. You can check the status of *mode* with *telnet*'s *status* command.

In line mode, *telnet* transmits fewer packets over the network than it does in character mode, as it sends a packet only when the user terminates a line rather than sending each character in its own packet. This is particularly useful if you are connecting over some X.25 networks that charge users on a per-packet basis. However, note that remote applications that expect to interpret user input character by character, such as *more(1)*, *csh(1)*, *ksh(1)*, and *vi(1)*, will not work correctly in line mode.

1. If you are not already at the *telnet* > prompt, enter the telnet escape character.

   **Result:**

   *Telnet* displays its prompt:

   ```
   telnet>
   ```

2. At the telnet > prompt, enter:

   ```
   mode character
   ```

   or

   ```
   mode line
   ```

   **Result:**

   *Telnet* sets user input mode to character or line, depending on which you specified.

## Connecting to a Remote Host When You Invoke Telnet

The *telnet* command you give from your local HP-UX prompt can take the following form:

```
telnet remote_host
```

Specifying a remote host's name or alias (as listed in your local host's */etc/hosts* file) on the *telnet* command line causes *telnet* to connect to that remote host without your having to use *telnet*'s *open* command.

If you connect to a remote host in this way, *telnet*'s *close* command exits from *telnet*, instead of disconnecting from the remote host and remaining in *telnet*. Therefore, you can **not** connect to other remote hosts during the same *telnet* session. (You must reinvoke *telnet* to connect to another host.)

7

# Logging into a Host with Rlogin

*Rlogin* is a Berkeley Service that you use to log into a remote HP-UX or UNIX host from your local host.

Before you use *rlogin*, you should:

- determine if you need to change the *rlogin* escape character and
- determine what size characters to send using *rlogin*.

You should determine these things ahead of time because:

- they can affect whether *rlogin* operates properly and communicates properly with a remote host, and
- you can change the settings associated with these only when you invoke *rlogin*.

## Determining If You Need to Change the Rlogin Escape Character

The *rlogin* escape character, when combined with other particular characters, allows you to exit *rlogin* and allows you to temporarily return to HP-UX on your local host.

*Rlogin* presets its escape character to a tilde ( ~ ). You need to set the *rlogin* escape character to something else if:

- that character performs a function for a program you are running now on your local host, and
- you plan to run *rlogin* from within that local program.

Otherwise when you enter the character, the program you are currently running will respond to it, instead of *rlogin*.

## Caution: What Not to Change the Rlogin Escape Character To

If you must change the *rlogin* escape character, do **not** change it to a character that may conflict with your terminal configuration. HP-UX associates certain characters with specific functions. These characters are listed below:

| Function | Predefined Character in HP-UX |
|---|---|
| You can change these characters, so they may be different in your terminal configuration: | |
| End of File | **CTRL-D** (EOT) |
| Interrupt | **CTRL-C** (DEL or Rubout) |
| Quit | **CTRL-** (FS) |
| Erase | **CTRL-H** (#) |
| Kill | **CTRL-U** (@) |
| End of Line | **CTRL-@** (NUL) |
| Shell Layers Switch | **CTRL-Z** |

| Function | Predefined Character in HP-UX |
| --- | --- |
| You can **not** change these characters in your terminal configuration: | |
| New Line/Line Feed | **CTRL-J** (LF) |
| Stop or XOFF (Transmit Off) | **CTRL-S** (DC3) |
| Start or XON (Transmit On) | **CTRL-Q** (DC1) |
| Enquire | **CTRL-E** (ENQ) |
| Acknowledge | **CTRL-F** (ACK) |

To find out what characters are used in your terminal configuration, at your HP-UX prompt, enter:

```
stty -a
```

Also, do **not** change the *rlogin* escape character to a period (.) or an exclamation mark (!), because you combine these characters with the *rlogin* escape character either to exit *rlogin* or to return temporarily to HP-UX on your local host.

Note that it is inconvenient to change the *rlogin* escape character to one that you may frequently enter at the beginning of lines for a program running on a remote host. (This is because you would need to enter the character twice to allow the program to respond to the character, instead of *rlogin*.)

You change the *rlogin* escape character by invoking *rlogin* with the **–e** option, as described after the next section of this chapter.

# Determining What Size Characters to Send with Rlogin

Before you run *rlogin*, you must determine what size characters to send with *rlogin.* You must decide this ahead of time because:

- the character size can affect whether you can communicate properly with a remote host you login into with *rlogin*, and
- you can change the character size only when you invoke *rlogin*.

*Rlogin* sends eight-bit characters to a remote host unless you tell *rlogin* to send seven-bit characters instead.

In general, send seven-bit characters with *rlogin* if sending eight-bit characters with *rlogin* (the preset behavior) causes problems communicating with a remote host.

## When You Can Send Eight-Bit Characters

For communication between your local host and a remote host to work properly with eight-bit characters, all of the following must be configured for eight-bit characters:

- the remote host's tty driver,
- the local host's tty driver, and
- your local terminal hardware.

The following instructions tell how to check if these are configured for eight-bit characters.

1. Check whether the remote host is configured for, and can support, eight-bit characters. **If the remote host runs HP-UX**, you can do this by performing the same steps **on the remote host** as shown below for checking this on your local host.

2. Check whether your local host is configured for eight-bit characters as follows:

   **a. At your local HP-UX prompt, enter:**

   ```
   stty -a
   ```

   **Result:**

   The command displays all of your local host's tty driver settings.

   **b. Check the output for the setting:**

   ```
   cs8
   ```

   This means that the character size is set to eight bits.

   **c. If the character size is not set to eight bits, at your local HP-UX prompt, enter:**

   ```
   stty cs8
   ```

   **Result:**

   This sets the character size for your local host's tty driver to eight bits.

3. Check whether your local terminal hardware is configured for eight-bit characters in its configuration menu or in its switch settings.

### When You Must Send Seven-Bit Characters

You must tell *rlogin* to send seven-bit, instead of eight-bit, characters if:

- you can not configure your local terminal hardware to send eight-bit characters or seven-bit characters with high bit 0 (null parity),
- you can not configure a remote host to receive eight-bit characters (if it is not already configured to do so), or
- you might send eight-bit characters that a remote host interprets differently than your local host would. This can cause unpredictable results.

To send seven-bit characters with *rlogin,* you invoke *rlogin* with the –7 option, as explained in the next section of this chapter.

## Using Rlogin

If you have an account on a remote host, then with *rlogin*, you can either:

- log into the remote host automatically (without supplying your remote login name and password) if the remote host is configured to allow this or
- log into the remote host manually by supplying your remote login name and password.

The following sections explain each of these options for logging into a remote host with *rlogin*.

## Automatic Login

Rlogin allows you to log into a remote host without supplying your remote login name and password if the remote host is configured in either of two ways:

**Either:**

- you must have an account on the remote host with the **same** login name as your local login name, **and**
- the name of your local host must be in the remote host's */etc/hosts.equiv* file,

**or:**

- you must have an account on the remote host, **and**
- the name of your local host **and** your local login name must be in a *.rhosts* file in your home directory on the remote host.

The next section explains how to create a remote *$HOME/.rhosts* file for yourself, if you need to do so. Otherwise, skip the next section.

### Creating a $HOME/.rhosts File on a Remote Host

If you have an account on a remote host, you can set up the account so that you can log into the remote host without having to supply your remote login name and password. To do this, you create a file named *.rhosts* in your remote home directory. You can find out what your remote home directory is by entering:

```
echo $HOME
```

on the remote host. You must place the name of your local host and your local login name in the *.rhosts* file you create.

---

**Caution**

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the directions below for "Protecting Your .rhosts File."

---

The entry you place in your *.rhosts* file must have the following format:

*your_local_host's_name your_local_login_name*

You can separate *your_local_host's_name* and *your_local_login_name* with any number of tabs or spaces. Put any comments after *your_local_login_name*.

#### Example $HOME/.rhosts File Entry

If your local host's name were *hpabsa* and your local login name were *richard*, on the remote host you would create a *$HOME/.rhosts* file with the following entry:

```
hpabsa richard
```

### Protecting Your $HOME/.rhosts File

It is important to protect your remote *.rhosts* file and home directory to prevent unauthorized users from gaining *rlogin* access to your remote account and host. Only you should be able to create a *.rhosts* file in your remote home directory and write entries to the file. To do so:

**1.** Insure that your remote *.rhosts* file is owned by you, the user.

**2.** Use the HP-UX or UNIX *chmod* command to protect your remote *.rhosts* file with 0400 (-r--------) permission.

**3.** Use the HP-UX or UNIX *chmod* command to protect your remote home directory so that no one else can read it or write to it. For example, you should protect your remote home directory with at least 0711 (-rwx--x--x) permission.

## Logging into the Remote Host Automatically

**At your HP-UX prompt, enter:**

`rlogin` *remote_host* [`-e` *character*] [`-7`]

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*.

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple aliases. The */etc/hosts* file may contain comments and other information as well.

---

- the brackets ([ ]) mean that the enclosed option is optional. Omitting the **–e** option sets the *rlogin* escape character to a tilde ( ~ ), and omitting the **–7** option sets the character size to eight bits.
- *character* is the character you want to change the *rlogin* escape character to. If you want to enter a control character for the *rlogin* escape character, hold down **CTRL** while pressing another character key. (Control characters are not displayed.)
- **–7** is an option that sets the character size to seven bits, with the eighth bit set to zero.

**Example Entries:**

```
rlogin hpabsb

rlogin hpabsb -7

rlogin hpabsb -e=

rlogin hpabsb -e= -7
```

**Result:**

The remote host logs you in, displaying:

*remote_host's_login_message*

*remote_host_prompt*

---

**Note**

*Rlogin* does not send **Break** to the remote host. Therefore, you cannot **Break** out of a program on the remote host when you are logged into the remote host with *rlogin*.

---

If you are now logged into the remote host, skip to the section called "If You Get Unexpected Results after Logging Into a Remote Host."

## Manual Login

You must log into a remote host manually by supplying your remote login name and password if the remote host is not configured to allow automatic login.

### Logging into the Remote Host Manually

**1. At your HP-UX prompt, enter:**

`rlogin` *remote_host* [`-e` *character*] [`-7`] `-l` *remote_login_name*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*.

> **Note**
>
> The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's
>
> *internet_address official_name alias ...*
>
> The ellipsis (...) means that a host may have multiple aliases. The */etc/hosts* file may contain comments and other information as well.

- the brackets ([ ]) mean that the enclosed option is optional. Omitting the **–e** option sets the *rlogin* escape character to a tilde ( ~ ), and omitting the **–7** option sets the character size to eight bits.
- *character* is the character you want to change the *rlogin* escape character to. If you want to enter a control character for the *rlogin* escape character, hold down **CTRL** while pressing another character key. (Control characters are not displayed.)

- **–7** is an option that sets the character size to seven bits, with the eighth bit set to zero.
- *remote_login_name* is your login name on the remote host.

**Example Entries:**

```
rlogin hpabsb -l peter

rlogin hpabsb -7 -l peter

rlogin hpabsb -e= -l peter

rlogin hpabsb -e= -7 -l peter
```

**Result:**

The remote host prompts you for your remote password, displaying:

```
Password:
```

**2. Enter your remote password.**

**Result:**

The remote host logs you in, displaying:

*remote_host's_login_message*

*remote_host_prompt*

**Note**

*Rlogin* does not send **Break** to the remote host. Therefore, you cannot **Break** out of a program on the remote host when you are logged into the remote host with *rlogin*.

# If You Get Unexpected Results after Logging into a Remote Host

The values set in the remote host's */etc/profile* file (for *sh* users) or */etc/csh.login* file (for *csh* users) may not match those you are accustomed to on your local host. For example, the terminal type or erase character on the remote host may be different from what you have set up on your local host. Therefore, you may get unexpected results while working on the remote host.

To set values on the remote host to match the values you are accustomed to on the local host, create or edit your own *$HOME/.profile* file (if you use *sh*) or *$HOME/.login* file (if you use *csh*) on the remote host. An easy way to do this is to copy your local *$HOME/.profile* file or *$HOME/.login* file to your home directory on the remote host.

The values in your own remote *.profile* file or *.login* file take precedence over the values in the remote host's default */etc/profile* or */etc/csh.login* file.

# Logging Out of the Remote Host and Exiting Rlogin

There are two ways to log out of the remote host and exit *rlogin*:

- **Log out of the remote host as you normally would (such as with CTRL-D).**

  **Result:**

  *Rlogin* logs you out of the remote host, disconnects from the remote host and returns you to HP-UX on your local host, displaying:

  ```
  Connection closed.
  ```

  *local_HP-UX_prompt*

- **At the beginning of a new line, enter:**

  *rlogin_escape_character*.

  (That is, the *rlogin* escape character followed by a period.)

  > **Note**
  >
  > *Rlogin* does not display its escape character until you enter the period, or second character. *Rlogin* recognizes its escape character only at the beginning of a new line.

  **Example Entry:**

  ```
  ~.
  ```

**Result:**

*Rlogin* logs you out of the remote host, disconnects from the remote host and returns you to HP-UX on your local host, displaying:

```
Closed connection.
```

*local_HP-UX_prompt*

# Temporarily Returning to HP-UX on Your Local Host

From within *rlogin*, you can temporarily invoke a local HP-UX shell. This is a new shell, descended from the one started when you logged into your local host. This allows you to work on your local host and then return to *rlogin*.

You can either:

- execute a single HP-UX command on your local host and automatically return to *rlogin* or
- work on your local host for as long as you need to before you return to *rlogin*.

## Executing a Single HP-UX Command on Your Local Host

**At the beginning of a new line, enter:**

*rlogin_escape_character*! *HP-UX_command*

### Note

*Rlogin* does not display its escape character until you enter the exclamation mark, or second character. *Rlogin* recognizes its escape character only at the beginning of a new line.

**Example Entries:**

```
~! pwd
```

```
~! hostname
```

**Result:**

A local HP-UX shell executes the command and returns you to the remote host, displaying:

```
[Returning to remote]
```

### Note

To redisplay the remote host's prompt, press **Return**.

## Working for an Extended Time on Your Local Host

**1. At the beginning of a new line, enter:**

*rlogin_escape_character*!

---

**Note**

*Rlogin* does not display its escape character until you enter the exclamation mark, or second character. *Rlogin* recognizes its escape character only at the beginning of a new line.

---

**Example Entry:**

~!

**Result:**

*Rlogin* gives you a local HP-UX shell to work in, displaying:

*local_HP-UX_prompt*

**2. Enter HP-UX commands.**

**Result:**

The local shell executes each command you enter and then redisplays the local HP-UX prompt.

3. **Exit the local HP-UX shell as you normally would (such as with CTRL-D).**

**Result:**

The local shell returns you to the remote host, displaying:

```
[Returning to remote]
```

---

**Note**

To redisplay the remote host's prompt, press **Return**.

---

## Passing the Rlogin Escape Character to a Remote Program

If you are running a program on the remote host, and you want to send the program the same character as the *rlogin* escape character, you can always do this as long as you do not need to enter the character at the beginning of a new line.

However, if you need to enter the character at the beginning of a new line, you must enter the character twice because *rlogin* reads the first one as its own escape character. If this is frequently necessary, you may want to change the *rlogin* escape character.

For example, suppose that you have logged into a remote host with *rlogin*, and you run the *vi* editor on the remote host from within *rlogin*. The current *rlogin* escape character would be a tilde ( ~ ) if you did not change it. Normally, if you wanted to capitalize a character at the beginning of a new line with *vi*, you would position the cursor at that character and enter:

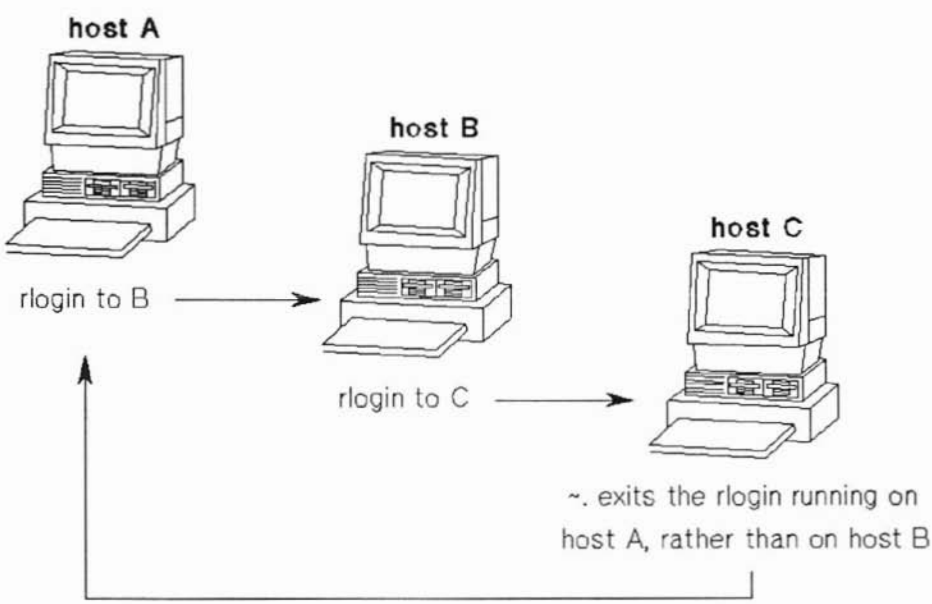
```
~
```

However, when you are running *vi* from within *rlogin*, your local *rlogin* interprets the tilde (~) first, causing it not to have any effect in the remote *vi*. To accomplish the capitalization at the beginning of a new line, you must enter:

```
~~
```

instead, so that *rlogin* understands that you want to pass a tilde (~) to the remote host.

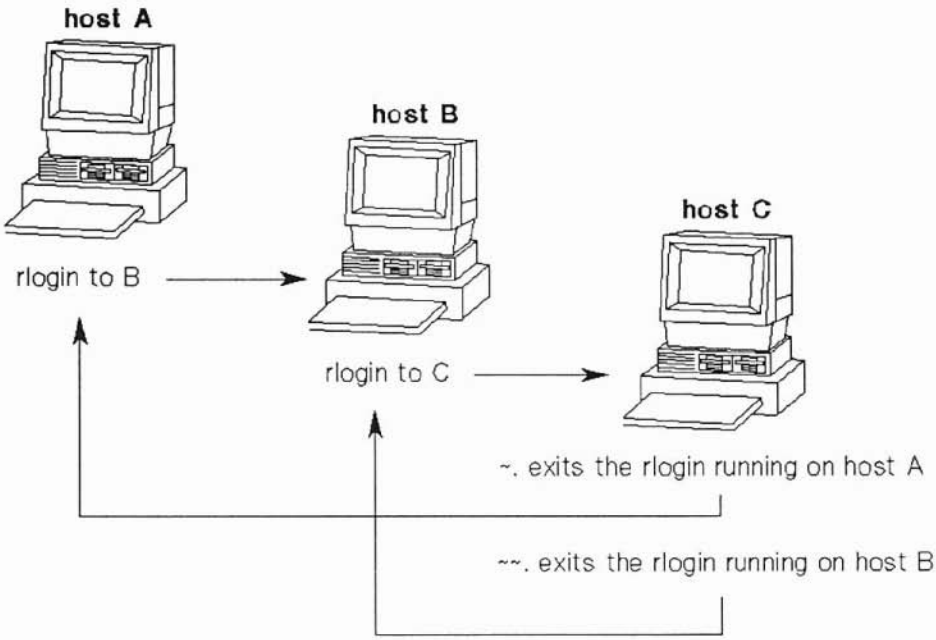
Another example of when you would need to enter the *rlogin* escape character more than once at the beginning of a new line is if you nested a series of *rlogin* commands over a chain of remote hosts. This case is illustrated below:



If you want to exit to host B, you must enter:

```
~~.
```

The first tilde (~) is interpreted by *rlogin* on host A, while the second one is passed to the program run from within *rlogin* on host B. That program happens to be *rlogin*, which responds to its exit escape sequence.




As a final example, if you wanted to invoke an interactive shell on host B instead of host A, you would enter:

```
~~!
```

(instead of ~!) at the beginning of a new line.

# Logging into a Remote Host as Someone Else

You can use *rlogin* to log into a remote host as someone else if you know that user's login name on the remote host, and either of the following two requirements are met:

**Either** that user must have your local host name and local login name in a *.rhosts* file in his or her home directory on the remote host,

**or** you must know that user's password on the remote host.

---

**Note**

If the remote user's account has no password, you can use *rlogin* to log into the remote host as that user without meeting either of the requirements above.

---

**1. At your HP-UX prompt, enter:**

`rlogin` *remote_host* `[-e` *character*`] [-7] -l` *remote_login_name*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*,
- the brackets ([ ]) mean that the enclosed option is optional,
- *character* is the character you may want to change the *rlogin* escape character to,
- **–7** is an option that sets the character size to seven bits, with the eighth bit set to zero, and
- *remote_login_name* is the login name of the remote user who you want to log into the remote host as.

**Example Entries:**

```
rlogin hpabsb -l alan

rlogin hpabsb -7 -l alan

rlogin hpabsb -e= -l alan

rlogin hpabsb -e= -7 -l alan
```

**Result:**

If the remote user has your local host name and local login name in his or her *$HOME/.rhosts* file on the remote host, then the remote host logs you in and displays:

*remote_host's_login_message*

*remote_host_prompt*

Otherwise, the remote host prompts you for the remote user's password, displaying:

```
Password:
```

2. **If the Password: prompt is displayed, enter the remote user's password.**

**Result:**

The remote host logs you in and displays:

*remote_host's_login_message*

*remote_host_prompt*

## Giving Other Remote Users Rlogin Access to Your Local Account

You can give remote users *rlogin* access to your local account by creating a *.rhosts* file. You place remote users' host names and login names in this file so that *rlogin* lets them log into your local host as you.

---

**Caution**

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the instructions below on "Protecting Your .rhosts File."

---

Your *.rhosts* file must be in your home directory on your local host. You can find out what your local home directory is by entering:

```
echo $HOME
```

on your local host.

Each entry you place in your *.rhosts* file must have the following format:

*remote_host_name remote_login_name*

Follow these rules when creating a *.rhosts* file:

- Each entry must contain a valid remote host name and remote login name.
- Separate the host name and login name with any number of tabs or blanks.
- Put any comments after the login name in any entry.

### Example .rhosts File Entry

If you wanted to give user *cdm* on remote host *hpabsc rlogin* access to your local account, you would create a *$HOME/.rhosts* file on your local host with the following entry:

```
hpabsc cdm
```

## Protecting Your .rhosts File

It is important to protect your *.rhosts* file and your local home directory to prevent unauthorized users from gaining *rlogin* access to your local account. Only you should be able to create a *.rhosts* file in your home directory and write entries to it. To do this:

1. Insure that your *.rhosts* file is owned by you, the user.
2. Use the HP-UX *chmod* command to protect your *.rhosts* file with 0400 (-r--------) permission.
3. Use the HP-UX *chmod* command to protect your local home directory so that no one else can read it or write to it. For example, you should protect your local home directory with at least 0711 (-rwx--x--x) permission.
4. Insure that your account has a password. Otherwise, remote users can log into your local host (with your login name) as you.

# Using Rlogin's "Shorthand" Syntax

If your local host is configured properly, you can enter an *rlogin* command line without the *rlogin* command. That is, an *rlogin* command line can start with the name of a remote host, omitting the *rlogin* command.

You can use this shorthand syntax only if two conditions hold true. You can omit the command *rlogin* from the *rlogin* command line if:

- you add the path */usr/hosts* to your command search path in your *.login*, *.cshrc*, or *.profile* file. Which file contains your *$PATH* variable depends on which shell you use.
- the super-user or node manager has linked */usr/bin/remsh* to */usr/hosts/host*, where *host* is the name or alias of a remote host (listed in */etc/hosts*) on which you want to execute a command.

---

**Note**

*Remsh* knows whether you mean to invoke *rlogin* or *remsh* by parsing what you give on the command line.

---

To find out which hosts you can use *rlogin*'s shorthand syntax for, list the contents of the directory */usr/hosts*.

8

# Transferring Files with Ftp

*Ftp* is an ARPA Service that allows you to transfer files among HP-UX, UNIX, and non-UNIX network hosts that support ARPA Services. *Ftp* is the file transfer program that uses the ARPA standard File Transfer Protocol (FTP).

*Ftp* has a one-line command syntax. This service not only allows you to perform file transfers, but also file management operations such as changing, listing, creating, and deleting remote directories.

## Using Ftp

To use *ftp*, you:

- invoke *ftp*,
- choose whether or not to display responses from any remote host you connect to,
- connect to a remote host, and then
- log into that host.

After that, you can use *ftp* to perform file management operations and file transfers.

## 1. Invoke Ftp

**At your HP-UX prompt, enter:**

```
ftp
```

**Result:**

*Ftp* displays its prompt:

```
ftp>
```

## 2. Choose Whether to Display Responses from a Remote Host

*Ftp* can display all responses from any remote host you connect to. These responses tell you whether *ftp* commands completed successfully. This feature is called **verbose mode**.

You can also choose **not** to have *ftp* display all responses from the remote host so that in most cases, on completing a command, *ftp* just returns its `ftp>` prompt. Even if verbose mode is off, when you change one of *ftp*'s settings, *ftp* displays the resulting state of the setting.

If *ftp*'s input comes from your keyboard (HP-UX terminal), *ftp* initially has verbose mode on. (This is the usual setting.) Otherwise, *ftp* has verbose mode off (for example, if *ftp*'s input is coming from a file).

---

**Note**

This chapter shows both verbose and non-verbose *ftp* responses.

---

## Turning Verbose Mode On or Off

**At the ftp> prompt, enter:**

```
verbose
```

**Result:**

If verbose mode was on, *ftp* turns it off and displays:

```
Verbose mode off.

ftp>
```

If verbose mode was off, *ftp* turns it on and displays:

```
Verbose mode on.

ftp>
```

## 3. Connect to a Remote Host

**At the ftp > prompt, enter:**

```
open remote_host
```

where *remote_host* is the name or alias of a host listed in */etc/hosts*.

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

To connect to a host **not** listed in */etc/hosts*, you can give the host's internet address where *remote_host* appears above. The internet address must be in dot notation (for example, 192.6.21.9).

---

**Example Entry:**

```
open hpabsa
```

**Result:**

*Ftp* connects you to the remote host and prompts for a remote login name.

- If verbose mode is off, *ftp* displays:

  `Name (`*remote_host*`:`*remote_login_name*`):`

- If verbose mode is on, *ftp* displays:

  `Connected to` *remote_host*`.`

  *remote_host* `FTP server...ready.`

  `Name (`*remote_host*`:`*remote_login_name*`):`

## 4. Log into the Remote Host

---

**Note**

For security reasons, you can only log into accounts that have passwords associated with them. You must be able to supply *ftp* with a valid login name and password on the remote host.

---

1. **To log in with the same remote login name as your local login name, press Return at the Name (...): prompt.**

   **To log in with a different remote login name, enter the remote login name at the Name (...): prompt .**

   **Result:**

   *Ftp* prompts for the remote password associated with the login name you gave, displaying:

   `Password (`*remote_host*`:`*remote_login_name*`):`

**2. Enter the password associated with the remote login name you gave.**

**Result:**

*Ftp* logs you into the remote host if the password is valid.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Password required for remote_login_name.

User remote_login_name logged in.

ftp>
```

**Problems?** *Ftp* may also prompt you for an account name, if the remote host you are logging into requires one.

The remote host may be configured to refuse *ftp* connections to specific users for security reasons.

# Disconnecting from a Remote Host and Exiting Ftp

You have two options for disconnecting from a remote host:

- You can exit *ftp*. This disconnects from a remote host if a connection exists and returns you to HP-UX.
- You can disconnect from a remote host and remain in *ftp*. This is useful if you want to connect to other remote hosts during the same *ftp* session.

## Exiting Ftp to Return to HP-UX on Your Local Host

**At the ftp> prompt, enter:**

```
quit
```

**or**

```
bye
```

**Result:**

If a connection exists to a remote host, *ftp* disconnects from the remote host and returns you to HP-UX on your local host.

- If verbose mode is off, your local host redisplays its HP-UX prompt.
- If verbose mode is on, *ftp* displays:

  ```
  Goodbye.
  ```

  and your local host redisplays its HP-UX prompt.

If a connection does not exist to a remote host, *ftp* returns you to HP-UX on your local host, and your local host redisplays its HP-UX prompt.

## Disconnecting from a Remote Host and Remaining in Ftp

**At the ftp> prompt, enter:**

```
close
```

**Result:**

*Ftp* disconnects from the remote host.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Goodbye.

ftp>
```

# Obtaining Help

You can obtain summary information about *ftp* commands with *ftp*'s *help* command. You can either list the *ftp* commands or get information about a specific *ftp* command.

## Listing the Ftp Commands

**At the ftp > prompt, enter:**

```
help
```

**or**

```
?
```

**Result:**

*Ftp* lists its commands, displaying:

```
Commands may be abbreviated.  Commands are:
```

*command_list*

```
ftp>
```

## Getting Information about a Specific Ftp Command

**At the ftp > prompt, enter:**

`help` *ftp_command*

**or**

`?` *ftp_command*

**Result:**

*Ftp* displays:

```
command            brief_description_of_command

ftp>
```

# Temporarily Returning to HP-UX on Your Local Host

From within *ftp*, you can temporarily invoke a local HP-UX shell. This is a new shell, descended from the one started when you logged into the local host. This allows you to work on the local host and then return to *ftp*.

You can either:

- execute a single HP-UX command on your local host and automatically return to *ftp* or
- work on your local host for as long as you need to before you return to *ftp*.

## Executing a Single HP-UX Command on Your Local Host

**At the ftp> prompt, enter:**

! *HP-UX_command*

where *command* is an HP-UX command line.

**Example Entries:**

```
! pwd

! hostname
```

**Result:**

A local HP-UX shell executes the command and returns you to *ftp* and the `ftp>` prompt.

## Working for an Extended Time on Your Local Host

**1. At the ftp > prompt, enter:**

```
!
```

**Result:**

*Ftp* gives you a local HP-UX shell to work in, displaying:

*local_HP-UX_prompt*

**2. Enter HP-UX commands.**

**Result:**

The local shell executes each command you enter and then redisplays the local HP-UX prompt.

**3. Exit the local HP-UX shell as you normally would (such as with CTRL-D).**

**Result:**

The local shell returns you to *ftp* and the `ftp>` prompt.

# How Ftp Treats "Wild Card" Characters, or Metacharacters

You can use *csh(1)* metacharacters in the directory and file names you specify in *ftp* commands. These metacharacters, or wild card characters, represent a set of characters or character strings and are a "shorthand" way of specifying a set of directory or file names. The following table is a quick reference to the meaning of the *csh(1)* metacharacters supported by *ftp*:

**Csh Metacharacters Supported by Ftp**

| Character | Matches |
|---|---|
| * | any string, including a null string |
| ? | any single character |
| [ ] | any of the enclosed characters |
| ~ | your home directory |
| ~*login_name* | *login_name*'s home directory |
| { , } | any of the enclosed character strings separated by commas |

The expansion of metacharacters into the directory and file names they match is called **globbing**. Globbing is on when you first invoke *ftp*. You must turn it off if you need to specify a directory or file name containing one of the characters listed above. That way, *ftp* interprets the character literally, instead of trying to match it to a set of characters. You can turn globbing on or off while in *ftp* at the `ftp>` prompt.

## Turning Globbing On or Off

**At the ftp> prompt, enter:**

```
glob
```

**Result:**

If globbing was on, *ftp* turns it off and displays:

```
Globbing off.

ftp>
```

If globbing was off, *ftp* turns it on and displays:

```
Globbing on.

ftp>
```

For **some** of its commands *ftp* **always** expands metacharacters, even if globbing is off. If you want a local or remote host to interpret a metacharacter literally when given one of these commands, precede the character with a backslash (\). For example, with some *ftp* commands, you would need to enter a directory named *my?s* as *my\?s*.

The following table summarizes globbing behavior for applicable *ftp* commands. These commands are discussed in more detail later in this chapter.

**Globbing Behavior for Ftp Commands**

| Ftp Command | Metacharacters Are |
|---|---|
| *dir* | **always** expanded |
| *ls* | **always** expanded |
| *mdelete* | expanded if globbing is on |
| *mdir* | expanded if globbing is on |
| *mget* | expanded if globbing is on |
| *mls* | expanded if globbing is on |
| *mput* | expanded if globbing is on |

In single-file *ftp* commands:

- *ftp* expands metacharacters in a remote file specification only if it begins with the tilde ( ~ ) metacharacter. *Ftp* then performs the operation on only the first file that matches the expanded specification.
- *ftp* always expands metacharacters in a local file specification. *Ftp* then performs the operation on only the first file that matches the expanded specification.

## Performing Directory Operations with Ftp

From within *ftp*, you can:

- change local and remote working directories,
- list the contents of remote directories,
- display the name of the remote working directory,
- create a remote directory,
- delete a remote directory, and
- change the name of a remote directory.

The following sections tell how to perform these directory operations.

## Changing the Local Working Directory

### To Your Local Home Directory

**At the ftp> prompt, enter:**

```
lcd
```

**Result:**

*Ftp* changes the local working directory to your local home directory and displays:

Local directory now *your_local_home_directory_path*

```
ftp>
```

### To Another Local Directory

**At the ftp> prompt, enter:**

lcd *local_directory_path*

where *local_directory_path* is the full or relative path to a local directory.

**Example Entries:**

```
lcd /users/richard/projects
```

```
lcd projects
```

**Result:**

*Ftp* changes the local working directory to the path you specify and displays:

```
Local directory now full_local_working_directory_path

ftp>
```

## Changing the Remote Working Directory

**At the ftp> prompt, enter:**

`cd` *remote_directory_path*

where *remote_directory_path* is the full or relative path to a remote directory.

**Example Entries:**

```
cd /users/lab/richard/xfers

cd xfers
```

**Result:**

*Ftp* changes the remote working directory to the path you specify.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
CWD command okay.

ftp>
```

## Listing the Contents of the Remote Working Directory

With *ftp*, you can get an extended or abbreviated listing of the contents of the remote working directory. You can send the listing to the display (actually, *stdout*) or to a local file.

### To Stdout (Usually the Display)

---

**Note**

Metacharacters are always expanded for *ftp*'s *dir* and *ls* commands.

---

**For an extended listing, at the ftp> prompt, enter:**

```
dir
```

**For an abbreviated listing, at the ftp> prompt, enter:**

```
ls
```

**Result:**

*Ftp* lists the contents of the remote working directory to the display (if *stdout* is not redirected).

- If verbose mode is off, *ftp* displays:

```
directory_contents_listing
ftp>
```

- If verbose mode is on, *ftp* displays:

```
PORT command okay.
Opening data connection for /bin/ls -l...
directory_contents_listing
Transfer complete.
number bytes received in number seconds...
ftp>
```

### To a Local File

---

**Note**

Metacharacters are always expanded for *ftp*'s *dir* and *ls* commands.

---

**For an extended listing, at the ftp > prompt, enter:**

`dir . ` *local_file_path*

**For an abbreviated listing, at the ftp > prompt, enter:**

`ls . ` *local_file_path*

where the period (.) represents the remote working directory, and *local_file_path* is the full or relative path to a local file.

**Example Entries:**

```
dir . /users/richard/projects/dirconts

ls . dirconts
```

**Result:**

*Ftp* lists the contents of the remote working directory to the local file you specify.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for /bin/ls -l...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

## Listing the Contents of a Remote Directory

With *ftp*, you can get an extended or abbreviated listing of the contents of a remote working directory. You can send the listing to the display (actually, *stdout*) or to a local file.

### To Stdout (Usually the Display)

---

**Note**

Metacharacters are always expanded for *ftp*'s *dir* and *ls* commands.

---

**For an extended listing, at the ftp > prompt, enter:**

`dir` *remote_directory_path*

**For an abbreviated listing, at the ftp > prompt, enter:**

`ls` *remote_directory_path*

where *remote_directory_path* is the full or relative path to a remote directory.

**Example Entries:**

```
dir /users/lab/richard/doc
```

```
ls doc
```

**Result:**

*Ftp* lists the contents of the remote directory you specify to the display (actually, *stdout*).

- If verbose mode is off, *ftp* displays:

```
directory_contents_listing
ftp>
```

- If verbose mode is on, *ftp* displays:

```
PORT command okay.
Opening data connection for /bin/ls -l...
directory_contents_listing
Transfer complete.
number bytes received in number seconds...
ftp>
```

### To a Local File

---

**Note**

Metacharacters are always expanded for *ftp*'s *dir* and *ls* commands.

---

**For an extended listing, at the ftp> prompt, enter:**

`dir` *remote_directory_path* *local_file_path*

**For an abbreviated listing, at the ftp> prompt, enter:**

`ls` *remote_directory_path* *local_file_path*

where *remote_directory_path* is the full or relative path to a remote directory, and *local_file_path* is the full or relative path to a local file.

**Example Entries:**

```
dir /users/lab/richard/doc /users/richard/projects/dirconts
```

```
ls doc dirconts
```

**Result:**

*Ftp* lists the contents of the remote directory you specify to the local file you specify.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for /bin/ls -l...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

## Listing the Contents of Multiple Remote Directories

With *ftp*, you can get an extended or abbreviated listing of the contents of multiple remote directories. You can send the listing to the display (actually, *stdout*) or to a local file.

### To Stdout (Usually the Display)

---

**Note**

Metacharacters are expanded for *ftp*'s *mdir* and *mls* commands if globbing is on.

---

**For extended listings, at the ftp > prompt, enter:**

`mdir` *remote_directory_path ...* –

**For abbreviated listings, at the ftp > prompt, enter:**

`mls` *remote_directory_path ...* –

where *remote_directory_path* is the full or relative path to a remote directory, and – represents *stdout*. The ellipsis (...) means that you can specify multiple *remote_directory_path*s.

**Example Entries:**

```
mdir /users/lab/richard /users/lab/richard/doc -

mls doc code code/source -
```

**Result:**

If interactive mode is on, *ftp* prompts you to verify that *stdout* (the display) is really where you want the contents of the directories listed. *Ftp* displays:

```
local-file -?
```

- If this is **not** what you want to do, enter:

  ```
  N
  ```

  and *ftp* cancels the directory listings and redisplays its `ftp>` prompt.

- If this is what you want to do, enter:

  ```
  Y
  ```

  and continue with the description below. Interactive mode is explained in detail in a later section of this chapter.

*Ftp* displays (lists to *stdout*) the contents of the remote directories you specify.

- If verbose mode is off, *ftp* displays:

```
directory_contents_listing
directory_contents_listing
.
. (repeated for each directory listed)
.
ftp>
```

- If verbose mode is on, *ftp* displays:

```
PORT command okay.
Opening data connection for /bin/ls...
directory_contents_listing
Transfer complete.
number bytes received in number seconds...
PORT command okay.
Opening data connection for /bin/ls...
directory_contents_listing
Transfer complete.
number bytes received in number seconds...
.
. (repeated for each directory listed)
.
ftp>
```

### To a Local File

With *ftp*, you can send an extended or abbreviated listing of the contents of multiple remote directories to a local file.

---

**Note**

Metacharacters are expanded for *ftp*'s *mdir* and *mls* commands if globbing is on.

---

**For extended listings, at the ftp> prompt, enter:**

`mdir` *remote_directory_path..*   *local_file_path*

**For abbreviated listings, at the ftp> prompt, enter:**

`mls` *remote_directory_path ...*   *local_file_path*

where *remote_directory_path* is the full or relative path to a remote directory, and *local_file_path* is the full or relative path to a local file. The ellipsis (...) means that you can specify multiple *remote_directory_path*s.

**Example Entries:**

```
mdir /users/lab/richard /users/lab/richard/doc dirlists
```

```
mls doc code code/source dirlists
```

**Result:**

If interactive mode is on, *ftp* prompts you to verify that the local file you specified is really where you want the contents of the directories listed. *Ftp* displays:

```
local-file local_file_path ?
```

- If *local_file_path* is **not** correct, enter:

  ```
  N
  ```

  and *ftp* cancels the directory listings and redisplays its `ftp>` prompt.

- If *local_file_path* is correct, enter:

  ```
  Y
  ```

  and continue with the description below. Interactive mode is explained in detail in a later section of this chapter.

*Ftp* lists the contents of the remote directories you specify to the local file you specify.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

  `PORT command okay.`

  `Opening data connection for /bin/ls...`

  `Transfer complete.`

  *`number`* `bytes received in` *`number`* `seconds...`

  `PORT command okay.`

  `Opening data connection for /bin/ls...`

  `Transfer complete.`

  *`number`* `bytes received in` *`number`* `seconds...`

  .
  . (repeated for each directory listed)
  .

  `ftp>`

## Displaying the Name of the Remote Working Directory

**At the ftp > prompt, enter:**

```
pwd
```

**Result:**

*Ftp* displays:

```
"full_remote_working_directory_path" is the current working
directory.

ftp>
```

## Creating a Remote Directory

**At the ftp> prompt, enter:**

mkdir *remote_directory_path*

where *remote_directory_path* is a full or relative path to a remote directory.

**Example Entries:**

```
mkdir /users/lab/richard/temp

mkdir temp
```

**Result:**

*Ftp* creates the remote directory you specify.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  MKDIR command okay.

  ftp>
  ```

## Deleting a Remote Directory

---

**Note**

To delete a remote directory, the directory **must** be empty.

---

**At the ftp> prompt, enter:**

`rmdir` *remote_directory_path*

**or**

`delete` *remote_directory_path*

where *remote_directory_path* is a full or relative path to a remote directory.

**Example Entries:**

```
rmdir /users/lab/richard/temp
```

```
delete temp
```

**Result:**

*Ftp* deletes the remote directory you specify.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
RMDIR command okay.

ftp>
```

**or**

```
DELE command okay.

ftp>
```

## Changing the Name of a Remote Directory

---

**Note**

You can only rename a remote directory. You can **not** change the path to (move) a remote directory.

---

**At the ftp> prompt, enter:**

`rename` *old_remote_directory_path new_remote_directory_path*

where *old_remote_directory_path* is a full or relative path to an existing remote directory, and *new_remote_directory_path* is the same as *old_remote_directory_path*, except for the directory name.

### Note

If you do not specify *new_remote_directory_path*, *ftp* prompts you for it by displaying:

```
(to-name)
```

**Example Entries:**

```
rename /users/lab/rich/doc /users/lab/rich/comment
```

```
rename doc comment
```

**Result:**

*Ftp* changes the name of the existing directory to the new name you specify.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  File exists, ready for destination name.

  RNTO command okay.

  ftp>
  ```

# Transferring Files with Ftp

*Ftp* lets you set up the file transfer environment before you actually transfer any files. This includes:

- setting the local and remote working directories,
- setting the file transfer type,
- turning on (or off) options to monitor file transfer progress, and
- turning on (or off) prompting to confirm each file transfer.

Once you have set up the file transfer environment, you can transfer files between your local host and a remote host.

## 1. Set the Local and Remote Working Directories

Use *ftp*'s *lcd* and *cd* commands to set the local and remote working directories before you begin the file transfer. These commands are discussed earlier in this chapter.

## 2. Set the File Transfer Type

The file transfer type you choose depends on the other operating system involved in the file transfer, **not** on the type of file you are transferring.

When you transfer files to or from a host with the HP-UX or UNIX operating system, you can use either the *ascii* or *binary* file transfer type. However, the file transfer is faster if you set the type to *binary*.

When you transfer files to or from a host with a non-HP-UX or non-UNIX operating system, you must use the *ascii* file transfer type. This insures that the files are transferred in a format appropriate for the other (different) operating system.

The only time you may want to use the faster *binary* file transfer type for transferring files to or from a non-HP-UX or non-UNIX host is to archive them (for storage only, not access). The files would not be in a meaningful format if accessed on the destination host after the transfer. You would need to transfer the files back to a host with the same type of operating system from which they originated to access them in a meaningful form.

When you first invoke *ftp*, the file transfer type is preset to *ascii*.

### Displaying the Current File Transfer Type

**At the ftp > prompt, enter:**

```
type
```

**Result:**

If the current file transfer type is ascii, *ftp* displays:

```
Using ascii mode to transfer files.

ftp>
```

If the current file transfer type is binary, *ftp* displays:

```
Using binary mode to transfer files.

ftp>
```

### Changing the File Transfer Type to Binary

**At the ftp > prompt, enter:**

```
binary
```

**or**

```
type binary
```

**Result:**

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Type set to I.

ftp>
```

## Changing the File Transfer Type to Ascii

**At the ftp> prompt, enter:**

```
ascii
```

**or**

```
type ascii
```

**Result:**

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Type set to A.

ftp>
```

## 3. Choose Options to Monitor File Transfer Progress

Within *ftp*, there are two ways to monitor the progress of a file transfer.

- You can have *ftp* display a hash (#) sign for each 1024 bytes transferred.
- You can have *ftp* sound a bell after each file transfer completes.

Both of these options are off when you first invoke *ftp*.

### Turning Hash Sign Displaying On or Off

**At the ftp > prompt, enter:**

```
hash
```

**Result:**

If hash sign displaying was off, *ftp* turns it on and displays:

```
Hash mark printing on (1024 bytes/hash mark).

ftp>
```

If hash sign displaying was on, *ftp* turns it off and displays:

```
Hash mark printing off.

ftp>
```

### Turning Bell Sounding On or Off

**At the ftp > prompt, enter:**

```
bell
```

**Result:**

If bell sounding was off, *ftp* turns it on and displays:

```
Bell mode on.

ftp>
```

If bell sounding was on, *ftp* turns it off and displays:

```
Bell mode off.

ftp>
```

## 4. Turn on Interactive Mode for Selective File Transfers

When you perform an operation involving multiple files, *ftp* can list the files one by one, asking for each whether you want to perform the operation or not. This allows you to perform the operation on only the files you select. *Ftp* calls this feature **interactive mode**.

Use interactive mode when you want to select from among multiple files to transfer during a single file transfer operation.

You can also use interactive mode when deleting multiple remote files and listing the contents of multiple remote directories to a local destination. However, in the latter case, *ftp* prompts you to confirm the local destination, not each remote directory listing.

Interactive mode is on when you first invoke *ftp*.

### Turning Interactive Mode On or Off

**At the ftp> prompt, enter:**

```
prompt
```

**Result:**

If interactive mode was on, *ftp* turns it off and displays:

```
Interactive mode off.

ftp>
```

If interactive mode was off, *ftp* turns it on and displays:

```
Interactive mode on.

ftp>
```

## 5. Perform One or More File Transfers

---

**Note**

When you transfer files with *ftp*, you are really copying those files from one place to another. Transfers do not move or delete the original files.

---

**Ftp File Transfer Options**

| With *Ftp*, You Can Transfer | To |
|---|---|
| a remote file | a local file with the same directory path and name |
| | a local file with a different directory path and/or name |
| multiple remote files | multiple local files with the same directory paths and names |
| a local file | a remote file with the same directory path and name |
| | the end of a remote file with the same directory path and name |
| | a remote file with a different directory path and/or name |
| | the end of a remote file with a different directory path and/or name |
| multiple local files | multiple remote files with the same directory paths and names |

The following sections explain all of these *ftp* options for transferring files.

### Caution

In an *ftp* file transfer, insure that the source file and destination file are **not** the **same file** on the same host. Otherwise, *ftp* destroys the contents of the file.

## From a Remote File
## To a Local File with the Same Directory Path and Name

This special case of an *ftp* file transfer requires a special syntax. The syntax you use causes *ftp* to behave in the following way:

- *Ftp* copies a file from the remote working directory to a file with the **same name** in the local working directory. The remote and local working directory paths may be different.

- *Ftp* copies a file from any other remote directory to a local file with the **same directory path** and **name** you identify the remote file with.

**At the ftp> prompt, enter:**

`get` *remote_file_path*

**or**

`recv` *remote_file_path*

where *remote_file_path* is either:

- the name of the source file if the file is in the remote working directory or

- the full or relative path to the source file if the file is in another remote directory.

**Note**

Any remote **directory path** you specify as part of a *remote_file_path* **must also exist on the local host.** Otherwise, *ftp* will not transfer the file.

**Example Entries:**

```
get program1
```

```
recv /usr/bin/game
```

**Result:**

If the file is in the remote working directory, *ftp* copies the file to a file with the same name in the local working directory.

Otherwise, *ftp* copies the remote file to a local file with the same directory path and name you identified the remote file with.

If the destination file already exists, *ftp* replaces its contents with the source file's contents.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for remote_file_name...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

## From a Remote File
## To a Local File with a Different Directory Path and/or Name

**At the ftp> prompt, enter:**

get *remote_file_path local_file_path*

**or**

recv *remote_file_path local_file_path*

where *remote_file_path* is a full or relative path to the source file in a remote directory and *local_file_path* is a full or relative path to the destination file in a local directory.

---

**Note**

*Local_file_path* **must** include the destination file's name.

---

**Example Entries:**

```
get program1 firstprog

recv program1 code/source/program1

get /users/lab/richard/graphics/design1 design1

recv graphics/design1 graphics/work/firstdsgn
```

**Result:**

*Ftp* copies the file from the remote directory to the file in the local directory. If the destination file already exists, *ftp* replaces its contents with the source file's contents.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  PORT command okay.

  Opening data connection for *remote_file_name*...

  Transfer complete.

  *number* bytes received in *number* seconds...

  ftp>

## From Multiple Remote Files
## To Multiple Local Files with the Same Directory Paths and Names

*Ftp* behaves in the following way when you transfer multiple files from a remote host to a local host:

- *Ftp* copies files from the remote working directory to files with the **same names** in the local working directory. The remote and local working directories may be different.
- *Ftp* copies files from any other remote directory to local files with the **same directory paths** and **names** you identify the remote files with.

---

**Note**

Metacharacters are expanded for *ftp*'s *mget* command if globbing is on.

---

### If Interactive Mode Is On

**1. At the ftp> prompt, enter:**

`mget` *remote_file_path ...*

where *remote_file_path* is either:

- the name of a file if the file is in the remote working directory or
- the full or relative path to a file if the file is in another remote directory.

The ellipsis (...) means that you can specify multiple *remote_file_path*s.

---

**Note**

Any remote **directory path** you specify as part of a *remote_file_path* **must also exist on the local host**. Otherwise, *ftp* will not transfer the file(s).

---

**Example Entries:**

```
mget /bin/c*
```

```
mget program1 program2 program3
```

**Result:**

*Ftp* asks if you want to transfer the first remote file that matches what you specified. This gives you the option of **not** transferring the remote file.

*Ftp* displays:

mget *first_remote_file_path*?

**2. To not transfer the remote file, enter:**

```
N
```

**To transfer the remote file, enter:**

```
Y
```

**Result:**

If you enter N, *ftp* does **not** transfer the remote file and asks if you want to transfer the next remote file that matches what you specified, displaying:

mget *next_remote_file_path*?

If you enter Y, and:

- the file is in the remote working directory, *ftp* copies the file to a file with the same name in the local working directory.
- the file is in another remote directory, *ftp* copies the remote file to a local file with the same directory path and name.

Then *ftp* asks if you want to transfer the next remote file that matches what you specified.

- If verbose mode is off, *ftp* displays:

  mget *next_remote_file_path*?

- If verbose mode is on, *ftp* displays:

  PORT command okay.

  Opening data connection for *remote_file_path*...

  Transfer complete.

  *number* bytes received in *number* seconds...

  mget *next_remote_file_path*?

**3. Repeat the previous step until ftp redisplays its ftp> prompt instead of the mget ...? prompt.**

This means that there are no more files that match what you originally entered.

### If Interactive Mode Is Off

**At the ftp > prompt, enter:**

`mget` *remote_file_path ...*

where *remote_file_path* is either:

- the name of a source file if the file is in the remote working directory or
- the full or relative path to a source file if the file is in another remote directory.

The ellipsis (...) means that you can specify multiple *remote_file_path*s.

---

**Note**

Any remote **directory path** you specify as part of a *remote_file_path* **must also exist on the local host**. Otherwise, *ftp* will not transfer the file(s).

---

**Example Entries:**

```
mget /bin/c*
```

```
mget program1 program2 program3
```

**Result:**

*Ftp* copies any files in the remote working directory to files with the same names in the local working directory.

*Ftp* copies any other remote files to local files with the same directory paths and names you identified the local files with.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

PORT command okay.

Opening data connection for *remote_file_path*...

Transfer complete.

*number* bytes received in *number* seconds...

PORT command okay.

Opening data connection for *remote_file_path*...

Transfer complete.

*number* bytes received in *number* seconds...

.
. (repeated for each file transferred)
.

ftp>

## From a Local File
## To a Remote File with the Same Directory Path and Name

This special case of an *ftp* file transfer requires a special syntax. The syntax you use causes *ftp* to behave in the following way:

- *Ftp* copies a file from the local working directory to a file with the **same name** in the remote working directory. The local and remote working directory paths may be different.

- *Ftp* copies a file from any other local directory to a remote file with the **same directory path** and **name** you identify the local file with.

**At the ftp> prompt, enter:**

`put` *local_file_path*

**or**

`send` *local_file_path*

where *local_file_path* is either:

- the name of the source file if the file is in the local working directory or

- the full or relative path to the source file if the file is in another local directory.

---

**Note**

Any local **directory path** you specify as part of a *local_file_path* **must also exist on the remote host**. Otherwise, *ftp* will not transfer the file.

---

**Example Entries:**

```
put program2

send /tmp/printout
```

**Result:**

If the file is in the local working directory, *ftp* copies the file to a file with the same name in the remote working directory.

Otherwise, *ftp* copies the local file to a remote file with the same directory path and name you identified the local file with.

If the destination file already exists, *ftp* replaces its contents with the source file's contents.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for remote_file_name...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

### From a Local File To the End of a Remote File with the Same Directory Path and Name

This special case of an *ftp* file transfer requires a special syntax. The syntax you use causes *ftp* to behave in the following way:

- *Ftp* copies a file from the local working directory to the end of a file with the same name in the remote working directory. The local and remote working directory paths may be different.
- *Ftp* copies a file from any other local directory to the end of a remote file with the **same name** and **directory path** you identify the local file with.

**At the ftp> prompt, enter:**

append *local_file_path*

where *local_file_path* is either:

- the name of the source file if the file is in the local working directory or
- the full or relative path to the source file if the file is in another local directory.

---

**Note**

Any local **directory path** you specify as part of a *local_file_path* **must also exist on the remote host.** Otherwise, *ftp* will not transfer the file.

---

**Example Entries:**

```
append form
```

```
append /tmp/testdata
```

**Result:**

If the file is in the local working directory, *ftp* copies the file to the end of a file with the same name in the remote working directory.

Otherwise, *ftp* copies the local file to the end of a remote file with the same directory path and name you identified the local file with.

If the destination file does not exist, *ftp* creates it.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for remote_file_name...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

## From a Local File
## To a Remote File with a Different Directory Path and/or Name

**At the ftp> prompt, enter:**

`put` *local_file_path remote_file_path*

**or**

`send` *local_file_path remote_file_path*

where *local_file_path* is a full or relative path to the source file in a local directory and *remote_file_path* is a full or relative path to the destination file in a remote directory.

---

**Note**

*Remote_file_path* **must** include the destination file's name.

---

**Example Entries:**

```
put report results

send doc/internal/issues issues

put status /users/lab/richard/mail/update

send email/urgent/schedule project/schedule
```

**Result:**

*Ftp* copies the file from the local directory to the file in the remote directory. If the destination file already exists, *ftp* replaces its contents with the source file's contents.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  PORT command okay.

  Opening data connection for remote_file_name...

  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

### From a Local File
### To the End of a Remote File with a Different Directory Path and/or Name

**At the ftp > prompt, enter:**

append *local_file_path remote_file_path*

where *local_file_path* is a full or relative path to the source file in a local directory and *remote_file_path* is a full or relative path to the destination file in a remote directory.

---

**Note**

*Remote_file_path* **must** include the destination file's name.

---

**Example Entries:**

```
append /users/richard/doc/section2 comment/chapter1

append aliases .login

append email/bugreport /users/lab/richard/project/defects
```

**Result:**

*Ftp* appends the file from the local directory to the end of the file in the remote directory. If the destination file does not exist, *ftp* creates it.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
PORT command okay.

Opening data connection for remote_file_name...

Transfer complete.

number bytes received in number seconds...

ftp>
```

### From Multiple Local Files
### To Multiple Remote Files with the Same Directory Paths and Names

Ftp behaves in the following way when you transfer multiple files from a local host to a remote host:

- *Ftp* copies files from the local working directory to files with the **same names** in the remote working directory. The local and remote working directory paths may be different.
- *Ftp* copies files from any other local directory to remote files with the **same directory paths** and **names** you identify the local files with.

---

**Note**

Metacharacters are expanded for *ftp*'s *mput* command if globbing is on.

---

#### If Interactive Mode Is On

**1. At the ftp> prompt, enter:**

`mput` *local_file_path ...*

where *local_file_path* is either:

- the name of a file if the file is in the local working directory or
- the full or relative path to a file if the file is in another local directory.

The ellipsis (...) means that you can specify multiple *local_file_path*s.

---

**Note**

Any local **directory path** you specify as part of a *local_file_path* **must also exist on the remote host**. Otherwise, *ftp* will not transfer the file(s).

---

**Example Entries:**

```
mput /etc/*
```

```
mput memo1 memo2 memo3
```

**Result:**

*Ftp* asks if you want to transfer the first local file that matches what you specified. This gives you the option of **not** transferring the local file.

*Ftp* displays:

mput *first_local_file_path*?

2. **To not transfer the local file, enter:**

   ```
   N
   ```

   **To transfer the local file, enter:**

   ```
   Y
   ```

   **Result:**

   If you enter `N`, *ftp* does **not** transfer the local file and asks if you want to transfer the next local file that matches what you specified, displaying:

   mput *next_local_file_path*?

If you enter Y, and:

- the file is in the local working directory, *ftp* copies the file to a file with the same name in the remote working directory.
- the file is in another local directory, *ftp* copies the local file to a remote file with the same directory path and name.

Then *ftp* asks if you want to transfer the next local file that matches what you specified.

- If verbose mode is off, *ftp* displays:

  `mput` *`next_local_file_path`*`?`

- If verbose mode is on, *ftp* displays:

  `PORT command okay.`

  `Opening data connection for` *`local_file_path...`*

  `Transfer complete.`

  *`number`* `bytes received in` *`number`* `seconds...`

  `mput` *`next_local_file_path`*`?`

**3. Repeat the previous step until ftp redisplays its ftp> prompt instead of the mput ...? prompt.**

This means that there are no more files that match what you originally entered.

### If Interactive Mode Is Off

**At the ftp > prompt, enter:**

`mput` *local_file_path ...*

where *local_file_path* is either:

- the name of a source file if the file is in the local working directory or
- the full or relative path to a source file if the file is in another local directory.

The ellipsis (...) means that you can specify multiple *local_file_path*s.

---

**Note**

Any local **directory path** you specify as part of a *local_file_path* **must also exist on the remote host**. Otherwise, *ftp* will not transfer the file(s).

---

**Example Entries:**

```
mput /etc/*
```

```
mput memo1 memo2 memo3
```

**Result:**

*Ftp* copies any files in the local working directory to files with the same names in the remote working directory.

*Ftp* copies any other local files to remote files with the same directory paths and names you identified the local files with.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
PORT command okay.

Opening data connection for remote_file_path...

Transfer complete.

number bytes received in number seconds...

PORT command okay.

Opening data connection for remote_file_path...

Transfer complete.

number bytes received in number seconds...
```

.
. (repeated for each file transferred)
.

```
ftp>
```

# Performing Other File Operations with Ftp

From within *ftp*, you can:

- display the contents of a remote file,
- create a remote file,
- append text to the end of a remote file,
- delete one or more remote files, and
- change the name of a remote file.

The following sections tell how to perform these operations.

## Displaying the Contents of a Remote File

To display the contents of a remote file, you transfer the remote file to *stdout* (usually your display, or HP-UX terminal).

**At the ftp> prompt, enter:**

`get` *remote_file_path* –

**or**

`recv` *remote_file_path* –

where *remote_file_path* is a full or relative path to a remote file, and – represents *stdout* (usually the display).

**Example Entries:**

```
get /users/lab/richard/comment/readme -
```

```
recv comment/readme -
```

**Result:**

*Ftp* sends the contents of the remote file you specify to the display (actually, *stdout*).

- If verbose mode is off, *ftp* displays:

  *contents_of_file*

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  PORT command okay.

  Opening data connection for *remote_file*...

  *contents_of_file*

  Transfer complete.

  *number* bytes received in *number* seconds...

  ftp>

## Creating a Remote File

To create a remote file, you transfer from *stdin* (usually your keyboard input, or HP-UX terminal) to the remote file.

**1. At the ftp> prompt, enter:**

`put` – *remote_file_path*

**or**

`send` – *remote_file_path*

where – represents *stdin* (usually keyboard input), and *remote_file_path* is a full or relative path to a remote file.

**Example Entries:**

```
put - /users/lab/richard/comment/note
```

```
send  -  comment/note
```

**Result:**

*Ftp* creates the file you specify and waits for you to enter what you want to put into the file.

- If verbose mode is off, *ftp* displays nothing.
- If verbose mode is on, *ftp* displays:

```
PORT command okay.

Opening data connection for remote_file...
```

**2. At the keyboard, enter what you want to put into the file.**

**Example Entry:**

```
This is a test.

I am entering words into the file I am creating.

This is the last line.
```

**Result:**

*Ftp* displays what you enter as you enter it.

**3. When you finish entering the contents of the file, press Return.**

**Result:**

The cursor moves to a new line.

**4. Press CTRL-D.**

**Result:**

This signals the end of the file, and *ftp* adds what you entered to the file.

- If verbose mode is off, *ftp* displays:

  ```
  ftp>
  ```

- If verbose mode is on, *ftp* displays:

  ```
  Transfer complete.

  number bytes received in number seconds...

  ftp>
  ```

## Appending Text to the End of a Remote File

To append text to the end of a remote file, you append from *stdin* (usually your keyboard input, or HP-UX terminal) to the end of the remote file.

**1. At the ftp> prompt, enter:**

append – *remote_file_path*

where – represents *stdin* (usually keyboard input) and *remote_file_path* is a full or relative path to a remote file.

**Example Entries:**

```
append - /users/lab/richard/comment/note
```

```
append - comment/note
```

**Result:**

*Ftp* waits for you to enter what you want to append to the end of the file.

- If verbose mode is off, *ftp* displays nothing.
- If verbose mode is on, *ftp* displays:

  PORT command okay.

  Opening data connection for *remote_file*...

**2. At the keyboard, enter what you want to append to the end of the file.**

**Example Entry:**

```
These are words that I want appended to the end of a file.
```

```
This is the last line.
```

**Result:**

*Ftp* displays what you enter as you enter it.

**3. When you finish entering what you want to append to the file, press Return.**

**Result:**

The cursor moves to a new line.

4. **Press CTRL-D.**

**Result:**

This signals the end of the file, and *ftp* appends what you entered to the end of the file.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Transfer complete.

number bytes received in number seconds...

ftp>
```

## Deleting a Remote File

**At the ftp> prompt, enter:**

`delete` *remote_file_path*

where *remote_file_path* is a full or relative path to a remote file.

**Result:**

*Ftp* deletes the remote file you specify.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
DELE command okay.

ftp>
```

## Deleting Multiple Remote Files

---

**Note**

Metacharacters are expanded for *ftp*'s *mdelete* command if globbing is on.

---

### If Interactive Mode Is On

**1. At the ftp > prompt, enter:**

mdelete *remote_file_path ...*

where *remote_file_path* is a full or relative path to a remote file. The ellipsis (...) means that you can specify multiple *remote_file_path*s.

**Example Entries:**

```
mdelete /users/lab/richard/doc/spec?
```

```
mdelete doc/spec1 doc/spec2 doc/spec3
```

**Result:**

*Ftp* asks if you want to delete the first remote file that matches what you specified. This gives you the option of keeping the remote file.

*Ftp* displays:

mdelete *first_remote_file_path*?

**2. To keep the remote file, enter:**

```
N
```

**To delete the remote file, enter:**

```
Y
```

**Result:**

If you enter N, *ftp* keeps the remote file and asks if you want to delete the next remote file that matches what you specified, displaying:

```
mdelete next_remote_file_path?
```

If you enter Y, *ftp* deletes the remote file and asks if you want to delete the next remote file that matches what you specified.

- If verbose mode is off, *ftp* displays:

  ```
  mdelete next_remote_file_path?
  ```

- If verbose mode is on, *ftp* displays:

  ```
  DELE command okay.

  mdelete next_remote_file_path?
  ```

**3. Repeat the previous step until ftp redisplays its ftp> prompt instead of the mdelete ...? prompt.**

This means that there are no more files that match what you originally entered.

### If Interactive Mode Is Off

**At the ftp > prompt, enter:**

`mdelete` *remote_file_path* ...

where *remote_file_path* is a full or relative path to a remote file. The ellipsis (...) means that you can specify multiple *remote_file_path*s.

**Example Entries:**

```
mdelete /users/lab/richard/doc/spec?
```

```
mdelete doc/spec1 doc/spec2 doc/spec3
```

**Result:**

*Ftp* deletes **all** of the files that match what you specified.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
DELE command okay.

DELE command okay.

.
. (repeated for each file deleted)
.

ftp>
```

## Changing the Name of a Remote File

**Note**

You can use *ftp*'s *rename* command to change the path to (move) a remote file. You can **not** use *ftp*'s *rename* command to change the path to (move) a remote directory.

**At the ftp> prompt, enter:**

`rename` *old_remote_file_path new_remote_file_path*

where *old_remote_file_path* is a full or relative path to an existing remote file, and *new_remote_file_path* is a full or relative path to a new file.

**Note**

If you do not specify *new_remote_file_path*, *ftp* prompts you for it by displaying:

```
(to-name)
```

**Example Entries:**

```
rename /users/lab/richard/doc/note /users/lab/richard/memo
```

```
rename memo memo.tmp
```

**Result:**

*Ftp* changes the name and/or the path to the remote file.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
File exists, ready for destination name.

RNTO command okay.
```

# Obtaining Ftp Status

You can display the status of all of *ftp*'s feature settings. These include:

- whether or not a connection to a remote host exists and the name of the host to which a connection exists (if any),
- the file transfer type,
- whether or not *ftp* is set to sound a bell after each file transfer completes,
- whether or not interactive mode is on for multiple-file operations,
- whether or not globbing is on,
- whether or not *ftp* is set to display a hash mark for each 1024 bytes transferred during a file transfer, and
- other status information for settings that you cannot change or that need not be used in HP's implementation of *ftp*.

**At the ftp > prompt, enter:**

```
status
```

**Result:**

*Ftp* displays the status of all of its feature settings.

# Setting Up Automatic Remote Login for Ftp

You can create a file named *.netrc* that lets *ftp* log you into a remote host automatically. You place remote login names and passwords in this file so that *ftp* need not prompt you for these. This feature can be useful for programs that need to perform *ftp* operations unattended.

---

**Caution**

Having a *.netrc* file is a serious security risk. Passwords in this file are unencrypted. Be sure to follow the directions below for "Protecting Your .netrc File."

---

Your *.netrc* file must be in your home directory on your local host. You can find out what your home directory is by entering:

```
echo $HOME
```

Each entry you place in your *.netrc* file must have the following format:

`machine` *remote_host_name* `login` *remote_login_name* `password` *remote_password*

Follow these rules when creating a *.netrc* file:

- Each entry must contain a valid remote host name, remote login name and remote password. Valid remote host names are listed in the */etc/hosts* file on your local host.
- Separate each word in an entry with tabs, commas or blanks.
- Follow each remote host name with only one remote login name and one remote password.

**Example $HOME/.netrc File Entry:**

If you wanted to set up automatic login to the remote host *hpabsa*, and your remote login name and password on that host were *carolyn* and *driveway*, respectively, you would create a *$HOME/.netrc* file with the following entry:

```
machine hpabsa login carolyn password driveway
```

## Protecting Your .netrc File

It is important to protect your *.netrc* file and your home directory to prevent unauthorized users from gaining access to the remote hosts and accounts in your *.netrc* file. To do so:

1. Insure that your *.netrc* file is owned by you, the user.

2. Use the HP-UX *chmod* command to protect your *.netrc* file with 0400 (-r--------) permission.

3. Use the HP-UX *chmod* command to protect your home directory so that no one else can read it or write to it. For example, you should protect your home directory with at least 0711 (-rwx--x--x) permission.

4. Insure that your local account has a password.

To automatically log into a remote host once your *.netrc* file is set up, just invoke *ftp* and open a connection to the remote host. *Ftp* then automatically logs you into that host.

## Logging into a Remote Host with a Login Not in Your .netrc File

If you need to log into a remote host as someone else, you can override the automatic login (*.netrc* file) you set up for *ftp*. To do so, you:

- invoke *ftp* with the **–n** option to disable automatic login,
- choose whether you want *ftp* to display responses from the remote host (choose whether you want verbose mode on or off),
- connect to the remote host, and then
- log into the remote host with *ftp*'s *user* command.

### 1. Invoke Ftp with the –n Option

**At your HP-UX prompt, enter:**

```
ftp -n
```

**Result:**

*Ftp* displays its prompt:

```
ftp>
```

---

**Note**

Automatic login remains disabled for your entire *ftp* session.

---

### 2. Choose Whether You Want Verbose Mode On or Off

If you want *ftp* to display responses from the remote host, insure that verbose mode is on. Otherwise turn it off.

Use *ftp*'s *status* command to check the verbose mode setting, and use *ftp*'s *verbose* command to change the setting if you want to. These two *ftp* commands are explained in earlier sections of this chapter.

## 3. Connect to the Remote Host

Connect to the remote host as you normally would with *ftp*'s *open* command. This command is explained earlier in this chapter.

## 4. Log into the Remote Host with Ftp's User Command

There are two ways to log into a remote host with *ftp*'s *user* command. The faster way allows you to enter all of the login information on one line, but displays the remote password as you enter it. The other way causes *ftp* to prompt you for the remote password and does not display the remote password as you enter it.

### Logging into a Remote Host with a Single Command Line

---

**Caution**

This method displays the remote password as you enter it.

---

**At the ftp> prompt, enter:**

`user` *remote_login_name remote_password* [*account*]

where *remote_login_name* and *remote_password* must be valid on the remote host. Some remote hosts require you to enter a valid account name.

**Example Entry:**

```
user richard soccer
```

**Result:**

*Ftp* checks the remote login name, password, and account (if applicable) for validity and logs you into the remote host if they are valid.

- If verbose mode is off, *ftp* displays:

```
ftp>
```

- If verbose mode is on, *ftp* displays:

```
Password required for remote_login_name.

User remote_login_name logged in.

ftp>
```

### Logging into a Remote Host without Displaying the Remote Password

**1. At the ftp> prompt, enter:**

`user` *remote_login_name*

where *remote_login_name* must be valid on the remote host.

**Example Entry:**

```
user richard
```

**Result:**

*Ftp* prompts you for the remote password associated with the remote login name you entered.

- If verbose mode is off, *ftp* displays:

  `Password:`

- If verbose mode is on, *ftp* displays:

  `Password required for` *`remote_login_name`*`.`

  `Password:`

2. **Enter the remote password associated with the remote login name you gave.** (The password is not displayed as you enter it.)

   **Result:**

   *Ftp* checks the remote password for validity and logs you into the remote host if your password is valid. Some remote hosts may require you to enter a valid account name before you are logged in.

   - If verbose mode is off, *ftp* displays:

     `ftp>`

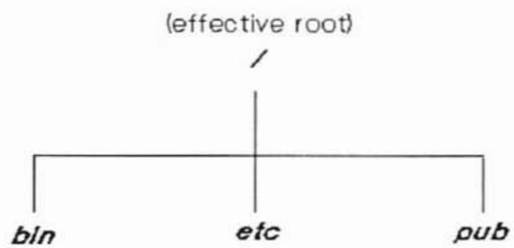
   - If verbose mode is on, *ftp* displays:

     `User` *`remote_login_name`* `logged in.`

     `ftp>`

# The Public Ftp Account

Some remote hosts may have a public (guest or anonymous) *ftp* account. When you log into this account, it becomes your **effective** root directory, and you cannot access anything above it. A public *ftp* account has the following directory structure:




**User's View of Public Ftp Directory Structure**

| Directory | Description |
|---|---|
| *bin* | This directory contains copies of the *ls(1)*, *sh(1)* and *csh(1)* programs to support *ftp*'s *dir*, *ls*, and *pwd* commands. |
| *etc* | This directory contains copies of the files *passwd(1)* and *group(1)*. These files must be present for *ftp*'s *dir*, *ls*, and *pwd* commands to work properly. |
| *pub* | Users can place files in this directory for public access. For example, this directory might contain announcements, requests for comment, or host tables. |

## Logging into the Public (Anonymous) Ftp Account

1. **Invoke *ftp*, insure that verbose mode is set to what you want, and connect to a remote host as you normally would.**

2. **When *ftp* prompts for a remote login name with the Name (...): prompt, enter:**

   ```
   ftp
   ```

   **Result:**

   *Ftp* prompts for the remote password associated with the *ftp* remote login name, displaying:

   Password (*remote_host*:ftp):

3. **Enter the name of your local host.**

   **Example Entry:**

   ```
   hpabsa
   ```

   **Result:**

   If a public (guest or anonymous) *ftp* account exists on the remote host, *ftp* logs you into that account and makes the *ftp* directory your working and effective root directory.

   - If verbose mode is off, *ftp* displays:

     ```
     ftp>
     ```

   - If verbose mode is on, *ftp* displays:

     ```
     Guest login ok, send ident as password.

     Guest login ok, access restrictions apply.

     ftp>
     ```

## Specifying Ftp Settings and Connecting to a Remote Host When You Invoke Ftp

You can change some of *ftp*'s settings when you first invoke *ftp*, if you know ahead of time how you want *ftp* set up. You can also connect to a remote host when you invoke *ftp*.

The *ftp* command you give from your local HP-UX prompt can take the following form:

`ftp` [`-g`] [`-i`] [`-n`] [`-v`] [*remote_host*]

Anything in brackets is optional. The following table explains the effect of each option above.

| Option | Effect |
|---|---|
| **–g** | This option turns off globbing (metacharacter expansion). |
| **–i** | This option turns off interactive mode for multiple-file operations. This is useful if, for example, you know ahead of time that you want to use *ftp* to perform bulk, instead of selective, file deletions or file transfers. |
| **–n** | This option disables automatic login (as set up by a $HOME/*.netrc* file). Use of this option is explained in an earlier section of this chapter. |
| **–v** | This option turns on verbose output (the display of responses from any remote host you connect to). If you invoke *ftp* from your keyboard (HP-UX terminal), *ftp* turns on verbose output without this option. This option is only useful if *ftp* is invoked indirectly. For example, if a program invokes *ftp* with this option, and *ftp*'s output is going to a file, the output file will contain a "log" of *ftp*'s results. |

Specifying a remote host's name or alias (as listed in your local host's */etc/hosts* file) on the *ftp* command line causes *ftp* to connect to that remote host without your having to use *ftp*'s *open* command.

# 9

# Transferring Files with Rcp

*Rcp* is a Berkeley Service that allows you to copy files between only HP-UX or UNIX hosts on the network.

*Rcp* can copy the contents of an entire directory. This includes all files and the contents of all subdirectories within that directory. From your local host, you can also copy files between two remote hosts.

*Rcp* allows file transfers to and from other hosts only if the configuration files that this service uses are set up properly. These files are mentioned later in this chapter.

## File Copy Concepts

With *rcp* you can copy files and directories

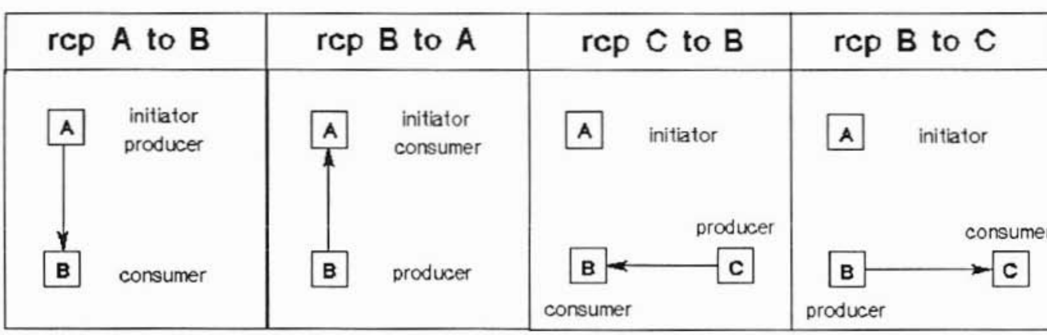
| From | To |
|---|---|
| your local host | a remote host |
| a remote host | your local host |
| a remote host | another remote host |

These scenarios can be represented by the **initiator/producer/consumer** model.

In any *rcp* operation, there may be up to three hosts involved: an initiator, a producer and a consumer.

- The **initiator** is your local host, the host on which you make the *rcp* request. It receives requests from you and starts file copies.
- The **producer** is the host that the source file or directory is on. It accesses the source file or directory and produces the data to be copied.
- The **consumer** is the host that the destination file or directory resides on or will reside on. It receives the incoming data and writes it to the destination file or directory.

In the diagram below, the arrows represent data being copied from a source file to a destination file. If host A is the initiator in each case, the location of the producer and consumer depends on the location of the source and destination files.




**Initiator/Producer/Consumer Model**

# Using Rcp

*Rcp* allows you to copy files to or from a remote host if the remote host is configured in either of two ways.

**Either:**

- you must have an account on the remote host with the **same** login name as your local login name, **and**

- the name of your local host must be in the remote host's */etc/hosts.equiv* file,

**or:**

- you must have an account on the remote host, **and**

- the name of your local host **and** your local login name must be in a *.rhosts* file in your home directory on the remote host.

The next section explains how to create a remote *$HOME/.rhosts* file for yourself, if you need to do so. Otherwise, skip the next section.

## Creating a $HOME/.rhosts File on a Remote Host

If you have an account on a remote host, you can give yourself *rcp* access to your remote account by creating a file named *.rhosts* in your remote home directory. You can find out what your remote home directory is by entering:

```
echo $HOME
```

on the remote host. You must place the name of your local host and your local login name in the *.rhosts* file you create.

### Note

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the directions below for "Protecting Your .rhosts File."

The entry you place in the remote *.rhosts* file must have the following format:

*your_local_host's_name your_local_login_name*

You can separate *your_local_host's_name* and *your_local_login_name* with any number of tabs or spaces. Put any comments after *your_local_login_name*.

**Example $HOME/.rhosts File Entry**

If your local host's name were *hpabsa* and your local login name were *richard*, on the remote host you would create a *$HOME/.rhosts* file with the following entry:

```
hpabsa richard
```

## Protecting Your $HOME/.rhosts File

It is important to protect your remote *.rhosts* file and home directory to prevent unauthorized users from gaining *rcp* access to your remote account and host. Only you should be able to create a *.rhosts* file in your remote home directory and write entries to the file. To do so:

**1.** Insure that your remote *.rhosts* file is owned by you, the user.

**2.** Use the HP-UX or UNIX *chmod* command to protect your remote *.rhosts* file with 0400 (-r--------) permission.

**3.** Use the HP-UX or UNIX *chmod* command to protect your remote home directory so that no one else can read it or write to it. For example, you should protect your remote home directory with at least 0711 (-rwx--x--x) permission.

# Performing Copy Operations with Rcp

---

**Note**

When you copy remote files and directories, the working directory for *rcp* on the remote host is your remote *$HOME* directory.

---

With *rcp*, you can copy from:

- a single local or remote file,
- multiple local and/or remote files,
- a single local or remote directory,
- multiple local and/or remote directories,
- any combination of local and/or remote files and directories.

What *rcp* can copy **to** depends on what *rcp* is copying **from**.

---

**Note**

*Rcp* can copy **from** only ordinary files and directories, **not** special files and directories (such as */dev* files). However, *rcp* can copy **to** special files (such as */dev* files).

---

**Note**

In *rcp* file transfers you **must** explicitly specify the destination file or directory.

Any output generated by commands in a *.login*, *.profile*, or *.cshrc* file on the remote host can cause *rcp* errors.

**Caution**

Do not specify the same source and destination files. This can corrupt the file's contents.

When *rcp* completes a copy operation, your local host redisplays its prompt.

The following sections explain all of *rcp*'s copy options.

## From a Local Producer to a Remote Consumer

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Local File | A New Remote File | *Rcp* creates the destination file and copies the source file's contents into the destination file. |
| | An Existing Remote File | *Rcp* overwrites the destination file's contents with the source file's contents. |
| | An Existing Remote Directory | *Rcp* copies the source file into the destination directory. |
| Local Files | An Existing Remote Directory | *Rcp* copies the source files into the destination directory. |

If the destination is a link to a file, the file to which the destination is linked is overwritten, and all links to the file remain the same.

**At your HP-UX prompt, enter:**

`rcp` *local_path ... remote_host:remote_path*

where:

- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipsis (...) means that you can specify multiple *local_paths*,

- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp /users/alan/program mail/defects hpabsb:project
```

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Local Directory | A New Remote Directory | *Rcp* creates the destination directory and copies the contents of the source directory into the destination directory. |
| | An Existing Remote Directory | *Rcp* copies **the source directory itself** along with its contents into the destination directory. *Rcp* overwrites any existing files. |
| Local Directories | An Existing Remote Directory | *Rcp* copies **the source directories themselves** along with their contents into the destination directory. |
| Any Combination of Local Files and Directories | An Existing Remote Directory | *Rcp* copies the source files and **the source directories themselves** along with their contents into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp -r` *local_path ...* *remote_host:remote_path*

where:

- **–r** (recursive option) causes *rcp* to copy the contents of any source directories,
- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipsis (...) means that you can specify multiple *local_path*s,

- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp -r mail /users/alan/memos hpabsb:correspondence
```

## From One or More Remote Producers to a Local Consumer

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Remote File | A New Local File | *Rcp* creates the destination file and copies the source file's contents into the destination file. |
| | An Existing Local File | *Rcp* overwrites the destination file's contents with the source file's contents. |
| | An Existing Local Directory | *Rcp* copies the source file into the destination directory. |
| Remote Files | An Existing Local Directory | *Rcp* copies the source files into the destination directory. |

If the destination is a link to a file, the file to which the destination is linked is overwritten, and all links to the file remain the same.

**At your HP-UX prompt, enter:**

`rcp` *remote_host:remote_path ...* *local_path*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*,

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory,
- the ellipsis (...) means that you can specify multiple *remote_path*s, and
- *local_path* is the path relative to your local working directory or the full path from the local root directory.

**Example Entry:**

```
rcp hpabsb:/users/alan/graphics/logo hpabsb:form templates
```

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Remote Directory | A New Local Directory | *Rcp* creates the destination directory and copies the contents of the source directory into the destination directory. |
| | An Existing Local Directory | *Rcp* copies **the source directory itself** along with its contents into the destination directory. *Rcp* overwrites any existing files. |
| Remote Directories | An Existing Local Directory | *Rcp* copies **the source directories themselves** along with their contents into the destination directory. |
| Any Combination of Remote Files and Directories | An Existing Local Directory | *Rcp* copies the source files and **the source directories themselves** along with their contents into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp -r` *remote_host:remote_path ...* *local_path*

where:

- **–r** (recursive option) causes *rcp* to copy the contents of any source directories,

- *remote_host* is the name or alias of a host listed in */etc/hosts*,

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory,
- the ellipsis (...) means that you can specify multiple *local_path*s, and
- *local_path* is the path relative to your local working directory or the full path from the local root directory.

**Example Entry:**

```
rcp -r hpabsb:/users/alan/document hpabsb:paper textfiles
```

## From One or More Remote Producers to a Remote Consumer

*Rcp* allows you to copy files between two remote hosts if the remote consumer host is configured in either of two ways:

**Either:**

- you must have an account on the remote consumer host with the **same** login name you have on the remote producer host, **and**
- the name of the remote producer host must be in the remote consumer host's */etc/hosts.equiv* file,

**or:**

- you must have an account on the remote consumer host, **and**
- the name of the remote producer host and your login name on the remote producer host must be in a *.rhosts* file in your home directory on the remote consumer host.

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Remote File | A New Remote File | *Rcp* creates the destination file and copies the source file's contents into the destination file. |
| | An Existing Remote File | *Rcp* overwrites the destination file's contents with the source file's contents. |
| | An Existing Remote Directory | *Rcp* copies the source file into the destination directory. |
| Remote Files | An Existing Remote Directory | *Rcp* copies the source files into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp` *remote_host:remote_path ...* *remote_host:remote_path*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*,

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory, and
- the ellipsis (...) means that you can specify multiple *remote_path*s.

**Example Entry:**

```
rcp hpabsb:graphics/logo hpabsb:form hpabsc:templates
```

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| A Remote Directory | A New Remote Directory | *Rcp* creates the destination directory and copies the contents of the source directory into the destination directory. |
| | An Existing Remote Directory | *Rcp* copies **the source directory itself** along with its contents into the destination directory. *Rcp* overwrites any existing files. |
| Remote Directories | An Existing Remote Directory | *Rcp* copies **the source directories themselves** along with their contents into the destination directory. |
| Any Combination of Remote Files and Directories | An Existing Remote Directory | *Rcp* copies the source files and **the source directories themselves** along with their contents into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp -r` *remote_host:remote_path ...* *remote_host:remote_path*

where:

- **-r** (recursive option) causes *rcp* to copy the contents of any source directories,

- *remote_host* is the name or alias of a host listed in */etc/hosts*,

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory, and
- the ellipsis (...) means that you can specify multiple *remote_paths*.

**Example Entry:**

```
rcp -r hpabsb:document hpabsb:paper hpabsc:textfiles
```

## From Local and Remote Producers to a Local Consumer

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| Local and Remote Files | An Existing Local Directory | *Rcp* copies the source files into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp` *local_path ... remote_host:remote_path ... local_path*

where:

- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipses (...) mean that you can specify multiple *local_path*s or *remote_path*s,
- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp module1 hpabsb:module2 /code/integration
```

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| Local and Remote Directories | An Existing Local Directory | *Rcp* copies **the source directories themselves** along with their contents into the destination directory. |
| Any Combination of Local and Remote Files and Directories | An Existing Local Directory | *Rcp* copies the source files and **the source directories themselves** along with their contents into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp -r` *local_path... remote_host:remote_path... local_path*

where:

- **-r** (recursive option) causes *rcp* to copy the contents of any source directories,
- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipsis (...) means that you can specify multiple *local_path*s,
- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp -r /users/alan/drawings hpabsc:charts /lib/graphics
```

## From Local and Remote Producers to a Remote Consumer

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| Local and Remote Files | An Existing Remote Directory | *Rcp* copies the source files into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp` *local_path... remote_host:remote_path... remote_host:remote_path*

where:

- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipses (...) mean that you can specify multiple *local_path*s or *remote_path*s,
- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp logfile1 hpabsb:logfile2 hpabsc:/tests/results
```

| Source | Allowed Destinations | Result of Copy |
|---|---|---|
| Local and Remote Directories | An Existing Remote Directory | *Rcp* copies **the source directories themselves** along with their contents into the destination directory. |
| Any Combination of Local and Remote Files and Directories | An Existing Remote Directory | *Rcp* copies the source files and **the source directories themselves** along with their contents into the destination directory. |

**At your HP-UX prompt, enter:**

`rcp -r` *local_path... remote_host:remote_path... remote_host:remote_path*

where:

- **-r** (recursive option) causes *rcp* to copy the contents of any source directories,
- *local_path* is the path relative to your local working directory or the full path from the local root directory,
- the ellipses (...) mean that you can specify multiple *local_path*s,
- *remote_host* is the name or alias of a host listed in */etc/hosts*, and

---

**Note**

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple *alias*es. The */etc/hosts* file may contain comments and other information as well.

---

- *remote_path* is the path relative to your remote home directory or the full path from the remote root directory.

**Example Entry:**

```
rcp -r /users/alan/reports hpabsb:article hpabsc:newsitems
```

## Rcp's Effect on File Attributes

*Rcp* sets the last access time of any source files and/or directories to the time that the copy occurs.

| **If the destination file or directory** | **exists,** | **does not exist,** |
|---|---|---|
| **then the mode is** | unchanged | same as source file |
| **owner is** | unchanged | same as user's |
| **group is** | unchanged | same as user's |
| **last access time is** | unchanged | set to the time when copy occurred |
| **last modification time is** | set to the time when copy occurred | set to the time when copy occurred |

## Using "Wild Card" Characters, or Metacharacters with Rcp

In an *rcp* command, a local file or directory path can contain any metacharacters that are allowed by the shell you are using. Metacharacters, or wild card characters, stand for a set of characters or character strings and are a "shorthand" way of specifying a set of directory or file names. Your local shell expands the metacharacters into the directory and file names they match before *rcp* performs the copy operation.

In an *rcp* command, for any metacharacters in a remote file or directory path to be expanded on the remote host, not on the local host, you must enclose each remote source path in single (‘’) or double (" ") quotes. You can also escape individual metacharacters by preceding them with a backslash (\) so that the remote host expands them.

**Example Entries:**

```
rcp -r hpabsb:"*.c" /users/alan/cprograms
```

```
rcp -r hpabsb:\*.c /users/alan/cprograms
```

Remember that if the source specification includes any directories (not files only), you must use the `rcp -r` syntax.

## Copying Remote Files and Directories as Someone Else on the Remote Host

With *rcp* you can assume the identity of another user on a remote host if:

- you know that user's login name on the remote host **and**
- that user has your local host name and local login name in a *.rhosts* file in his or her home directory on the remote host.

When you copy remote files and directories under these conditions, the working directory for *rcp* on the remote host is the user's remote home directory.

To assume the identity of another user on a remote host, you use the following syntax for a remote file or directory:

*remote_host.remote_login_name:remote_path*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*,
- *remote_login_name* is the login name of the remote user, and
- *remote_path* is the path relative to the remote user's home directory or the full path from the remote root directory.

**Example File Syntax:**

```
hpabsb.alan:cprograms/module1.c
```

## Giving Other Remote Users Rcp Access to Your Local Account

You can give remote users *rcp* access to your local account by creating a *.rhosts* file. You place remote users' host names and login names in this file so that *rcp* lets them assume your identity when copying files to or from your local host.

---

**Caution**

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the instructions below for "Protecting Your .rhosts File."

---

Your *.rhosts* file must be in your home directory on your local host.

Each entry you place in your local *.rhosts* file must have the following format:

*remote_host_name remote_login_name*

Follow these rules when creating a *.rhosts* file:

- Each entry must contain a valid remote host name and remote login name.
- Separate the host name and login name with any number of tabs or blanks.
- Put any comments after the login name in any entry.

### Example $HOME/.rhosts File Entry

If you wanted to give user *cdm* on remote host *hpabsb rcp* access to your local account, you would create a local *$HOME/.rhosts* file with the following entry:

```
hpabsb cdm
```

## Protecting Your .rhosts File

It is important to protect your local *.rhosts* file and your local home directory to prevent unauthorized users from gaining *rcp* access to your account and local host. Only you should be able to create a *.rhosts* file in your home directory and write entries to it. To do this:

**1.** Insure that your *.rhosts* file is owned by you, the user.

**2.** Use the HP-UX *chmod* command to protect your local *.rhosts* file with 0400 (-r--------) permission.

**3.** Use the HP-UX *chmod* command to protect your local home directory so that no one else can read it or write to it. For example, you should protect your local home directory with at least 0711 (-rwx--x--x) permission.

# 10

# Executing Commands with Remsh

*Remsh* is a Berkeley Service that allows you to execute commands on a remote HP-UX or UNIX host on the network. *Remsh* is the same command as *rsh* in 4.2 BSD and later versions.

This chapter will cover:

- Setting Up Permission to Use Remsh on a Remote Host
- Executing Commands on a Remote Host as Yourself
- Executing Commands on a Remote Host as Someone Else
- Giving Other Remote Users Remsh Access to Your Local Account
- Executing More Than One Remote Command with Remsh
- Using Shell Metacharacters with Remsh
- Using Remsh with Remote Commands That Do Not Take Input
- Using Remsh's "Shorthand" Syntax

# Setting Up Permission to Use Remsh on a Remote Host

**Caution**

Do **not** use *remsh* to run an interactive command, such as *vi* or *more*. With some interactive commands, *remsh* hangs. To run interactive commands, log into the remote host with *rlogin*.

*Remsh* allows you to execute a command on a remote host if the remote host is configured in either of two ways.

**Either:**

- you must have an account on the remote host with the **same** login name as your local login name, **and**
- the name of your local host must be in the remote host's */etc/hosts.equiv* file,

**or:**

- you must have an account on the remote host, **and**
- the name of your local host **and** your local login name must be in a *.rhosts* file in your home directory on the remote host.

The next section explains how to create a remote *$HOME/.rhosts* file for yourself, if you need to do so. Otherwise, skip the next section.

For more information about remote hosts, see the *hosts.equiv (4)* reference pages.

## Creating a $HOME/.rhosts File on a Remote Host

If you have an account on a remote host, you can give yourself *remsh* access to your remote account by creating a file named *.rhosts* in your remote home directory. You can find out what your remote home directory is by entering:

```
echo $HOME
```

on the remote host. You must place the name of your local host and your local login name in the *.rhosts* file you create.

---

**Caution**

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the directions below for "Protecting Your .rhosts File."

---

The entry you place in your *.rhosts* file must have the following format:

*your_local_host's_name*   *your_local_login_name*

You can separate *your_local_host's_name* and *your_local_login_name* with any number of tabs or spaces. Put any comments after *your_local_login_name*.

**Example $HOME/.rhosts File Entry**

If your local host's name were *hpabsa* and your local login name were *richard*, on the remote host you would create a *$HOME/.rhosts* file with the following entry:

```
hpabsa richard
```

### Protecting Your $HOME/.rhosts File

It is important to protect your remote *.rhosts* file and home directory to prevent unauthorized users from gaining *remsh* access to your remote account and host. Only you should be able to create a *.rhosts* file in your remote home directory and write entries to the file. To do so:

1. Insure that your remote *.rhosts* file is owned by you, the user.
2. Use the HP-UX or UNIX *chmod* command to protect your remote *.rhosts* file with 0400 (-r--------) permission.
3. Use the HP-UX or UNIX *chmod* command to protect your remote home directory so that no one else can read it or write to it. For example, you should protect your remote home directory with at least 0711 (-rwx--x--x) permission.

# Executing Commands on a Remote Host as Yourself

### Note

When you execute a command on a remote host, the working directory for *remsh* on the remote host is your remote *$HOME* directory.

*Remsh* passes interrupt, terminate, quit and hangup signals to the remote command you execute.

**At your HP-UX prompt, enter:**

`remsh` *remote_host command*

where *remote_host* is the name or alias of a host listed in */etc/hosts* and *command* is a non-interactive HP-UX or UNIX command to execute on the remote host.

### Note

The file */etc/hosts* contains entries for hosts with which you can communicate using ARPA/Berkeley Services. For each host, the file has a line containing the host's:

*internet_address official_name alias ...*

The ellipsis (...) means that a host may have multiple aliases. The */etc/hosts* file may contain comments and other information as well.

**Example Entry:**

```
remsh hpabsb cp form form.bkp
```

**Result:**

*Remsh* searches for the command you specify in the following remote directories in the order shown:

1. */bin*

2. */usr/bin*

3. */usr/contrib/bin*

4. */usr/local/bin*

On finding the command, *remsh* executes the command on the remote host and then your local host redisplays its prompt.

Note that if you do not give any command on the *remsh* command line, *remsh* interprets any options in the command line as *rlogin* options and runs *rlogin*.

# Executing Commands on a Remote Host as Someone Else

With *remsh* you can execute a command as another user on a remote host if that user has your local host name and local login name in a *.rhosts* file in his or her home directory on the remote host.

---

**Note**

When you execute a command under these conditions, the working directory for *remsh* on the remote host is the remote user's home directory.

---

If the remote user's account has no password, you can use *remsh* to execute remote commands as that user without having your local host name and local login name in the user's *$HOME/.rhosts* file.

**At your HP-UX prompt, enter:**

`remsh` *remote_host* `-l` *remote_login_name* *command*

where:

- *remote_host* is the name or alias of a host listed in */etc/hosts*,
- *remote_login_name* is the login name of the remote user who you want to execute the command as, and
- *command* is a command to execute on the remote host.

# Giving Other Remote Users Remsh Access to Your Local Account

You can give remote users *remsh* access to your local account by creating a *.rhosts* file in your local home directory. You place remote users' host names and login names in this file so that *remsh* lets them execute commands as you on your local host. (For more information, see the *hosts.equiv (4)* reference pages.)

---

**Caution**

A *$HOME/.rhosts* file creates a significant security risk. Be sure to follow the instructions below on "Protecting Your .rhosts File."

---

Your *.rhosts* file must be in your home directory on your local host.

Each entry you place in your *.rhosts* file must have the following format:

*remote_host_name   remote_login_name*

Follow these rules when creating a *.rhosts* file:

- Each entry must contain a valid remote host name and remote login name.
- Separate the host name and login name with any number of tabs or blanks.
- Put any comments after the login name in any entry.

### Example $HOME/.rhosts File Entry

If you wanted to give user *cdm* on remote host *hpabsc remsh* access to your local account, you would create a *$HOME/.rhosts* file on your local host with the following entry:

```
hpabsc cdm
```

## Protecting Your .rhosts File

It is important to protect your *.rhosts* file and your local home directory to prevent unauthorized users from gaining *remsh* access to your local account. Only you should be able to create a *.rhosts* file in your home directory and write entries to it. To do this:

1. Insure that your *.rhosts* file is owned by you, the user.

2. Use the HP-UX *chmod* command to protect your *.rhosts* file with 0400 (-r--------) permission.

3. Use the HP-UX *chmod* command to protect your local home directory so that no one else can read it or write to it. For example, you should protect your local home directory with at least 0711 (-rwx--x--x) permission.

4. Insure that your account has a password. Otherwise, anyone can execute commands as you (with your login name) on your local host.

# Executing More Than One Remote Command with Remsh

When you use *remsh* to execute more than one remote command, be aware of the following: a new remote shell executes the command(s) on each *remsh* command line. Therefore, when a remote command terminates, its process attributes (such as its environment and current working directory) disappear along with the shell that executed the command. For example, a remote *cd* command executed with *remsh* isolates the change of working directory to that instance of the command. A subsequent remote *pwd* command executed with *remsh* does not reflect the previous change in the working directory.

To execute more than one remote command without losing process attributes from one command to the next, you must:

- put the commands on a single *remsh* command line,
- separate the commands with semicolons (;), and
- enclose the string of commands in quotes (" ") so that the remote host executes every command. Otherwise, your local host executes any command(s) after the first one on the command line.

You can also have the remote host execute the string of commands by preceding each semicolon separator with a back slash (\).

**At your HP-UX prompt, enter:**

`remsh` *remote_host "command;...command"*

**or**

`remsh` *remote_host command* \;...*command*

where *remote_host* is the name or alias of a host listed in */etc/hosts*, and *command* is a non-interactive HP-UX or UNIX command to execute on the remote host. The ellipsis (...) means that you can specify more than one *command*; or *command* \;.

**Example Entries:**

```
remsh hpabsb "pwd; cd reports; pwd"
```

```
remsh hpabsb pwd \; cd reports \; pwd
```

**Result:**

*Remsh* executes the commands in sequence on the remote host, and your local host redisplays its prompt. Each remote command inherits the preceding one's process attributes. For example, the last *pwd* command in the example entries above would show *reports* as the remote working directory.

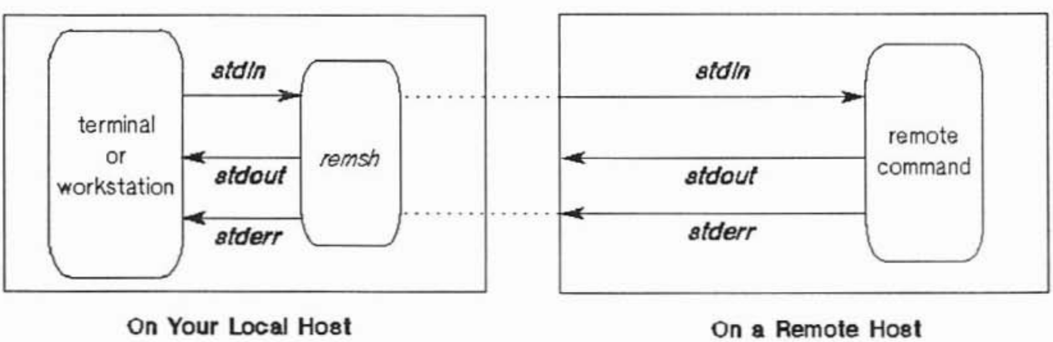
# Using Shell Metacharacters with Remsh

Commands can contain metacharacters to be interpreted on either the local host or the remote host. The metacharacters you may use are those allowed by the shell you are using.

| To Have Metacharacters Interpreted | Do This |
|---|---|
| On Your Local Host | Specify them as you normally would for local commands. |
| On a Remote Host | Enclose them in double quotes (" "). |

## Stdin, Stdout, and Stderr for Remsh

*Remsh*'s *stdin* becomes the remote command's *stdin*, and the remote command's *stdout* and *stderr* become *remsh*'s *stdout* and *stderr*, as illustrated below.



This means that you can use metacharacters to:

- redirect a remote command's input (*stdin*), output (*stdout*) and diagnostic output (*stderr*),
- pipe the output of a remote command into another remote or local command, and
- pipe the output of another remote or local command into a remote command.

For example, the command

```
remsh hpabsb cat remotefile > localfile
```

appends the remote file *remotefile* to the local file *localfile*. In contrast, the command

```
remsh hpabsb cat remotefile1 ">" remotefile2
```

appends the remote file *remotefile1* to the other remote file *remotefile2*.

The command

```
remsh hpabsb cat /tmp/broadcastmsg | wall
```

displays the message in the remote file *broadcastmsg* on all of the local host's terminals. In contrast, the command

```
remsh hpabsb cat /tmp/broadcastmsg "|" wall
```

displays the message in the remote file *broadcastmsg* on all of the remote host's terminals.

Enclosing the metacharacters in double quotes causes the remote host to interpret them, instead of the local host.

# Using Remsh with Remote Commands That Do Not Take Input

*Remsh* cannot determine if a remote command requires input. Therefore, *remsh* operates on the assumption that all remote commands require input. This behavior can cause problems if you use *remsh* to execute a remote command that does not require input. The *remsh* command attempts to read input (*stdin*) on the local host, even though the remote command requires none. The following examples illustrate this behavior.

### Example 1:

Suppose you enter a local command while *remsh* is running a remote command that requires no input. Normally, the command would go into your type-ahead buffer and would be executed as soon as the *remsh* command finished. Instead, *remsh* reads the local command as input and the local command never executes (your local shell never gets the command).

### Example 2:

This example involves shell scripts. Suppose you had a file named *text* with the following lines in it:

```
first line
second line
third line
```

and suppose that you wrote the following shell script named *test*:

```
#!/bin/sh
remsh hpabsb sleep 3
grep "second"
.
.
.
```

If you executed:

```
test < text
```

you would expect the shell script to find and display the line

```
second line
```

but instead the script displays nothing. This is because any command in the shell script (including *remsh*) inherits *stdin*, which is the input file *text*. Therefore *remsh* reads the file *text* as input and the following *grep* command never sees the file.

### Example 3 (for ksh and csh only):

Suppose you put the following command in the background:

```
remsh hpabsb echo hello &
```

Instead of seeing

```
hello
```

you see the following message after you enter the next carriage return:

```
[1] + stopped (tty input) remsh hpabsb echo hello
```

The *remsh* in the background tries to read its *stdin* (your terminal input). Since the shell does not allow background processes to read your terminal, the shell stops the background process, and notifies you.

In all 3 examples, to prevent such mishaps, *remsh* provides an option, **–n**, that redirects *remsh*'s input from */dev/null*. Whenever you use *remsh* to run a remote command that requires no input, it is good practice to invoke *remsh* with the **–n** option.

**At your HP-UX prompt, enter:**

`remsh` *remote_host* `-n` *command*

where *remote_host* is the name or alias of a host listed in */etc/hosts* and *command* is a command requiring no input to execute on the remote host.

**Example Entries:**

```
remsh hpabsb -n who

remsh hpabsb -n sleep 3

remsh hpabsb -n echo hello &
```

**Result:**

*Remsh* executes the command on the remote host, taking the command's input from */dev/null*.

## Using Remsh's "Shorthand" Syntax

Your local host can be configured so that you can enter a *remsh* command line without the *remsh* command. That is, a *remsh* command line can start with the name of a remote host, omitting the *remsh* command.

In order to do this:

- you must add the path */usr/hosts* to your command search path in your *.login*, *.cshrc*, or *.profile* file. Which file contains your *$PATH* variable depends on which shell you use.

- the super-user or node manager must link */usr/bin/remsh* to */usr/hosts/host*, where *host* is the name or alias of a remote host (listed in */etc/hosts*) on which you want to execute a command.

To find out which hosts you can use *remsh*'s shorthand syntax for, list the contents of the directory */usr/hosts*.

11

# Interprocess Communication

This chapter describes HP's implementation of the 4.2 BSD Interprocess Communication (IPC) facilities. The chapter includes the following sections:

- an IPC overview using the Client-Server model;
- a description of important terms and concepts;
- the details of IPC using stream sockets;
- advanced IPC concepts for stream sockets;
- the details of IPC using datagram sockets;
- advanced IPC concepts for datagram sockets;
- a list of programming hints;
- how to add a server process to *inetd*; and
- tables of the available system and library calls.

**Note**

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

# Overview of IPC

The 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 chapter describes the steps involved in establishing and using IPC connections. It also describes the protocols you must use and how the IPC system calls interact. The details of each system call are described in the section 2 entries of the *ARPA/Berkeley Services Reference Pages*.

To understand the general model for 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 an IPC connection is established. All 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 IPC

The best example of how IPC can be used is the ARPA/Berkeley Services themselves. The services use IPC to communicate between remote hosts. Using the 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; or
- spread subtasks across several hosts.

## The Client-Server Model

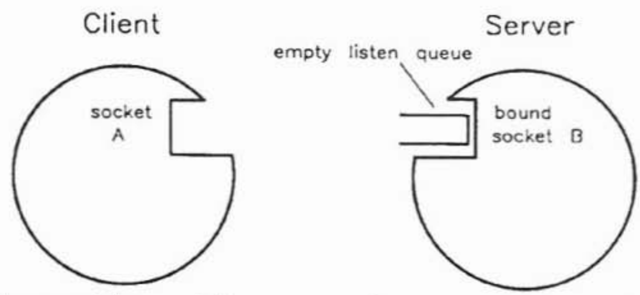
Typical IPC applications consist of two separate application level processes; one process (the **client**) requests a connection and the other process (the **server**) accepts it.

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 a connection is established, full-duplex (two-way) communication can occur between the two sockets.

This set of conventions must be implemented by both processes. Depending on 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

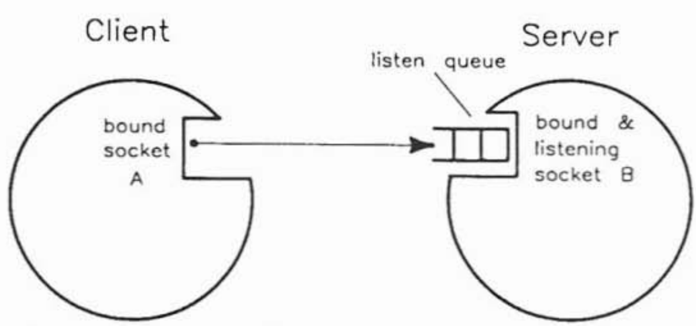
The following figures illustrate conceptual views of the client-server model at three different stages of establishing a connection. The steps that have been accomplished at each stage are listed below each figure.




- Client has created a socket.

- Server has created a socket.
- Server has bound an address to its socket.
- Server has set up the listen queue.

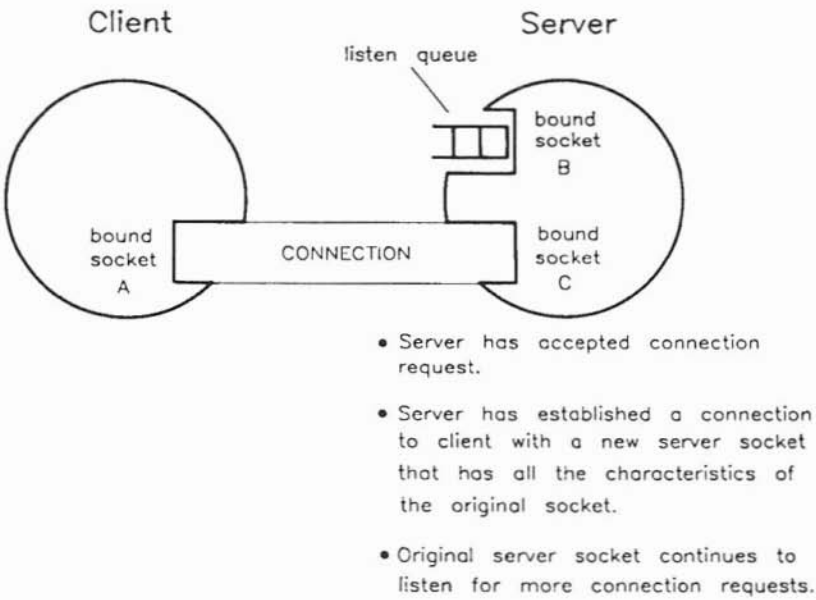
**Client-Server in a Pre-Connection State**




- Client has made a connection request.

- Server has received the request in the listen queue.

**Client-Server at Time of Connection Request**




**Client-Server When Connection Is Established**

A detailed description of the Client-Server model is discussed in the "IPC Using Stream Sockets" section of this chapter.

## IPC Library Routines

The library routines and system calls that you need to implement an 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 this chapter.

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 IPC. Many of the terms have more detailed explanations within this chapter 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 AF_INET (internet address family) domain is supported. The AF_UNIX (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 Berkeley UNIX 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.)

socket address
: For 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 AF_UNIX, the socket address is the directory path name of the vnode bound to the socket.

binding
: Associating 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
: An 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 the association is created in two steps and there is a chance that two associations could be alike. The host prevents this 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 IPC. They are TCP, which implements stream sockets, and UDP, which implements datagram sockets.

| | |
|---|---|
| TCP | Provides the underlying communication support for stream sockets. TCP is used to implement reliable, sequenced, flow-controlled two-way communication based on byte streams similar to pipes. Refer to the *TCP(7P)* entry in the *ARPA/Berkeley Services Reference Pages* for more information on TCP. |
| UDP | Provides the underlying communication support for datagram sockets. UDP is an unreliable protocol. A process receiving messages on a datagram socket could find messages 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 *ARPA/Berkeley Services Reference Pages* 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 LAN reference pages 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.

# IPC Using Internet Stream Sockets

This section describes the steps involved in creating an Internet stream socket IPC connection using the AF_INET address family. If you want to use datagram sockets, skip to the section called "IPC Using Datagram 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 an IPC connection using stream sockets. Each step is described in more detail in the sections that follow the table.

**Building an IPC Connection Using 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 connection | *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 IPC system calls; and
- 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; and
- 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; and
- 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 the "IPC Using Stream Sockets" section 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 IPC system calls are declared using a pointer to *sockaddr*, but it can also be 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 *ARPA/Berkeley Services Reference Pages* 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 |

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 *ARPA/Berkeley Services Reference Pages* 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 *ARPA/Berkeley Services 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 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.

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 an 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 IPC system calls |

Refer to the *socket(2)* entry in the *ARPA/Berkeley Services Reference Pages* 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 IPC system calls |

Refer to the *bind(2)* entry in the *ARPA/Berkeley Services Reference Pages* 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 *ARPA/Berkeley Services Reference Pages* 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; and
- is connected to the client process' socket.

*Accept* blocks until there is a connection request from a client process in the queue.

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

| Parameter | Description of 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 *ARPA/Berkeley Services Reference Pages* 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 an 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 *ARPA/Berkeley Services Reference Pages* 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 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_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 "Advanced Topics for Stream Sockets: Nonblocking I/O" section of this 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.

---

---

**Note**

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 *ARPA/Berkeley Services Reference Pages* 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 at the end of the 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 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 "Advanced Topics for Stream Sockets: Nonblocking I/O" section of this chapter.)

### When to Send Data

| Which Processes | When |
|---|---|
| server or client process | after connection is established |

Refer to the *send(2)* entry in the *ARPA/Berkeley Services Reference Pages* 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 "Advanced Topics for Stream Sockets: Nonblocking I/O" section of this 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 *flags* options are:

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

Use the MSG_OOB option if you want to receive out of band data. Refer to the "Advanced Topics for Stream Sockets, Sending and Receiving Out of Band Data" section of this chapter 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 *ARPA/Berkeley Services Reference Pages* 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 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 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. This delay in closing the socket can be controlled by the socket option SO_LINGER. See the "Socket Options" section for information on the SO_LINGER and SO_DONTLINGER options.

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 "Advanced Topics for Stream Sockets: Using Shutdown" section of this chapter.

## Example Using Stream Sockets

These program examples demonstrate how to set up and use stream sockets. 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; and
- uses the SO_LINGER option for a graceful disconnect. The SO_LINGER options is discussed in the "Socket Options" section, which follows the example.

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" section of this 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; and
- 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
```

- compile the programs with the `-lbsdipc` option.

The source code for these two programs follows. It is also located in the directory */usr/netdemo/socket*.

```
/*
 *                      S E R V . T C P
 *
 *      This is an example program that demonstrates the use of
 *      stream sockets as an 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.
 *      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 */

long linger = 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: example not found in /etc/services\n",
                                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);
	}
}
```

```
/*
 *                          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, (char *)&linger,
                                        sizeof(long)) == -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 beginning 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 an 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.  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 requests 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.
	 */
	for (i=1; i<=5; i++) {
		*(int *)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 beginning 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));
}
```

# 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. Datagram sockets are not currently supported for UNIX Domain.

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.

Building a UNIX Domain BSD IPC Connection Using 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 connection | *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; and
- 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*[108] | Specifies the pathname of the vnode 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 *LAN Reference Pages* 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 *LAN Reference Pages* 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 *LAN Reference Pages* 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; and
- is connected to the client process' socket.

*Accept* blocks until there is a connection request from a client process in the queue.

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

| Parameter | Description of 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 *LAN Reference Pages* 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 *LAN Reference Pages* 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<br>–1 if failure occurs |
| EXAMPLE SYSTEM CALL: | `struct sockaddr_un peeraddr;`<br>`...`<br>`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 *LAN Reference Pages* 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 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 *LAN Reference Pages* 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 *flags* 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 *LAN Reference Pages* 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 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.

## Examples Using UNIX Domain Stream Sockets

```
/*
 *      EXAMPLE PROGRAM
 *
 *      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
 *      the 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 "p_n_c"
#define BUFSIZE         32*1024-1

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 s, ns, recvsize, secs, usec;
        struct timezone tzp;

        struct sockaddr_un sa;

        signal(SIGPIPE, SIG_IGN);
        signal(SIGCLD, SIG_IGN);
        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);
}
```

```
/*
 *      EXAMPLE CLIENT PROGRAM
 *
 *      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 "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;

        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);
}
```

# Advanced Topics for Stream Sockets

## 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_DONTLINGER

The following options are supported for UNIX Domain stream sockets:

- SO_SNDBUF
- SO_RCVBUF

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.

## 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 | 0, or if optname = SO_LINGER, a pointer to the linger interval value of SO_LINGER, or if optname = SO_SNDBUF or SO_RCVBUF, a pointer to an integer containing the new buffer size value. |
| optlen | length of optval | 0 or 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.

For options that do not take an input value, *optval* and *optlen* should both be 0.

Refer to the *getsockopt(2)* entry in the *ARPA/Berkeley Services Reference Pages* 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 | Description of 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

–1 with errno = ENOPROTOOPT if specified option is not set

–1 with errno = some other value if failure occurs

EXAMPLE SYSTEM CALL:

```
getsockopt(s, SOL_SOCKET, SO_REUSEADDR, 0, 0);
```

For options that do not take an input value, *optval* and *optlen* should both be 0.

Refer to the *getsockopt(2)* entry in the *ARPA/Berkeley Services Reference Pages* section for more information on *getsockopt*.

## SO_REUSEADDR

Note that this option is not supported for UNIX Domain sockets.

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 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 Internet Protocol 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 Internet Protocol address (e.g. `*.2000`). The wildcard Internet Protocol address matches the Internet Protocol 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.

SO_REUSEADDR cannot be cleared once you set it.

An example usage of this option is:

```
setsockopt (s, SOL_SOCKET, SO_REUSEADDR, (char *)0, 0);
bind (s, &sin, sizeof(sin));
```

## SO_KEEPALIVE

Note that this option is not supported for UNIX Domain sockets.

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_KEEPALIVE cannot be cleared once you set it.

## SO_DONTROUTE

Note that this option is not supported for UNIX Domain sockets.

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 65535 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 65535 bytes.

**Example:**

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

### 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 | 65535 bytes |

## SO_LINGER

Note that this option is not supported for UNIX Domain sockets.

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

This option can be cleared by setting SO_DONTLINGER. The default is SO_DONTLINGER.

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

- If SO_LINGER is set with a nonzero timeout interval, the host blocks the *close* call until it is able to transmit the remaining data or until the protocol itself (TCP) expires. This is called a graceful disconnect.
- If SO_LINGER is set with a zero timeout interval, *close* **is not blocked even if queued data exist**. This is called a hard close, because it closes the socket immediately, whether data need to be sent or not. All unsent data are immediately lost.

**Example:**

```
int result;
int linger = 1;
result = setsockopt(s, SOL_SOCKET, SO_LINGER, &linger, sizeof(linger));
```

## SO_DONTLINGER

This option is the default. It can be overridden by setting SO_LINGER.

SO_DONTLINGER controls the actions taken when a *close* is executed on a socket. If SO_DONTLINGER is set on a stream socket with unsent data, the host allows the close call to return immediately, but it tells TCP to wait. Queued data are sent if possible, until TCP times out. This is also called a graceful disconnect.

**Summary of Linger Options on Close**

| Socket Option | Linger Interval | Graceful Close | Hard Close | Waits for Close | Does Not Wait for Close |
|---|---|---|---|---|---|
| SO_DONTLINGER | don't care | X | | | X |
| SO_LINGER | zero | | X | | X |
| SO_LINGER | nonzero | X | | X | |

## 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 **timout** parameter, *select* will block until one or more of the specified socket descriptors are 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 has arrived and is ready to be read without blocking; use the FIONREAD parameter to the *ioctl* system call to determine exactly how much data is available.

- a listening socket, because it determines when you can accept a connection without blocking.

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

- a connecting socket, because it determines when the 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.

Selecting for exceptional conditions is currently meaningless for Berkeley sockets. *Select* will always return true for sockets that 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 intergers. The following useful macros can be used to manipulate bit masks of this form.

```
#define BPI 32           /* bits per int */
#define FD_ZERO(p)    bzero((char *) (p), sizeof(*(p)))
#define FD_SET(n, p) ((p)-fdm_bits[(n)/BPI] |= (1 < ((n) % BPI)))
#define FD_CLR(n, p) ((p)-fdm_bits[(n)/BPI] &= ~(1 < ((n) % BPI)))
#define FD_ISSET(n, p)  ((p)-fdm_bits[(n)/BPI] & (1 < ((n) % BPI)))

struct fd_mask {
        u_long fdm_bits[NOFILE/BPI+1]    /* NOFILE max # of fd's per process */
};

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;               /* 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 a SIGIO signal when the socket's state changes. This state change can occur, for example, when new data arrives. Currently the user must issue a *select* system call to determine if data are 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 would 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* would return 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 aware that these assertions are guidelines only; any signal handler should be robust enough to handle signals in unexpected situations.

The delivery of 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. Secondly, 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; for more details, see the section in this chapter, "Sending and Receiving Out of Band Data."

### Example:

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

```
int ls;                         /* listen SOCK_STREAM socket */
        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 get 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 type could be 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 */
```

## 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*, or *read* call, the call returns immediately with the value of 0. 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 flag has precedence over the FIOSNBIO flag.

## 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 blocking 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 *ARPA/Berkeley Services Reference Pages* section for more information on *shutdown*.

## Using Read and Write to Make Stream Sockets Transparent

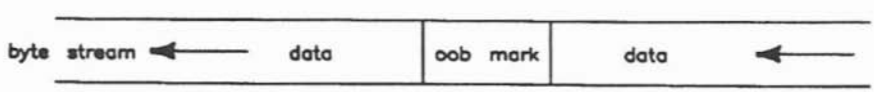
An example application of *read* and *write* with stream sockets is to fork a command with a socket descriptor as *stdout*. The peer process can *read* input from the command. The command can be any command and does not have to know that *stdout* is a socket. It might use *printf*, which results in the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

## Sending and Receiving Out of Band Data

Note that 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 arrives at the destination node in sequence and in stream, but is delivered independently of normal data; the out of band data receiver is notified with the SIGURG signal. 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 was sent, so that data before the message can be handled differently (if necessary) from data following the message.




**Data Stream with Out of Band Marker**

For a program to know when out of band data is 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)");
}
```

Refer to the *socket(7)* entry in the *ARPA/Berkeley Services Reference Pages* 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.

The out of band data byte is stored independently from the normal data stream. 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.

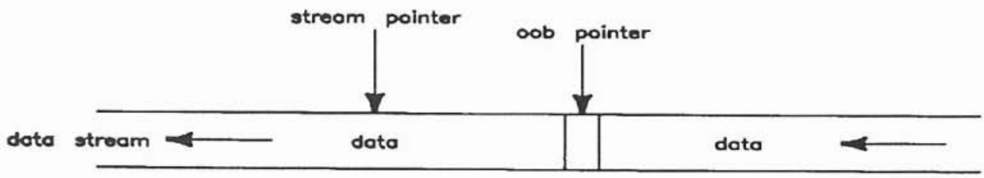
Usually the out of band data message indicates that all data currently in the stream can be flushed. This involves moving the stream pointer with successive *recv* calls, to the location of the out of band data pointer.

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.

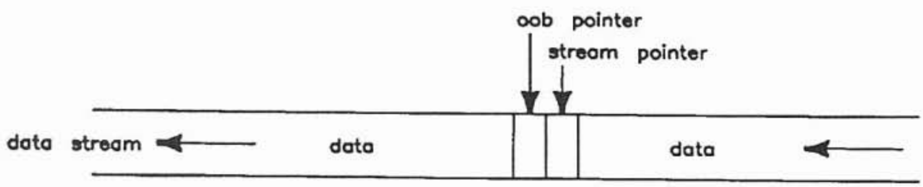
The following code segment illustrates how the SIOCATMARK request can be used in a SIGURG interrupt handler. The example also shows a buffer being flushed.

```
                    /*  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;
}
```




**Before Flushing Stream**




**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" section.

---

# IPC Using Internet Datagram Sockets

As discussed in the "Protocols" section, 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 are not supported for UNIX sockets.

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 server-client 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 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.

### 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()* | | |
| | | receive message | *recvfrom()* |
| | | send message | *sendto()* |
| receive message | *recvfrom()* | | |

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 IPC system calls; and
- 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 addresss variables;
- assign a wildcard address; and
- 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; and
- 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 the "IPC Using Datagram Sockets" section 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 IPC system calls are declared using a pointer to *sockaddr*, but it can also be 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 *ARPA/Berkeley Services 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*.

*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 *ARPA/Berkeley Services 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 *ARPA/Berkeley Services 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. See the "Advanced Topics for Datagram Sockets: Specifying a Default Socket Address" section for instructions.

**When to Create Sockets**

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

Refer to the *socket(2)* entry in the *ARPA/Berkeley Services 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>`<br>`#include <netinet/in.h>`<br>`#include <sys/socket.h>` |
| SYSTEM CALL: | `bind (s, addr, addrlen)`<br>`int s;`<br>`struct sockaddr *addr;`<br>`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:

```
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 IPC system calls |

Refer to the *bind(2)* entry in the *ARPA/Berkeley Services Reference Pages* for more information on *bind*.

## Sending and Receiving Messages

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

### Sending Messages

Use *sendto* to send messages.

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

*Send* is described in the "IPC Using Stream Sockets: Sending Data" section of this chapter and in the *send(2)* entry in the *ARPA/Berkeley Services 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));
```

If the message is too long to send as a single packet (largest size is 2860 bytes for this implementation), an error occurs.

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 *ARPA/Berkeley Services Reference Pages* for more information on *sendto*.

### Receiving Messages

Use *recvfrom* to receive messages.

*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.

*Recv* is described in the "IPC Using Stream Sockets: Receiving Data" section of this chapter and in the *recv(2)* entry in the *ARPA/Berkeley Services 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 | Description of 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*, *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 *flags* options are:

- 0 for no options or
- 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* returns 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 *ARPA/Berkeley Services Reference Pages* for more information on *recvfrom*.

## 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

These program examples demonstrate how to set up and use datagram sockets. 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 1's.

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

Before you run the example programs:

- make the following entry in the two hosts' */etc/services* files:

  ```
  example  22375/udp
  ```

- compile with the `-lbsdipc` option.

The source code for these two programs follows. It is also located in the directory */usr/netdemo/socket*.

```
/*
 *                  S E R V . U D P
 *
 *      This is an example program that demonstrates the use of
 *      datagram sockets as an 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.
 *
 *      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.
 *
 */
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: example not found in /etc/services\n",
                                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.
                 */
        close(stdin);
        close(stdout);
        close(stderr);
                /* This will open the /etc/hosts file and keep
                 * it open.  This will make accesses to it faster.
                 */
        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 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 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));
        }
}
```

# Advanced Topics for Internet Datagram Sockets

## 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*, 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* 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.

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* can 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 **timout** parameter, *select* will block until one or more of the specified socket descriptors are ready. If timeout 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 has arrived and is ready to be read without blocking; use the FIONREAD parameter to the *ioctl* system call to determine exactly how much data is available.

Selecting for exceptional conditions is currently meaningless for Berkeley sockets. *Select* will always return true for sockets that are no longer capable of being used (e.g. if a *close* or *shutdown* system call has been executed against them).

*Select* is used 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*.

## Sending and Receiving Data Asynchronously

Asynchronous sockets allow a user program to receive a SIGIO signal when the state of the socket changes. This state change can occur, for example, when new data arrives. A complete description of SIGIO can be found in the "Advanced Topics for Stream Sockets" 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. 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 flag has precedence over the FIOSNBIO flag.

---

## Using Broadcast Addresses

In place of a unique internet address or the wildcard address, you can also specify a broadcast address. The broadcast address is a local address portion of the internet address equal to all 1's. You must be the superuser to use a broadcast address.

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

# Programming Hints

---

**Note**

Refer to the "Portability Issues" appendix for information about the differences between 4.2 BSD and the HP-UX implementation of IPC.

---

## Troubleshooting

You can 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)*. Refer to the *ARPA/Berkeley Services Reference Pages* for information about errors returned by the 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 | NS daemon *rfadaemon* |

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, the *nftdaemon* will fail when you try to start it.

## 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 the *netstat* displays of 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 Maintaining NS-ARPA Services* manual.

# Adding a Server Process to the Internet Daemon

This section contains example IPC programs that use the internet daemon, called *inetd*. For more information on *inetd*, refer to the "Configuration and Maintenance" chapter of the *Installing and Maintaining NS-ARPA Services* manual and the *inetd(1M)* entry in the *ARPA/Berkeley Services Reference Pages*.

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 "Configuration and Maintenance" chapter of the *Installing and Maintaining NS-ARPA Services* manual or to the *inetd.conf(4)* entry in the *ARPA/Berkeley Services 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 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 (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 IPC system calls.

**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* | Sends data from the specified socket. |
| *recv, recvfrom* | Receives data at the specified socket. |
| *shutdown* | Disconnects the specified socket. |
| *getsockname* | Gets the socket address of the specified socket. |

| 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 IPC.

**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 "Closing a Socket" sections of this chapter. |
| *select* | Can be used to improve efficiency for a process that accesses multiple sockets or other I/O devices simultaneously. Refer to the "I/O Multiplexing with Select" sections of this chapter. |
| *ioctl* | Can be used for finding the number of receivable bytes with FIONREAD and for setting the nonblocking I/O flag. 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 flag. |

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 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.

### Library Calls

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

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

A

# Portability Issues

This appendix describes implementation differences between 4.2 BSD IPC and HP-UX IPC. It contains porting issues for:

- IPC functions and library calls; and for
- other functions and library calls typically used by IPC programs.

Because HP-UX IPC is based on 4.2 BSD IPC (it is a subset of 4.2 BSD), programs should port easily between HP-UX and 4.2 BSD systems. If you need to have portable applications, keep the information 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 be aware of if you want to port your IPC applications between HP-UX and 4.2 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 as to which pending connection on a listening socket will be 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.

## Errno Values

HP-UX IPC system calls have some *errno* values that are different from other implementations. These are listed in the following table.

### Errno Values that Differ for IPC System Calls

| System Call | Error | HP-UX Implementation | Other Implementations |
|---|---|---|---|
| *connect* | socket is a listening socket | EINVAL | ETIMEDOUT |
| *socket* | invalid socket type | EPROTONOSUPPORT and EPROTOTYPE | EPROTOTYPE |
| *socket* | invalid protocol | EPROTONOSUPPORT | EPROTOTYPE |

### Losing a TCP Connection

On a stream socket connection, if the connection has been lost due to some error, HP-UX systems return the same *errno* value for each subsequent *recv*. Some other implementations only return the error on the first *recv* after the connection is lost, and then return the end-of-file condition on subsequent *recv* calls.

### Unsupported IPC Features

The following is a list of 4.2 BSD IPC features which are not supported on HP-UX systems.

The HP-UX implementation does not support:

- AF_UNIX or other addressing domains (only AF_INET is supported);
- the use of *readv(2)* and *writev(2)* on sockets;
- the *sendmsg(2)* and *recvmsg(2)* system calls; or
- the SOCK_RAW socket type.

# 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 be aware of when you port your IPC applications between HP-UX and 4.2 BSD systems.

## Ioctl and Fcntl Calls

4.2 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*.

## Dup2

You must use the *–lBSD* compile option to use the 4.1 or 4.2 BSD version of the *dup2(2)* system call on an HP-UX system.

## Size Limit for Send

For Series 300 HP-UX systems, the maximum size message that can be sent on a datagram socket or on a nonblocking stream socket is 9216 bytes. For Series 800 HP-UX systems, the maximum size message is 2048 bytes. For 4.2 BSD systems, the maximum size message is 2048 bytes.

## Utmp

The 4.2 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(5)* entry in the *HP-UX Reference* manual for details.

## Library Equivalencies

Certain commonly used library calls in 4.2 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.

**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 4.2 BSD signals. See the *sigvector(2)* entry in the *HP-UX Reference* manual for information on signal implementation. Note the following signal mapping.

**Definitions of Signal Equivalents**

| 4.2 BSD Signal | is mapped to | HP-UX Signal |
|---|---|---|
| SIGCHLD | | SIGCLD |

### Sprintf Return Value

For 4.2 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

**Account name:** A synonym for user name or login name.

**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 a socket occurs. It is primarily used for sending and receiving data without blocking.

**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."

**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."

**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."

**Equivalent account:** An account (or user name) specified in the *$HOME/.rhosts* file that allows the specified remote users to access the local user's account without requiring a password.

**Equivalent host:** A remote host that is considered by your local host as an "equivalent computer." Users from equivalent hosts can bypass password validation if they have the same account name on both hosts.

**Equivalent user:** See "Equivalent account."

**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."

**Gateway:** A node that connects two or more networks together and routes packets between those networks.

**Host:** A node that has primary functions other than switching data for the network.

**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."

**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.

**Message:** In IPC, the data sent in one UDP packet. When using *sendmail* a message is the information unit transferred by mail.

**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.

**Official host name:** The first host name in each entry in the */etc/hosts* file. The official host name cannot be an alias.

**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).

**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.

**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)*).

**Simple Mail Transfer Protocol:** A standard protocol for transporting mail reliably and efficiently through LANs or the ARPANET. *Sendmail* uses the SMTP protocol.

**SMTP:** See "Simple Mail Transfer Protocol."

**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.

**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.

**UDP:** See "User Datagram Protocol."

**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.

**Virtual Terminal Protocol:** A protocol that provides terminal access to interactive services on remote hosts (e.g., *telnet(1)*).

**UNIX Domain Address:** A character string continuing 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.

# Index

## !

# A



# B



# C

## D



## E

# F

# G



# H

## I

## L

## M



## N



## O



## P

## Q



## R

# S

# T



# U



# V



# W

# Reader Comment Card

**HP 9000 Series 300**
**Using ARPA Services**
**50952-90001, E0189**

We welcome your evaluation of this manual. Your comments and suggestions will help us improve our publications. Please tear this card out and mail it in. Use and attach additional pages if necessary.

**Please circle the following Yes or No:**

| | | |
|---|---|---|
| • Is this manual well organized? | Yes | No |
| • Is the information technically accurate? | Yes | No |
| • Are instructions complete? | Yes | No |
| • Are concepts and wording easy to understand? | Yes | No |
| • Are examples and pictures helpful? | Yes | No |
| • Are there enough examples and pictures? | Yes | No |

**Comments:** ______________________________

______________________________

______________________________

______________________________

______________________________

Name: ______________________________

Title: ______________________________

Company: ______________________________

Address: ______________________________

City & State: ______________________________

Zip: ____________

HEWLETT PACKARD

Printed in USA

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 37 LOVELAND,CO

POSTAGE WILL BE PAID BY ADDRESSEE

**Hewlett-Packard Company**
**Colorado Networks Division**
**3404 East Harmony Road**
**Fort Collins, CO 80525**

ATTN: User Information Development Department

Fold Here

Tape

Please do not staple

Tape

HEWLETT PACKARD

Printed in U.S.A.
50952-90001 E0189

50952-90001