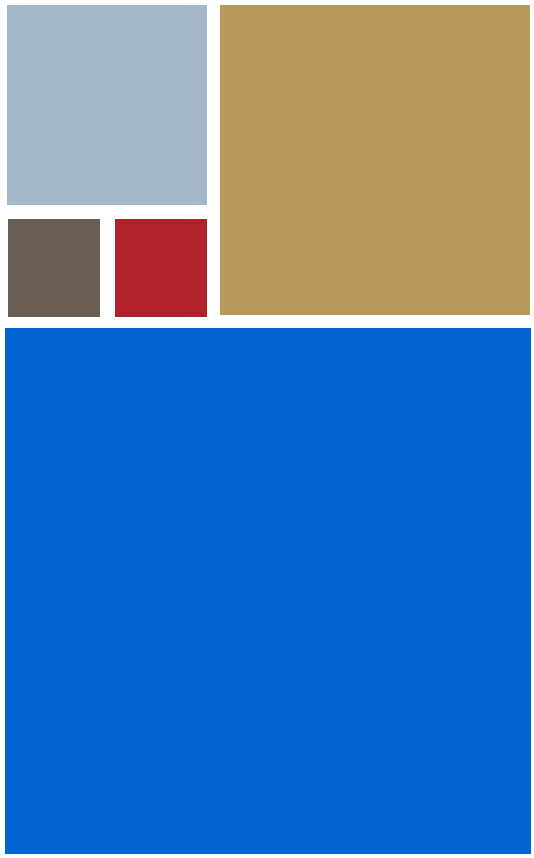


Home

Using LAN Communications

Version 4.7

RadiSys
THE POWER OF WE

www.radisys.com

Revision A • July 2006

Copyright and publication information

This manual reflects version 4.7 of LAN Communications. Reproduction of this document, in part or whole, by any means, electrical, mechanical, magnetic, optical, chemical, manual, or otherwise is prohibited, without written permission from RadiSys Microware Communications Software Division, Inc.

Disclaimer

The information contained herein is believed to be accurate as of the date of publication. However, RadiSys Corporation will not be liable for any damages including indirect or consequential, from use of the OS-9 operating system, Microware-provided software, or reliance on the accuracy of this documentation. The information contained herein is subject to change without notice.

Reproduction notice

The software described in this document is intended to be used on a single computer system. RadiSys Corporation expressly prohibits any reproduction of the software on tape, disk, or any other medium except for backup purposes. Distribution of this software, in part or whole, to any other party or on any other system may constitute copyright infringements and misappropriation of trade secrets and confidential processes which are the property of RadiSys Corporation and/or other parties. Unauthorized distribution of software may cause damages far in excess of the value of the copies involved.

Contents

Chapter 6: BOOTP Server

Chapter 7: Utilities

Chapter 8: Programming

1 Networking Basics

The Local Area Network (LAN) Communications enables your OS-9® system to communicate with other computer systems connected to a TCP/IP network. This enables you to send data, receive data, and log on to other computer systems.

This chapter introduces you to the basics of networking and provides you with a working knowledge of internetworking. The following sections are included:

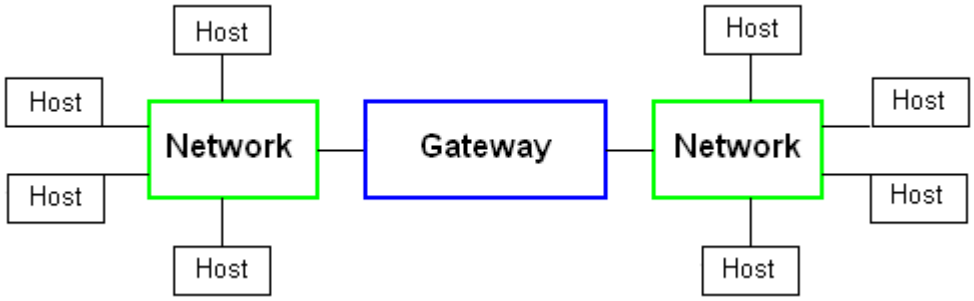
Basic Networking Terminology

A computer network is the hardware and software enabling computers to communicate with each other. Each computer system connected to the network is a "host". There can be different types of host computers, and they can be (and usually are) located at different sites. For example, you may have your OS-9 system connected to a network consisting of other OS-9 systems, as well as UNIX and DOS systems.

An "internet" is the connection of two or more networks; it uses the Internet Protocol, which enables computers on one network to communicate with computers on another network. An internet is sometimes referred to as an "internetwork".

Networks are connected to each other by "gateways". Gateways are computers dedicated to connecting two or more networks. Figure 1-1 illustrates two networks connected by a gateway.

Figure 1-1. Gateway Connected Networks



Datagrams

When information (data) is passed from one host to another, either on the same network or across gateways, the data are carried in packets. Packets are the actual physical data transfered across the network layer. A datagram is a specific type of packet and is the basic unit of information passed on a network. In internetworking, this unit is called an Internet Protocol or IP datagram.

A datagram is divided into a header area and a data area. The datagram header contains the source and the destination Internet Protocol address and a type field identifying the datagram's content.

Figure 1-2 illustrates the basic concept of a datagram.

Figure 1-2. Basic Datagram

Fragmentation

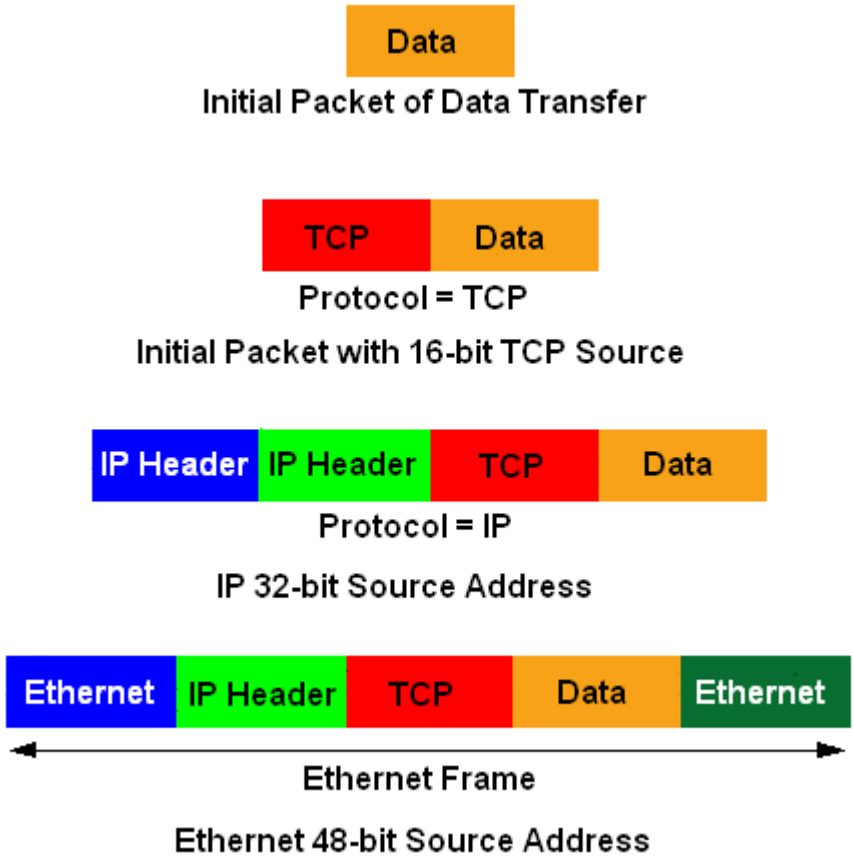
The datagram size depends on the network's Maximum Transfer Unit (MTU). Because each network may have a different MTU, hosts and routers divide large datagrams into smaller fragments when the datagram needs to pass through a network having a smaller MTU. The process of dividing datagrams into fragments is known as "fragmentation".

Fragmentation usually occurs at a gateway somewhere along the path between the datagram source and its final destination. The gateway receives a datagram from a network with a large MTU and must route it over a network where the MTU is smaller than the datagram size. The size of the fragment must always be a multiple of eight and is chosen so each fragment can be shipped across the underlying network in a single frame. A frame is passed across the data link layer and contains an encapsulated datagram. Hosts may also fragment large datagrams into multiple packets according to the size of the local MTU. Fragments are reassembled at the final destination to produce a complete copy of the original datagram before it can be processed by the upper protocol layers.

Encapsulation

The addition of information to the datagram is called "encapsulation". Datagrams are encapsulated with information as they pass through layers of the network. Figure 1-3 illustrates this concept.

Figure 1-3. Encapsulated Datagram Example



Client and Server

The terms client and server appear frequently in networking documentation. A server process provides a specific service accessible over the network. A client process is any process requesting to use a service provided by a server.

Available Network Protocols

When client and server processes communicate, both processes must follow a set of rules and conventions. These rules are known as “protocols”. Protocols allow hosts to communicate with each other.

Below are the protocols you need to be familiar with when using networks:

- Internet Protocol (IP)
- Transmission Control Protocol (TCP)
- User Datagram Protocol (UDP)

Internet Protocol (IP)

IP is the datagram delivery protocol. It is a lower-level protocol located above the network interface drivers and below the higher-level protocols such as the UDP and the TCP. IP provides packet delivery service for higher level protocols such as TCP and UDP.

Due to the IP layer's location, datagrams flow through the IP layer in two directions:

- network up to user processes
- user processes down to the network

The IP layer supports fragmentation and reassembly. If the datagram is larger than the MTU of the network interface, datagrams are fragmented on output. Fragments of received datagrams are dropped from the reassembly queues if the complete datagram is not reconstructed within a short time period.

The IP layer provides for a checksum of the header portion, but not the data portion of the datagram. IP computes the checksum value and sets it when datagrams are sent. The checksum is checked when datagrams are received. If the computed checksum does not match the checksum in the header, the packet is discarded.

The IP layer also provides an addressing scheme. Every computer on an Internet receives one (or more) 32- or 128-bit address. This allows IP datagrams to be carried over any medium.

A checksum is a small, integer value used for detecting errors when data is transmitted from one machine to another.

Transmission Control Protocol (TCP)

TCP is layered on top of the IP layer. It is a standard transport level protocol allowing a process on one machine to send a stream of data to a process on another machine. TCP provides reliable, flow controlled, orderly, two-way transmission of data between connected processes. You can also shut down one direction of flow across a TCP connection, leaving a one-way (simplex) connection.

Software-implementing TCP usually resides in the operating system and uses IP to transmit information across the underlying Internet. TCP assumes the underlying datagram service is unreliable. Therefore, it performs a checksum of all data to help implement reliability. TCP uses IP host level addressing and adds a per-host port address. The endpoints of a TCP connection are identified by the combination of an IP address and a TCP port number. The TCP packets are encapsulated as shown in Figure 1-4.

Figure 1-4. IP Datagrams

User Datagram Protocol (UDP)

The User Datagram Protocol (UDP) is also layered on top of the IP layer. UDP is a simple datagram protocol that allows an application on one machine to send a datagram to an application on another using IP to deliver the datagrams. The important difference between UDP and IP datagrams is that UDP includes a protocol port number; this enables the sender to distinguish among multiple application programs on the remote machine. Like TCP, UDP uses a port number along with an IP address to identify the endpoint of communication.

UDP datagrams are not reliable. They can be lost or discarded in a variety of ways, including failure of the underlying communication mechanism. UDP implements a checksum over the data portion of the packet. If the checksum of a received packet is incorrect, the packet is dropped without sending an error message to the application. Each UDP socket is provided with a queue for receiving packets. This queue has a limited capacity. Datagrams that arrive after the capacity of the queue has been reached are silently discarded.

Network Addressing

The LAN Communications includes support for both Internet Protocol version 4 (IPv4) and Internet Protocol version 6 (IPv6). Where IPv4 addresses have a range of 32 bits, IPv6 addresses are 128 bits. This ability to hold large addresses is becoming increasingly necessary as more devices become connected to the Internet (both fixed and mobile). In addition, IPv6 contains the following features:

- stateless, address auto-configuration
- quality of service (QoS)
- end-to-end IP security (IPSec)
- simultaneous support of IPv4 and IPv6 traffic

Current vs. Legacy LAN Com Stack

Refer to Appendix D, LAN Communications Stack Migration Reference.

IPv4 Addresses

IP addresses are usually represented visually as four decimal numbers, where each decimal digit encodes one byte of the 32-bit IP address. This is referred to as dot notation. IP addresses specified using the dot notation use one of the following forms:

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

Usually, all four parts of an address are specified. In this form, the most significant byte is written first and the least significant byte is written last.

At times, a written address may omit one or more of the four bytes. When this happens, the address is expanded to a normal four-part address by replacing the missing bytes with zeros. The following list demonstrates this process:

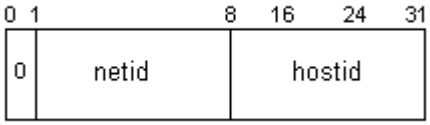
Short Address	Expanded Address
127	0.0.0.127
127.1	127.0.0.1
127.1.2	127.1.0.2

Network Classes

The 32-bit address space is divided into five groups, or network classes. The first three classes consist of the unicast addresses assigned to hosts, the fourth class contains multicast addresses, and the fifth class is reserved for future use. In the first three classes, each 32-bit address is divided into two parts—the network and host portion. These identify the network the host is on and which host it is within that network.

Class A

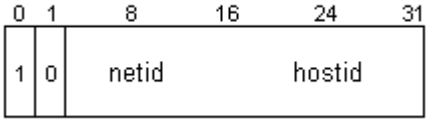
All addresses with a 0 as the first bit in their binary representation are considered Class A addresses.



The first byte represents the `netid` portion of the address, and the remaining 3 bytes represent the `hostid`. Class A addresses range from 0.0.0.0 to 127.255.255.255.

Class B

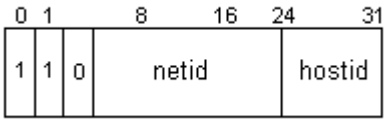
Addresses that start with a binary 10 are in Class B.



These addresses contain 2 bytes for each of the `netid` and `hostid` portions. Class B addresses range from 128.0.0.0 to 191.255.255.255.

Class C

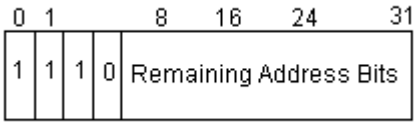
Class C addresses are distinguished by 110 as the first three bits in their binary representation.



The first three bytes of the address form the `netid`, and the remaining byte is used for the `hostid`. Class C addresses range from 192.0.0.0 to 223.255.255.255.

Class D

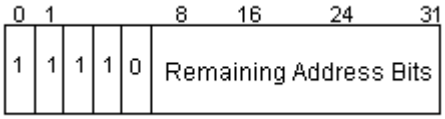
Class D addresses are multicast addresses with the first 4 bits set to 1110.



Class D addresses range from 224.0.0.0 to 239.255.255.255.

Class E

Class E addresses are reserved for future use and have 11110 as the first five bits in their binary representation.



Class E addresses range from 240.0.0.0 to 247.255.255.255

Subnet Masks

The A, B, and C class distinctions are now mostly historical. Instead of using the first few bits of the address to determine the boundary between the `netid` and the `hostid`, a `subnet mask` is used. Wherever the bits in the subnet mask are set to 1, the corresponding bits in the address are part of the `netid`. Wherever the bits are zero, the corresponding address bits are part of the `hostid`. For example:

```
Address: 172.16.193.27
Subnet mask: 255.255.255.0

netid: 172.16.193.0 (or just 172.16.193)
hostid: 0.0.0.27 (or just 27)
```

While it is common to use byte boundaries for subnet masks, it is not required. Another example is:

```
Address: 172.16.193.27
Subnet mask: 255.255.192.0 (18 bits)
```

```
netid: 172.16.192.0
hostid: 0.0.1.27
```

IPv6 Addresses

There are three types of IPv6 addresses:

1. Unicast: an identifier for a single interface

   A packet sent to a unicast address is delivered to the interface identified by that address.

2. Anycast: an identifier for a set of interfaces

   A packet sent to an anycast address is delivered to one of the interfaces identified by that address.

3. Multicast: an identifier for a set of interfaces.

   A packet sent to a multicast address is delivered to all interfaces identified by that address.

There are no broadcast addresses in IPv6. All interfaces are required to have at least one link-local unicast address.

Representation

There are three conventional forms for representing IPv6 addresses as test strings. The preferred form is `x:x:x:x:x:x:x:x`, where `x` is the hexadecimal value of the eight 16-bit pieces of the address.

Example: `FE80:0:0:0:0:FFFF:7F01:2`

In order to make writing addresses containing zero bits easier, use the "`::`" symbol to indicate multiple groups of 16 bits of zeros. The `::` symbol can only appear once in each address. The above example address may be represented as `FE80::FFFF:7F01:2`. The `::` symbol can also be used to compress the leading and trailing zeros in an address. The loopback address is represented as `::1`.

A mixed environment of IPv4 and IPv6 nodes can be represented as `x:x:x:x:x:x:d.d.d.d`, where `x` is the hexadecimal value of each six high-oder 16-bit pieces of the address, and `d` is the decimal value of each of the four low-order 8-bit pieces of the address. The above example address may be represented as `FE80::FFFF:127.1.0.2`.

Prefixes

An IPv6 address prefix is represented by the notation ipv6-address/prefix-length, where "ipv6-address" is an IPv6 address in any of the notations listed above, and "prefix-length" is a decimal value specifying how many of the leftmost contiguous bits of the address comprise the prefix. For example, the 60-bit prefix for the example address is represented as `FE89::FFFF:7F01:2/60`.

2 Serial Line Internet Protocol (SLIP) Driver

The SPF Serial Line Internet Protocol device driver (`spslip`) provides a point-to-point interface between serial connections for transferring IP packets. The following sections are included:

- SPSLIP Introduction
- SPSLIP Transmission Process
- SPSLIP Device Descriptor

SPSLIP Introduction

spslip is based on the RFC 1055 specification for SLIP. In addition, it supports multicasting and the Van Jacobson CSLIP protocol enhancements.

spslip is typically used as an Internet interface to SCF devices (generally an RS-232 serial port) to perform telnet and FTP sessions and other communication functions.

The following table lists the driver and descriptor provided for SLIP:

Table 2-1. spslip Driver and Descriptor

Driver	Descriptor
spslip	spsl0

Installation

To install the spslip device driver on your system, perform the following steps:

Step 1. Setup the inetdb and inetdb2 data files.

You may also use the ifconfig utility to add the slip interface after starting IP.

Refer to *Chapter 7, Utilities* and *Appendix A, Configuring LAN Communications* for more information about setting up the inetdb files.

Step 2. Modify the spf_desc.h file to set baud rate, stop bits, and parity. This file is located in the following directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/SLIP/DEFS
```

For OS-9 systems with only one serial port, be sure to also set the serial device name to be that of your console. For example:

```
#define I_DEV_NAME"/term"
#define O_DEV_NAME"/term"
```

Step 3. Modify the spf_desc.h file to set baud rate, stop bits, and parity. This file is located in:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/SLIP/DEFS
```

Step 4. Remake the descriptor spsl0.

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/SLIP

os9make
```

After running the make, the descriptor binary can be found in the directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/CMDS/BOOTOBJS/SPF
```

The driver spslip can be found in the directory:

```
MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF
```

Step 5. On disk-based systems, initialize the protocol stack manually or with the `startspf` script (see `startspf` example). `startspf` is located in `MWOS/SRC/SYS`. Otherwise, enter the following commands:

```
mbinstall

ipstart
```

For OS-9 systems with only one serial port, be sure to run `inetd` (or `ftpd/telnetd`) after initializing the protocol stack if you want to be able to handle incoming service requests (e.g. using telnet to regain access to a shell).

For example:

```
mbinstall
ipstart; inetd<>>>/nil&
```

Step 6. On disk-based systems, initialize the protocol stack by hand or with the `startspf` script (see `startspf` example). `startspf` is located in `MWOS/SRC/SYS`. Otherwise, enter the following commands:

```
mbinstall

ipstart
```

Step 7. Verify that everything is working correctly by attempting to `ping` to the remote host.

If you have more than one SLIP connection, make sure each connection has a different port address (`PORTADDR` in the driver descriptor configuration section).

- Compression is not negotiated and is ON by default.
- The default serial device used is `/t1`.
- Baud rate is 19200 by default.
- Baud rates, parity, compression, and MTU must match on both ends.

SPSLIP Transmission Process

The `spslip` driver processes the data packets it sends and receives in four ways, including:

- Initialization
- Sending Data
- Reading Data
- Header Compression

Initialization

During initialization, the driver acquires the input and output device names to be opened from the device descriptor. After opening the input and output devices, it sets the options for the input and output paths such as baud rate, echo, and X-ON/X-OFF. `spslip` then creates two processes called `spslip`. These processes control the data flow from the input and output paths. To see the two processes, enter a `procs -e` command after starting up the system. If both processes are not

running for every SLIP port in the system, the SLIP driver did not initialize successfully.

Sending Data

When data is sent over the serial line, `spslip` checks to see if the amount of data being sent is less than the protocol's Maximum Transmission Unit (MTU). The default MTU is 1006, as defined in RFC 1055. To change the MTU, update the `spf_desc.h` file and remake `sps10`.

The sending output process actually performs the SLIP data packaging. Basically, SLIP defines two special characters:

- `END (octal 300)`
- `ESC (octal 333)`

The output process starts sending the data over the serial line. When a data byte is detected to be the `END` character, the process replaces it with the `ESC` and octal 334 for transmission. If the data byte is the same as the `ESC` character, `ESC` and octal 335 are sent instead. When the last character of the packet has been sent, an `END` is sent.

Reading Data

When data is received, `spslip` calls an entry point to push data up to SPF to perform the read. The input process reverses the procedures of the far end SLIP transmitter. When the `END` character is detected, the input process places the packet in the receive queue.

Header Compression

The macro `COMPRESS_FLAG` turns header compression on and off.

To turn compression on (the default), in `spf_desc.h`, use:

```
#define COMPRESS_FLAG(1)
```

To turn compression off, in `spf_desc.h`, use:

```
#define COMPRESS_FLAG(0).
```

SPSLIP Device Descriptor

The `sps10` device descriptor is created by updating the `spf_desc.h` file in the directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/SLIP/DEFS
```

The following is an example of configurable sections of `spf_desc.h`:

```
**Device Descriptor for SPF Slip driver
/************************************************************/
/**** Device Descriptor for the SPF slip driver (spslip)  ****/
/****                                                     ****/
/****  This section contains the configurable parameters  ****/
```

```
/************************************************************/
#define MAXSLIPMTU      SLIPMTU
#define PORTADDR        0
/*
** Name of serial device
*/
#define I_DEV_NAME      "/t1"
#define O_DEV_NAME      "/t1"
/*
** serial data format
*/
#define SL_RCV_BUF_SIZ  4096    /* input raw receive buffer */
#define SL_PAR_BITS     0x00    /* parity/stopbits          */
#define SL_BAUD_RATE    0x10    /* baud rate                */
/*
** These values come from the ISP slip driver and are used to
** set the buffer sizes used by spslip
*/
#define OUTBUFSIZE  SLIPMTU * 2 + 36
#define INBUFSIZE   SLIPMTU + 32
/*
** thread priority
*/
#define SL_IN_PRIOR     128
#define SL_OUT_PRIOR    128
/************************************************************/
```

The previous example makes the `spslip` driver's descriptor `/spsl0` which uses `/t1` as its port.

Table 2-2. Baud Rates

Baud Rate	OS-9 for 68K Value	OS9 Value
9600	0x0e	0x0f
19200 (default)	0x0f	0x10

Refer to the *OS9 for 68K Technical I/O Manual* for 68K processors or the *OS-9 Device Descriptor and Configuration Module Reference* for all other processors.

3 LAN Communications Overview

This chapter is an overview of the Local Area Network (LAN) Communications. The following sections are included:

- Introduction
- LAN Communications Components
- Software Description
- LAN Communications Architecture

Introduction

The LAN Communications provides a small footprint Internet package that enables small embedded devices to communicate in a network environment. The User Datagram Protocol (UDP), Transmission Control Protocol (TCP), Internet Protocol (IP), as well as a raw socket interface are supported in this package.

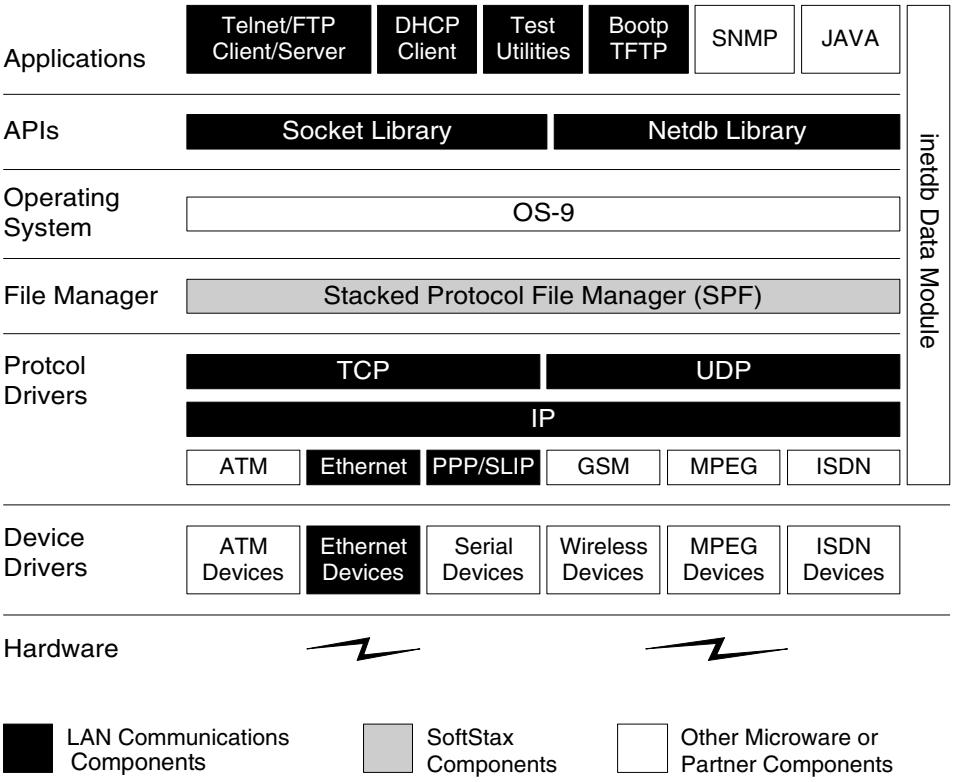
LAN Communications Requirements

LAN Communications uses SoftStax®, the Stacked Protocol File Manager (SPF), for its I/O system. SoftStax is required and provides a complete SPF environment for creating applications and drivers.

LAN Communications Components

LAN Communications provides local area connectivity support for SoftStax™, the Microware integrated communications and control environment for OS-9®. It also adds device-level Ethernet and serial interface capability using Point-to-Point Protocol (PPP) and Compressed Serial Line Internet Protocol (CSLIP).

Figure 3-1. LAN Communications Components



- Telnet provides the user interface for communication between systems connected to the Internet and enables log-on to remote systems.
- File Transfer Protocol (FTP)transfers files to and from remote systems.
- Dynamic Host Configuration Protocol (DHCP) enables a host to retrieve network configuration from a server.

Application Programming Interfaces (APIs)

LAN Communications supports the standard Berkeley Socket Library (`socket.l`) and network/host library `netdb.l`. Socket applications access the `netdb.l` library or `netdb` trap handler for network/host functions and for local or DNS client resolution. `netdb` locates configuration information in the `inetdb` data module or through a DNS client lookup.

File Manager

LAN Communications plugs into the SoftStax™ environment underneath SPF. The tight network/OS integration of SPF enables the speed and efficiency crucial for maximizing throughput while minimizing footprint and CPU use over local and wide area networks.

Protocol Drivers

LAN Communications provides drop-in protocol drivers that can be stacked and unstacked as required by applications to communicate over LANs. These include:

Transmission Control ProtocolProvide reliable data transfer service over IP.

User Datagram ProtocolProvide datagram services over IP.

Internet Protocol Provide Internet packet forwarding.

Ethernet Protocol Provide Ethernet connectivity, ARP requests, and reply layers for Ethernet hardware.

Point-to-Point ProtocolSupport IP over serial links.

Serial Line Internet Protocol (SLIP)Support IP over serial links.

Data Module

LAN Communications stores Internet configuration information in resident data modules with a prefix of `inetdb`. `inetdb` is a database containing Internet configurations for the local machine as well as the hosts, networks, protocols, and services that are available.

`inetdb` functions on a standard file system or can be configured to work in small embedded environments that require Internet connectivity. `inetdb` contains information on machine names, IP addresses and interfaces, and network protocol names and their identification values.

Refer to *Appendix A, Configuring LAN Communications* for information about the files that compose the `inetdb` data modules.

Software Description

Table 3-1 gives a short description of each of the LAN Communications utilities. For more information on a specific utility, click on its hyperlink.

Table 3-1. LAN Communications Examples and Utilities

Examples/Utilities	Purpose
arp	Print and update the ARP table.
beam.c, target.c, beam6.c, target6.c	Example UDP/IP socket program. Source and objects are provided.
bootpd	BOOTP server daemon.
bootpdc	BOOTP connection handler (bootpdc is forked by bootpd).
bootptest	Test the BOOTP server connection.
dhcp	Dynamic Host Configuration Protocol (DHCP) client for setting host networking paramater.
ftp	File Transfer Protocol (FTP) that handles sending and receiving files.
ftpd	FTP server daemon.
ftpdc	FTP server connection handler. (ftpdc is forked by ftpd or inetd.)
hostname	Prints or sets the string returned by the socket library.
idbdump	Dumps the contents of the inetdb data module.
idbgen	inetdb data module generator for host, networks, protocols, services, DNS resolving gethostname() function, interfaces, and routes.
ifconfig	Interface configuration utility.
inetd	Internet Services Master Daemon; inetd can be configured to fork a particular program to handle data for a particular protocol/port number combination. For example, inetd can replace the ftpd and telnetd server daemons.
ipstart	Initializes the IP stack.
ndbmod	Adds, removes, or modifies information stored in the inetdb data module.
netstat	Reports network information and statistics.
ping, ping6	Sends ICMP ECHO_REQUEST packets to host.
pppauth	Utility for configuring PPP authentication.
pppd	Utility to initiate a PPP connection.
route	Add or delete entries from the routing table.
routed,route6d	Dynamic routing daemon.
rtsol	Router solicitation.
tcpsend.c, tcprecv.c,	Example TCP/IP socket program. Source and objects are provided.
telnet	Telnet User Interface; telnet provides the ability to log on to remote systems.
telnetd	Telnet Server Daemon.

Table 3-1. LAN Communications Examples and Utilities (Continued)

Examples/Utilities	Purpose
telnetdc	Telnet Server Connection Handler. (telnetdc forked by telnetd or inetd.)
tftpd	TFTP Server Daemon.
tftpdc	TFTP Server Connection Handler. (tftpdc forked by tftpd)

Chapter 7, Utilities contains more detailed information about utilities and example applications.

Table 3-2. LAN Communications Libraries

Libraries	Purpose
socket.l	Berkeley socket library.
netdb.l	Library for local or remote host name resolution and network/host functions. Uses the netdb trap handler.
netdb_dns.l	DNS client host name resolution and network functions. Does not use the netdb trap handler. The code is inlined in the application.
ndblib.l	Library for inetdb data module manipulation.

Refer to the OS-9 Networking Programming Reference for detailed information concerning libraries.

Table 3-3. LAN Communications Descriptors and Drivers

Drivers & Descriptors	Purpose
netdb_dns	Local and remote host name resolution module. Client DNS (Domain Name Service) support.
spenet, enet	Driver and descriptor for Ethernet layer. Source files are provided for the descriptor.
spip, ip0, ip0_router	Driver and descriptor for IP protocol. Source files are provided for the descriptor.
spipcp, ipcp0 splcp, lcp0 sphdlc, hdlc0	Drivers and descriptors for the Point-to-Point Protocol (PPP). Source files are provided for the descriptors.
spslip, sps10	Driver and descriptor for the Serial Line Internet Protocol device driver (SLIP). The driver and descriptor provide a point-to-point serial interface between serial connections for transmitting TCP/IP packets. Source files are provided for the descriptor.
spudp, udp0	Driver and descriptor for the UDP protocol. Source files are provided for the descriptor.
sptcp, tcp0	Driver and descriptor for the TCP protocol. Source files are provided for the descriptor.

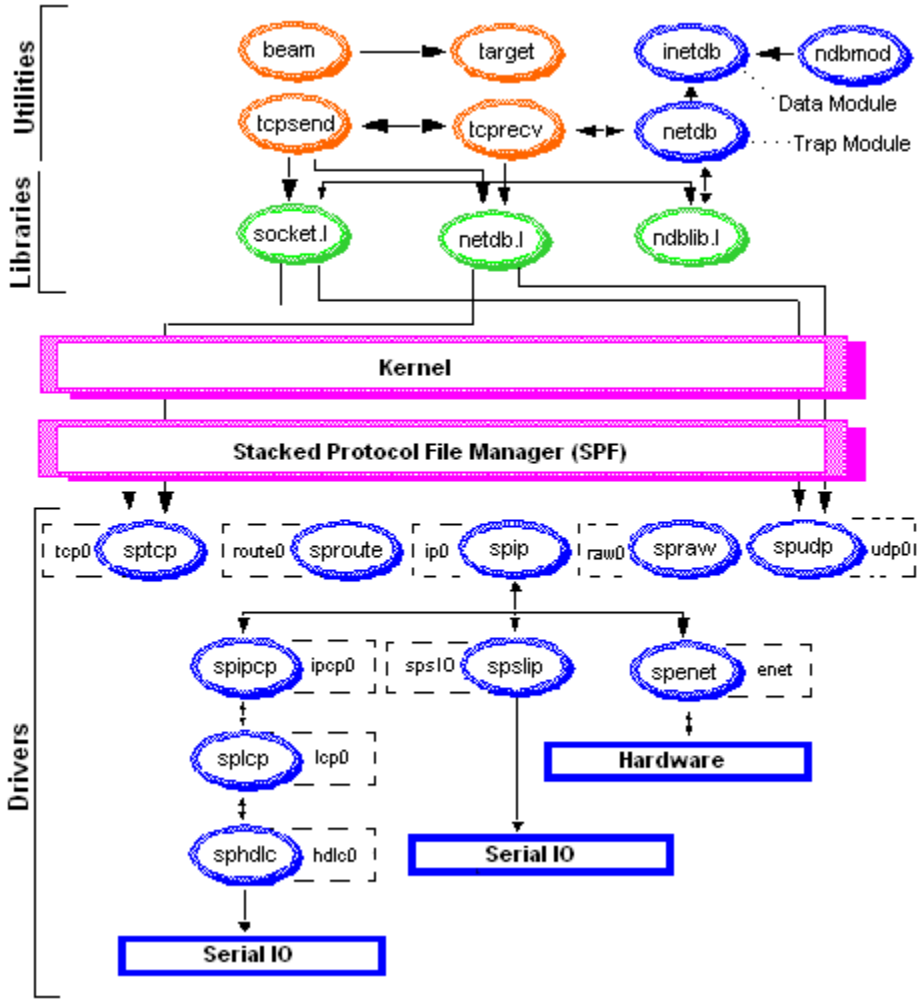
Table 3-3. LAN Communications Descriptors and Drivers (Continued)

Drivers & Descriptors	Purpose
`spraw, raw0`	Driver and descriptor for raw IP support. Source files provided for the descriptor.
`sproute, route0`	Driver and descriptor for IP routing domain support. Source files provided for the descriptor.

LAN Communications Architecture

The following figure shows the architecture and organization of the modules in the LAN Communications. The example applications provided use the socket libraries (`socket.l` and `netdb.l` for network/host functions) to make standard BSD socket calls.

Figure 3-2. LAN Communications Architecture



The `idbgen` or `ndbmod` utilities create the `inetdb` data module containing host/network and other configuration information. The `ndbmod` utility allows

dynamic `inetdb` generation for entries in the `inetdb` data module on the resident system.

See *Appendix A, Configuring LAN Communications* for more information about files that compose the `inetdb` data module.

4 Point-to-Point Protocol (PPP)

This chapter discusses the OS-9 PPP protocol implementation. It includes the following sections:

Introduction to PPP

The PPP device driver provides a point-to-point serial interface between serial connections for transferring TCP/IP packets (as described in *Request for Comment (RFC) 1661* and *1662*). The PPP device drivers provide PPP functionality in the SoftStax® environment.

The following PPP topics are discussed in conjunction with PPP:

- installation
- transmission process
- device descriptors
- utilities

PPP is typically used as an Internet interface to Serial Character File Manager (SCF) devices--generally on an RS-232 serial port--to perform telnet sessions, File Transfer Protocol (FTP) sessions, and debugging capabilities.

PPP Drivers and Descriptors

The following is a list of drivers and descriptors provided for PPP:

Table 4-1. PPP Drivers and Descriptors

Drivers	Descriptors
`sphdlc`	`hdlc0`
`spipcp`	`ipcp0`
`splcp`	`lcp0`
`sppscf`	`pscf<n>` (corresponds to SCF device /t<n>)

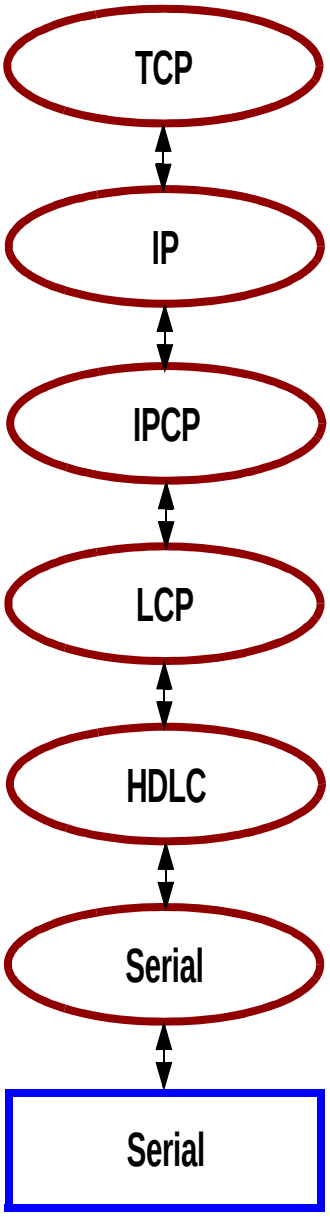
Protocol Drivers

The SPF PPP protocol driver is implemented as a stack of four protocol drivers:

- Internet Protocol Control Protocol (IPCP) driver
- Link Control Protocol (LCP) driver
- link-level High-Level Data Link Control (HDLC) framer
- Softstax serial protocol driver

Figure 4-1 shows how the drivers communicate with each other and the serial device.

Figure 4-1. PPP Data Flow



Utility Programs

Two utility program examples are provided for the PPP driver:

Table 4-2. Utility Programs

Program	Name	Source Code Location
PPP Daemon Utility	`pppd`	`MWOS/SRC/SPF/PPP/UTILS/PPPD`
Authentication setup utility	`pppauth`	`MWOS/SRC/SPF/PPP/UTILS/PPP_AUTH`

Installation

To install PPP on your system, complete the following steps:

Step 1. Set up `inetdb` and `inetdb2`.

You may also use the `ifconfig` utility to add the PPP interface after starting IP.

Refer to *Chapter 7, Utilities* and Appendix A, Configuring LAN Communications for more information about the `inetdb` and `inetdb2` data modules.

Step 2. If necessary, modify the `spf_desc.h` file for the `sppscf` descriptor. SPF descriptor information, such as baud rate, can be set up in this file. `spf_desc.h` is located in the following directory:

```
OS-9:            MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/DEFS
OS-9 for 68k:    MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K/DEFS
```

Step 3. If necessary, modify the `spf_desc.h` file for the IPCP descriptor. `spf_desc.h` is located in the following directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/IPCP/DEFS
```

Step 4. If necessary, modify the `spf_desc.h` file for the LCP descriptor. `spf_desc.h` is located in the following directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/LCP/DEFS
```

Step 5. Remake the `pscf<n>`, `hdlc0`, `lcp0`, `ipcp0`, and `descriptors`. For each descriptor, run `os9make`.

Below are the directories in which the makefiles can be found. (These pathnames correspond with the descriptors listed above.)

```
OS-9:            MWOS/SRC/DPIO/SPF/DRVR/SPPSCF
OS-9 for 68k:    MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/HDLC
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/LCP
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/IPCP
```

Step 6. Load the system modules `inetdb` and `inetdb2`.

Step 7. Start SPF. (For an example, refer to the `startspf` script.) Load the PPP system modules either manually or with the `loadspf` file. `loadspf` is located in `MWOS/SRC/SYS`.

Step 8. Start the system. If you are using a disk-based system, use `startspf` in `MWOS/SRC/SYS`. Otherwise, enter the following commands:

```
mbinstall
ipstart
pppd pcsf<n>&
```

PPP Initialization

During initialization, the following items are set up:

- the `pscf<n>/hdlc0/lcp0/ipcp0` stack

- the IPCP and LCP state tables

sppscf calls the appropriate SCF driver in order to gather information about incoming data and transmit outgoing data to the serial device.

Sending Data

During the sending data process, the following events occur:

1. IP sends data to the IPCP driver, which checks to verify whether or not the packet is IP, and compresses the TCP header, if necessary.
2. The IPCP driver adds a protocol field to the front of the packet and passes it to the LCP driver.
3. The LCP driver passes the unprocessed packet to the HDLC driver.
4. The HDLC driver adds an HDLC frame to the packet and passes it to sppscf.
5. The sppscf puts it on the transmit queue for the serial device to transmit.

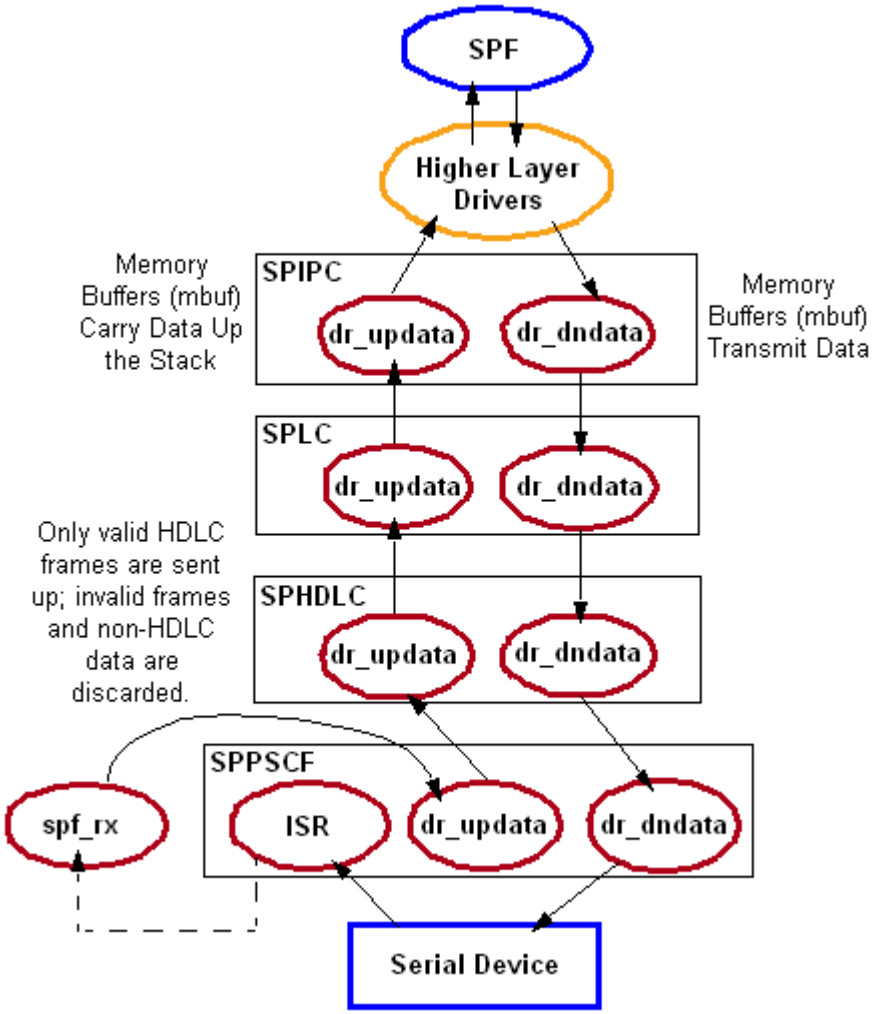
Reading Data

During the reading data process, the following events occur:

1. The sppscf ISR gathers data and wakes the SPF receive thread (spf_rx) in order to take the data up the stack.
2. The HDLC driver processes the data for proper HDLC framing. The valid frames have the HDLC frame removed and the remaining packet is sent up the stack. Invalid frames and non-HDLC data are discarded.
3. The LCP driver examines the packet to check for an LCP message, and handles it accordingly.
4. Other packets are sent to higher level drivers according to the protocol field in the packet. Packets that cannot be delivered for any reason are discarded.

Figure 4-2 details the data flow through the HDLC framer.

Figure 4-2. Data Flow through HDLC Framer



PPP Protocol Functions

The Point-to-Point Protocol (PPP) Application Programming Interface (API) provides four types of function calls: stack configuration, CHAT scripting, authentication database, and connection/disconnect. In addition, this API defines structures that provide error reporting and other functionalities between the PPP stack and the software using the API.

Stack Configuration

Stack configuration consists of deciding which, if any, default options within the stack need to be modified, and then modifying those values. The calls that provide this functionality include those listed below:

`ppp_get_options()`
: Get negotiable stack options.

`ppp_set_options()`
: Set negotiable stack options.

`ppp_option_block()`
: Communicate the current, desired values between the stack and the API.

The API also provides functions to get and set the asynchronous parameters of the PPP link. These parameters include, among others, baud rate, parity, and word size. The calls that provide this functionality are listed below:

`ppp_get_asynch_params()`
: Get asynchronous parameters of the PPP link.

`ppp_set_asynch_params()`
: Set aynchronous parameters of the PPP link.

`ppp_modem_p()`
: Communicate the current, desired values between the stack and the API.

Chat Scripting

CHAT scripting is the process of setting up the data link between the PPP client and server. This may involve sending or receiving commands, logging onto a UNIX shell account and running a PPP startup command, or performing messaging before the client and server exchange PPP configuration packets. The calls that provide this functionality are listed below:

`ppp_chat_open()`
: Open a raw CHAT path.

`ppp_chat_close()`
: Close CHAT path.

`ppp_chat_write()`
: Write data to CHAT path.

`ppp_chat_read()`
: Read data from CHAT path.

`ppp_chat_script()`
: Run a CHAT script.

`ppp_connect()`
: Run optional CHAT script; establish a PPP link.

The `ppp_chat_open()`, `ppp_chat_close()`, `ppp_chat_write()`, and `ppp_chat_read()` functions are low-level functions provided so that you can implement your own version of a CHAT scripting engine. In most cases, the API's built-in engine is adequate; thus, the `ppp_chat_script()` or `ppp_connect()` function is used.

Authentication Database

The authentication database is a memory module referenced by the PPP stack during the authentication phase of a PPP connection. If no authentication is needed, this database may be nonexistent. The currently supported authentication protocols are PAP (Password Authentication Protocol) and CHAP (Challenge-Handshake Authentication Protocol). Using the authentication calls listed below, the administrative program may store the PAP/CHAP information needed to log onto any number of servers.

`ppp_auth_create_mod()`
: Create a new authentication module (database).

`ppp_auth_link_mod()`
: Link to an existing authentication module.

`ppp_auth_unlink_mod()`
: Unlink from an authentication module.

`ppp_auth_get_cur_chap()`
: Get CHAP name/secret for currently set peer.

`ppp_auth_get_cur_pap()`
: Get PAP name/secret for currently set peer.

`ppp_auth_get_peer_name()`
: Get name of currently set peer.

`ppp_auth_set_peer_name()`
: Set current peer.

`ppp_auth_add_chap()`
: Add a new CHAP entry.

`ppp_auth_add_pap()`
: Add a new PAP entry.

`ppp_auth_del_chap()`
: Delete an existing CHAP entry.

`ppp_auth_del_pap()`
: Delete an existing PAP entry.

Connect/Disconnect

Connect/disconnect calls cause the PPP stack to begin the negotiation and establishment of a PPP link, or the termination of an existing PPP link. The calls that provide this functionality are listed below:

`ppp_connect()`
: Run optional CHAT script & establish PPP link.

`ppp_start()`
: Establish a PPP link.

`ppp_disconnect()`
: Terminate current PPP link.

Other miscellaneous functions are provided by this API. (Some of these functions are required and some are optional, as indicated.) The calls are listed below:

`ppp_init()`
: Initialize the PPP API. (required)

`ppp_term()`
: Terminate the PPP API. (required)

`ppp_open()`
: Open the PPP stack. (required)

`ppp_close()`
: Close the PPP stack. (required)

`ppp_get_params()`
: Obtain negotiated stack parameters. (optional)

`ppp_get_statistics()`
: Obtain current stack statistics. (optional)

`ppp_reset_statistics()`
: Reset stack statistics. (optional)

For information on how to use PPP function calls, refer to the *OS-9 Network Programming Reference*.

PPP Device Descriptors

The PPP device descriptors are modified by updating the `spf_desc.h` file in the following directories:

OS-9: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/DEFS
OS-9 for 68k: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/HDLC.API/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/LCP/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/IPCP/DEFS

The PPP device descriptors pscf<n>,hdlc0, lcp0, and ipcp0 contain the device settings for making a PPP connection. Below are the locations in which default settings for each PPP device descriptor can be found. (These pathnames correspond with the descriptors listed above.)

OS-9: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/defs.h
OS-9 for 68k: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K/defs.h

MWOS/SRC/DPIO/SPF/DRVR/SP_PPP/HDLC/defs.h

MWOS/SRC/DPIO/SPF/DRVR/SP_PPP/LCP/defs.h

MWOS/SRC/DPIO/SPF/DRVR/SP_PPP/IPCP/defs.h

The values that may be selected for device descriptor options are defined in the following location:

MWOS/SRC/DEFS/SPF/ppp.h

The file defs.h (in MWOS/SRC/DPIO/SPF/DRVR/SP_PPP/HDLC) contains default settings for the hdlc0 device descriptor that shipped prior to release of Microware OS-9 v3.0; that version of OS-9 has been retired.

Overriding Default Settings

To override the default settings, add a new option definition to spf_desc.h in the following directories: (Do not modify defs.h to change settings.)

OS-9: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/DEFS
OS-9 for 68k: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/HDLC/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/LCP/DEFS

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/IPCP/DEFS

PPP Descriptor Makefiles

The makefile for each PPP descriptor can be found in the following directories:

OS-9: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF
OS-9 for 68k: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/HDLC

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/LCP

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/PPP/IPCP

Rebuilding the Descriptor

Run the following command in each makefile directory to rebuild the descriptor:

```
os9make
```

The rebuilt descriptor modules (hdlc0, lcp0, and ipcp0) can be found in the following directory:

MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/SPF/CMDS/BOOTOBJS/SPF

The serial driver descriptor, pscf<n>, can be found in the following directory:

MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF

Example: Changing the Baud Rate

To change the baud rate for your processor to, for example, 38400, complete the following steps:

Step 1. Because the baud rate is set in the SPPSCF descriptor (refer to MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/defs.h), you will have to open spf_desc.h from the following directory:

OS-9: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF/DEFS
OS-9 for 68k: MWOS/SRC/DPIO/SPF/DRVR/SPPSCF68K/DEFS

Step 2. Once you have opened the spf_desc.h file, scroll to the section that contains the macro definitions for your specific pscf descriptor. For example, if you are using the /t1 interface on your system, find the section containing the macros definitions and code for the pscf1 descriptor.

If you are using an unlisted device descriptor, you will need to create a section of code with the appropriate information for your descriptor. For example, suppose you are using the unlisted interface, /t5. You will need to add the following lines:

```
#ifdef pscf5          /* Macro definitions for the "pscf5" descriptor */
#define SCF_DEVNAME  "/t5" /* SCF driver path to use */
#define LUN         0x05    /* log.unit num: dd_lu_num */
#endif /* pscf5 ************************************************/
```

Step 3. Once at the section discussed in step two, locate the line of code that contains the SCF driver path:

```
#define SCF_DEVNAME   "/t<n>"  /* SCF driver path to use */
```

Underneath the code containing the driver path, enter the following command (This will override the default baud rate set in defs.h.):

```
#define BAUD_RATE BAUDRATE_38400
```

This section of code should now look similar to the following example:

```
#ifdef pscf<n>    /* Macro definitions for the "pscf<n>" descriptor */
#define SCF_DEVNAME   "/t<n>"      /* SCF driver path to use */
#define BAUD_RATE BAUDRATE_38400  /* Override default baude rate */
#define LUN         0x0<n>         /* log.unit num: dd_lu_num */
#endif /* pscf<n>************************************************/
```

The values that may be selected for baud rates are defined in MWOS/SRC/DEFS/SPF/ppp.h.

Step 4. Save the file.

Step 5. Enter cd.. to move up one directory.

Step 6. Run os9make.

Step 7. The rebuilt pscf<n> descriptor resides in the following location:
/MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF

Utilities

The following sections detail the pppd and pppauth utilities provided with PPP.

PPP Daemon Utility

The PPP daemon utility (pppd) opens a connection to the PPP stack, then sleeps indefinitely. This pppd utility can also change some of the device settings in the driver stack.

The source code for pppd is located in the following directory:

```
MWOS/SRC/SPF/PPP/UTILS/PPPD
```

PPP Daemon Command Line Arguments

```
$ pppd -?
```

The command line for the daemon program is shown below:

```
pppd [<options>] <stack_name> [<parameters>]
```

Function: Set up point-to-point connection.

Options:

-c=<name>	Run the chat script located in <name>. <name> may either be a disk file.
-d=<dev>	Use <dev> as the chat device. (The default is /hdlc0.) This requires the -c option.
-i=<index number>	Specify the PPP stack index number.
-k	Terminate the pppd session specified by the -i option.
-p <name>	Select <name> in ppp_auth (used for authentication). This is the equivalent to pppauth -h <name>.
-v	Turn on verbose mode.
-x	Terminate all pppd sessions.
-z	Read commands from stdin.
-z=<name>	Read commands from file or data module.
stack_name	This is the name of stack to open. (If no forward slash (/) is specified, then /hdlc0/lcp0/ipcp0 is automatically appended.)
parameters	These are the PPP stack configuration parameters.

pppd Script Commands

The following commands may be used in a pppd script. The commands may be used in any order. Each command must be on a separate line. Comments may be included in the file using a pound (#) sign and can be placed on a separate line or at

the end of a command line. The commands are not case-sensitive, but device names are used exactly as entered in the script.

`set auth_challenge`
Request LCP to challenge the server for authentication.

This feature is not currently supported.

`set baud[rate] <rate>`
Set baud rate. `<rate>` must be one of the following values: 50, 75, 110, 134.5, 150, 300, 600, 1200, 1800, 2000, 2400, 3600, 4800, 7200, 9600, 19200, 31250, 38400, MIDI.

This feature is not available for OS9 for 68K systems.

`set flow <RTS | XON>`
Set hardware (RTS/CTS) or software (X-On/X-Off) type flow control.

`set ipcp accept-local`
During IPCP negotiation, allow the peer to set the local address.

`set ipcp accept-remote`
During IPCP negotiation, allow the peer to set the remote address.

`set ipcp cslot <value>`
Set flag for compressing slot identification in TCP header compression. (0=do not compress; 1=compress.)

`set ipcp defaultroute`
Set the system IP default route to the address of the system at the other end of the link.

`set ipcp scn <value>`
Perform the maximum number of attempts allowed to send `configure-nak` messages without sending a `configure-ack`, prior to assuming negotiation is impossible.

This feature is not currently supported.

`set ipcp stv <value>`
Perform the maximum number of attempts allowed to send `terminate-request` messages without receiving a `terminate-ack` response. The default value is typically two.

`set ipcp mslot <value>`
Perform maximum slot identification for TCP header compression. The typical values that allow slot identification are between 0 to 15.

`set ipcp timeout <value>`
This is the wait time (in milliseconds) for the IPCP retry timer. The default value is typically 3000 ms (three seconds).

`set mode <option,option...>`

Set mode flags. Choose options from the following: `nowait`, `passive`, `updata`, `modem`, `loopback`, `norxcomp`, `notxcomp`, `nopap`, `nochap`, `nopfc`, `noacfc`. Options are described in Table 4-3.

`set parity <mode>`
Set parity mode. Choose `<mode>` from the following: `None`, `Odd`, `Even`, `Mark`, `Space`.

`set rx accm <value>`
Set RX->receive Async Control Character Map.

`set rx acfc <value>`
Set RX->receive Address/Control Field Compression flag. Set to `1` if Address/Control Field Compression flag peer is desired.

This feature is currently not implemented.

`set rx buffer <value>`
Set RX->receive buffer size.

`set rx device <name>`
Set receive port device. The maximum length of the device name is 16.

`set ipcp proto <value>`
Specify the IP compression protocol to be used. This may be zero for no compression, or `COMPRESSED_TCP` for the Van Jacobsen compression algorithm.

`set rx mru <value>`
Set RX->Receive Max Receive Unit.

`set rx pfc <value>`
Set RX->Receive Protocol Field compression flag. Set to one if Protocol Field compression flag peer is desired.

This feature is currently not supported.

`set scr <value>`
Set and perform maximum number of attempts to send configuration requests (sent by LCP layers) without receiving a valid `configure-ack`, `configure-nak`, or `configure-reject` message. The default value is typically ten.

`set stop <bits>`
Set the number of stop bits. Choose <bits> from `1`, `1.5`, or `2`.

`set str <value>`
Set and perform maximum number of attempts to send `terminate-request` messages without receiving a `terminate-ack` response. The default value is typically `2`.

`set timeout <value>`
Set the number of seconds for time-out value.

This is the wait time (in milliseconds) for the LCP retry timer. The default value is typically 3000 ms (seconds).

`set tx accm <value>`
Set TX->Transmit Async Control Character Map.

`set tx acfc <value>`
Set TX->Transmit Address/Control Field Compression flag.

This feature is currently not supported.

`set tx block[size] <value>`
Set the maximum block size for transmit.

`set tx device <name>`
Set the transmit port device. The maximum length of device name is `16`.

`set tx mru <value>`
Set the TX->Transmit Max Receive Unit.

`set tx pfc <value>`
Set the TX->transmit Protocol Field Compression flag.

This feature is currently not implemented.

`set word[size] <size>`
Set word size. Select `<size>` from the following options: `5`, `6`, `7`, `8`.

Mode Settings

The mode setting controls the behavior of the `HDLC`, `LCP`, and `IPCP` drivers. It is a bitmask and can have multiple values. The different values and their meanings are described in Table 4-3. (The symbols in parentheses indicate descriptor settings defined in `ppp.h`.)

Table 4-3. Mode Settings for HDLC, LCP, and IPCP Drivers

Mode	Description
`modem`	(`WAIT_FOR_MODEM`) HDLC will delay coming up until the chat script executes `port_ready on` and finishes.
`passive`	(`PASSIVE_OPEN`) LCP and IPCP will not initiate the connection, but wait for a configuration request from the peer.
`nowait`	(`NO_WAIT_ON_OPEN`) LCP and IPCP will not wait for a lower layer to enable the I/O path.
`nopap`	(`NO_PAP`) LCP will not use PAP authentication.
`nochap`	(`NO_CHAP`) LCP will not use CHAP authentication.
`nopfc`	(`NO_PFC`) LCP will not use protocol field compression.
`noacfc`	(`NO_ACFC`) LCP will not use address/control field compression.
`norxcomp`	(`NO_RX_COMPRESS`) IPCP will reject compression configuration requests from the peer.
`notxcomp`	(`NO_TX_COMPRESS`) IPCP will not request compression during link negotiation.

Table 4-3. Mode Settings for HDLC, LCP, and IPCP Drivers (Continued)

Mode	Description
updata	(XPARENT_UPDATA_OK) Allows a driver to be stacked below HDLC and causes it to send data up unaltered.
loopback	(LOOPBACK_MODE) Notifies HDLC to loopback characters. This is only useful for testing.

Chat Scripting Commands

A CHAT script is a series of commands that controls the setup of a connection pathway. (It is necessary to set up a connection pathway before PPP can begin negotiating its own parameters.) The CHAT script may control a modem, log a user onto a host computer, and run a PPP-startup shell command on a host computer. However, a CHAT script is not always required, as some links require no pre-PPP setup.

CHAT scripts are very simplistic and are often referred to as "send/expect" exchanges. A CHAT script consists of a series of commands that are executed sequentially (much like a shell script). Commands may exist in any combination of upper/lower case characters; however, some commands require a string parameter. For example, the send command requires a string in order to send out the CHAT path. Strings may or may not be enclosed within quotation marks. In addition, it is possible to embed nonprintable characters within a string. Below is a list of embeddable escape sequences:

\???	??? is the octal value of the byte to be inserted in the string.
\x??	?? is the hexadecimal value of the byte to be inserted into the string.
\c	Do not send carriage return at end of string (for send command, only).
\n	Insert a newline ($0D) into the string.

An example send command is shown below:

```
send "Hello\nGoodbye"
```

The above is the same as the following command:

```
send Hello\x0dGoodbye
```

In addition, comment lines may be inserted into a script by starting the line with either a pound (#) or asterisk (*) symbol. Empty lines are treated as comment lines. If a CHAT script is contained within a data memory module, the end of the script must be terminated by a NULL ('\0') character to denote “end-of-file”. Below is a list of commands supported by the PPP API's CHAT scripting engine.

abort <string>	Initiate abort sequence if the indicated string is received from the CHAT path.
expect <string>	Wait until the indicated string is received from the CHAT path before proceeding.
flush	Clear all input data from CHAT path.

`if_abort <delay>, <string>`

Add the indicated string to the list of modem commands to be sent during the abort sequence. `if_abort` strings are sent in the order by which they were placed in the list. The script engine will wait for delay seconds before sending the assigned `if_abort` string.

`end`

Successfully terminates the CHAT script.

`quiet <ON | OFF>`	Set quiet flag accordingly. When quiet flag is on, characters sent to the `log_path` are translated as asterisks (*). This is useful when sending a password that is embedded in the chat script. Default for quiet is OFF.
`send <string>`	Send the indicated string to the CHAT path.
`show_data <ON | OFF>`	Set the state of the show_data flag accordingly. When this flag is on, characters received from the CHAT path are echoed to the log_path. Default for `show_data` is OFF.
`timeout <seconds>`	Set the value of the expect timeout timer. This may be specified as often as needed, resulting in different timeout periods for various sections on the chat script.
`wait <seconds>`	Pause for the specified time interval before proceeding with the chat script.

Example CHAT script to send to the username "foo" and password "bar":

```
* Define some if_abort strings…
if_abort 2, "+++\c"
if_abort 2, "ATH"
* Set up some options…
show_data ON
timeout 10
* Try to log in…
send "\n\c"
expect "user:"
send "foo"
expect "password:"
send "bar"
expect "successful"
end
```

Applications that use a `ppp_conninfo` structure may force the CHAT script to abort at any time by setting the `PPP_CIFLAG_CHATABORT` bit within the flags field of the `ppp_conninfo` structure. This is typically set within an application's signal handler since the `ppp_chat_script()` and `ppp_connect()` calls are synchronous (blocking) calls.

Troubleshooting Modem Settings for PPP

If your board uses a serial device that does not support hardware flow control (RTS/CTS), it may be necessary to turn off hardware flow control on your modem. One symptom that this may be necessary is the occurrence of data not returning to your target board after a modem-to-modem connection is made.

pppauth

Configure PPP Authentication

Syntax

```
pppauth <option>
```

Options

`-a`
Add mode: Add specified entry. `-c` or `-p` must be specified, along with `<ISP name>`, `<Auth ID>` and `<Secret>`.

`-c`
CHAP specifier: Operate on CHAP entry(ies).

`-d`
Delete mode: Delete specified entry. `-c` or `-p` must be specified along with `<ISP name>`.

`-f <num entries>`
Free entries so `<num entries>` are available.

`-h <ISP name>`
Set current ISP name.

`-i <Auth ID>`
Used in Modify mode to specify new Auth ID.

`-l`
List mode: List specified entries. `-c` or `-p` may be specified.

`-m`
Modify mode: Modify specified entry. `-c` or `-p` must be specified along with `<ISP name>` and parameters to change.

`-n`
New mode: Copy existing entry with new type. `-c` or `-p[ap]` must be specified along with `-t[ype]`.

`-p`
PAP specifier: Operate on PAP entry(ies).

`-s <Secret>`
Used in Modify mode to specify new Secret.

`-t [CHAP | PAP]`
Type specifier: Change type to CHAP or PAP. CHAP or PAP may be abbreviated `C` or `P`.

`-v`
Verbose mode. Show progress information.

Description

The `pppauth` utility creates a data module used by `splcp` during the link authentication process.

Prior to creating the authentication module, complete the following actions:

- Obtain information about the peer or Internet Service Provider (ISP) on the other end of the link.
- Know the authentication method being used (for example CHAP).
- Know the shared secret (or password).
- Know your authentication ID (user id).

Choose a name for this connection, or use the name provided by the ISP. This name (ISP name) is used with `pppauth` to store the authentication method, authentication ID, and the secret. The `-a` option is used to add a new entry to the data module. It must be accompanied by `-c` or `-p` to indicate CHAP or PAP authentication. You can add multiple ISP entries. Prior to making a PPP connection, use the `-h` option to select the ISP name to which you want to connect.

Example pppauth Setting

The ISP `cserve` uses CHAP authentication, and provides you with the user id "acct1304", and a password of "b5kosh". The ISP "webnet" uses PAP authentication, and provides you with the userid "webhead", and a password of "doc-oc". Use the following commands to add this information:

```
pppauth -a -c 'cserve' 'acct1304' 'b5kosh'

pppauth -a -p 'webnet' 'webhead' 'doc-oc'
```

Step 1. Prior to connecting to `cserve`, set it as the current ISP by entering the following command:

```
pppauth -h 'cserve'
```

Step 2. Examine the current settings by using the `-l` option:

```
pppauth -l

Current ISP name is 'cserve'.

Type ISP NameAuthentication ID / Secret

CHAP 'cserve''acct1304' 'b5kosh'

PAP 'webnet''webhead' 'doc-oc'
```

Step 3. Once the authentication module (`ppp_auth`) has been created, it can be saved using the `save` command and loaded as part of the network startup procedure.

Setting Up the Client Machine

After the device descriptors are built, load the drivers and descriptors onto the client (target) machine, if it is different from the machine on which the drivers and descriptors were built. Using FTP or another file transfer mechanism, load the following files onto the machine that is to be used for the PPP client:

Step 1. Load `pppd` and `pppauth` from the directory:

```
MWOS/<OS>/<PROCESSOR>/CMDS
```

Step 2. Load `sppscf`, `pscf<n>`, `sphdlc`, `splcp`, and `spipcp` from the following directory:

```
MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF
```

Step 3. Load `hdlc0`, `lcp0`, and `ipcp0` from the directory:

```
MWOS/<OS>/<PROCESSOR>/PORTS/PROTOCOLS/CMDS/BOOTOBJS/SPF
```

Step 4. Set up the `inetdb` and `inetdb2` files, making sure to include a PPP interface.

Refer to *Chapter 7, Utilities* for more information about the `inetdb` file.

Step 5. If necessary, add default route using the `route` command if the descriptors have not already been configured to do so.

Refer to *Chapter 7, Utilities* for more information about the `route` utility.

Prepare Chat Script

On the client machine, prepare a `CHAT` script based on the CHAT Scripting section in chapter three of the *OS-9 Network Programming Reference* manual included with this CD.

Setup Authentication

If you are using Authentication, create the `ppp_auth` module using the `pppauth` utility, or load the module previously created and saved.

Refer to the pppauth section for more information on `ppp_auth`.

Start PPP Daemon Process

To begin the daemon process, complete the following steps:

Step 1. On the client machine, run the following command in the background:

```
pppd -v pscf<n> &
```

Step 2. The daemon program prints status information.

For example: `Device/stack pscf<n>/hdlc0/lcp0/ipcp0 open, path = n`

`pppd` also can read information from a file:

```
pppd -v -z=setup.pppd pscf<n> &
```

Running PPP Over a Modem Link

To dial the modem and connect to the server, run the following command on the client machine:

```
pppd -v -c=<chat file name> -d=<device name> pscf<n> &
```

(If the device name is not specified, the default device name is `/hdlc0`.)

5 Protocol Drivers

This chapter provides information about the IP, RAW, ROUTE, TCP, UDP, and ethernet protocol drivers. The following sections are included:

SPF IP (spip) Protocol Driver

The SPF IP (spip) protocol driver is an IP protocol implementation used in embedded systems requiring internet routing, gateway, fragmentation, and reassembly capabilities. The IP protocol functionality is based on IPv6-enabled NetBSD code. The spip driver also contains support for ICMP and ICMPv6 control messages, such as redirects, port and destination unreachable, time exceeded, and source quenches. It also responds to the ICMP_ECHO messages used by ping and ping6. The following table lists the driver and descriptor provided for IP:

Table 5-1. IP Driver and Descriptor

Driver	Descriptor
spip	ip0
	ip0_router

Data Reception and Transmission Characteristics

The spip driver receives incoming IP packets from the driver below it, and maps the protocol field in the IP header to the appropriate protocol driver above it. For transmission, spip sends the packet to the appropriate interface below it based on the routing tables it maintains. If non-IP packets, or IP packets without a corresponding protocol driver above spip are received, they are discarded.

The spip driver can send and receive packets from multiple SPF drivers below it. These drivers must support certain IP specific setstats that are described later in this chapter.

Upon reception, the packet is passed to the appropriate driver above spip based on the protocol field in the IP header. Protocol drivers above spip are tightly coupled to the internals of spip. Therefore, generic SPF drivers are not supported directly above spip. Additional protocol support can be done through the raw socket interface provided by spraw. If IP fragments are received they are reassembled back into the original packet before being delivered to higher layer protocols.

For transmission, spip passes the packet to the appropriate driver below it based on the destination IP address and the routing tables it maintains. If the packet is too large for the selected interface it is fragmented.

Default Descriptor Values for spip

ip0 and ip0_router are the only IP descriptor files provided with LAN Communications. The files' module name is ip0; there should only be one ip0 descriptor for the machine. By default, ip0 is used as a host machine and ip0_router is used for a router machine.

The following discussion explains how to configure this descriptor and change it by editing the spf_desc.h file in the SPIP directory:

MWOS/SRC/DPIO/SPF/DRVR/SPIP/DEFS/spf_desc.h

Configuring the ip0 Descriptor

To configure or change the ip0 descriptor, complete the following steps:

Step 1. Edit/update `spf_desc.h` in the `/DEFS` directory.

Step 2. Change to the root `/SPIP` directory and run `os9make`.

This process creates an updated descriptor in the following directory:

`MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF`.

Refer to the *Using SoftStax* manual for more information about the contents and usage of the `spf_desc.h` file.

Considerations for Other Drivers

There are extra operations that `spip` performs at certain times with protocol drivers above and below it.

Drivers Above SPIP

Only those protocol drivers (above `spip`) shipped with LAN Communications are supported. However, you can implement new protocols using the raw socket interface provided by `spraw`.

Drivers Below SPIP

Drivers below `spip` are notified of their current IP addresses through the `SPF_SS_SETADDR` and `SPF_SS_DELADDR` setstats. `spip` can associate multiple addresses with the same interface.

When applications join and leave multicast groups, the appropriate interface is notified with an `IP_SS_IOCTL` setstat. This setstat contains a pointer to an `ifreq` structure which contains the address of the multicast group being joined or left.

When drivers below `spip` are opened (during `ipstart` or a `SPF_SS_ATTIF` setstat) an `SPF_GS_SYMBOLS` getstat is sent down the newly opened stack looking for a `bsd_if_data` symbol. If the driver does not implement this getstat, `netstat` will not print interface statistics for this interface.

When `spip` is transmitting data, it passes some additional information within the mbuf. If the packet should be sent as a link layer broadcast, the `M_BCAST` flag will be set in the mbuf's `m_flags` field. If the packet is not a broadcast packet, the four bytes immediately preceding the mbuf data are zero. In addition, if the hardware information of the host is known, it is stored in front of the four bytes. Otherwise, the IP address of the host to deliver it to is contained in the four bytes immediately preceding the mbuf data. This is a different address than the destination address in the IP header when the next hop is an intermediate gateway.

When `spip` receives data from drivers below through its `dr_updata` entry point the data must be 4-byte aligned. This ensures that the source and destination IP addresses may be accessed as 4-byte integers. In addition, `mbufs` passed up to `spip` must have at least 16 unused bytes between the `mbuf` header and the start of data.

If a driver below spip supports multiple interfaces (such as spenet), it must set lu_pathdesc in the logical unit statics to the path descriptor associated with the interface that received the packet before passing it up to spip. Drivers that do not support multiple interfaces do not have to set lu_pathdesc. However, these will suffer a slight performance loss.

Getstats and Setstats above SPIP

This section provides details about getstats and setstats sent by applications and protcol drivers above spip.

SPF_SS_ATTIF

This setstat is used to dynamically add an interface to spip. It causes SPF to open a path to the specified protocol stack but it does not become usable until an address is added via IP_SS_IOCTL.

Example Usage:

```
#include <netdb.h>
#include <SPF/spf.h>
#include <net/if.h>

int s;
error_code error;
struct n_ifnet ifp;
struct spf_ss_pb pb;

s = socket(AF_INET, SOCK_RAW, 0); /* May also use SOCK_DGRAM or */
                                  /* SOCK_STREAM */
strcpy(ifp.if_name, "enet0");
strcpy(ifp.if_stack_name, "/spde0/enet");
ifp.if_flags = IFF_BROADCAST;   /* Initial interface status flags */
ifp.if_data.ifi_mtu = 0;        /* Use the stacks TXSIZE for an MTU */
ifp.if_data.ifi_metric = 0;     /* Not currently used */
pb.code = SPF_SS_ATTIF;
pb.param = &ifp;
pb.size = sizeof(struct n_ifnet);
pb.updir = SPB_GOINGDWN;
error = _os_setstat(s, SS_SPF, &pb);
```

SPF_SS_DETIF

This setstat is used to dynamically delete an interface to `spip`.

Example Usage:

```
#include <SPF/spf.h>


int s;
error_code error;
struct spf_ss_pb pb;


s = socket(AF_INET, SOCK_RAW, 0); /* May also use SOCK_DGRAM or */
                                  /* SOCK_STREAM */
pb.code = SPF_SS_DETIF;
pb.size = sizeof(name);
pb.param = name;
pb.updir = SPB_GOINGDWN;


error = _os_setstat(s, SS_SPF, &pb);
```

ioctl

This library function call implements the UNIX style interface I/O controls such as add and delete addresses, get and set netmasks, broadcast addresses, destination addresses, flags, and retrieve the interface table.

This function call can be used to add an IP address to an interface. Multiple addresses are supported by calling the `IP_SS_IOCTL` command `SIOCAIFADDR` more than once on the same interface. If the interface does not support more than one address or limits the number of addresses, the driver should return an error on the setstat.

Example Usage: Add an IPv4 address to an interface.

```
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <net/if.h>
#include <netinet/in.h>
#include <netinet/in_var.h>

int s;
error_code error;
struct in_aliasreq ifr;

s = socket(AF_INET, SOCK_RAW, 0); /* May also use SOCK_DGRAM or */
                                  /* SOCK_STREAM */
memset(&ifr, 0, sizeof(struct in_aliasreq));
strcpy(ifr.ifra_name, "enet0");
ifr.ifra_addr.sin_len = sizeof(struct sockaddr_in);
ifr.ifra_addr.sin_family = AF_INET;
ifr.ifra_addr.sin_addr.s_addr = htonl(0xac1002e2);
ifr.ifra_broadaddr.sin_len = sizeof(struct sockaddr_in);
ifr.ifra_broadaddr.sin_family = AF_INET;
ifr.ifra_broadaddr.sin_addr.s_addr = htonl(0xac10ffff);
ifr.ifra_mask.sin_len = sizeof(struct sockaddr_in);
ifr.ifra_mask.sin_family = AF_INET;
ifr.ifra_mask.sin_addr.s_addr = htonl(0xffff0000);
error = ioctl(s, SIOCAIFADDR, (caddr_t)&ifr);
```

Other Supported ioctl Commands

Most of the other supported `ioctl` commands are used similarly although the argument passed is a pointer to an `ifreq` or `in6_ifreq` structure rather than an `in_aliasreq` `in6_aliasreq` structure.

The following commands require the parameter described above.

`SIOCGIFCONF`	Get list of all interfaces.
`SIOCGIFADDR`	Get address of interface.
`SIOCGIFNETMASK`	Get netmask of interface.
`SIOCGIFNETMASK_IN6`	Get netmask of interface.
`SIOCGIFDSTADDR`	Get point-to-point address.

SIOCGIFBRDADDR	Get broadcast address of interface.
SIOCGIFFLAGS	Get interface flags.
SIOCSIFADDR	Set address of interface.
SIOCSIFADDR_IN6	Set address of interface.
SIOCSIFNETMASK	Set netmask of interface.
SIOCSIFDSTADDR	Set point-to-pint address.
SIOCSIFBRDADDR	Set broadcast address of interface.
SIOCSIFFLAGS	Get interface flags.

The following example deletes an IPv4 address from an interface:

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

int s;
error_code error;
struct ifreq ifr;
s = socket(AF_INET, SOCK_RAW, 0); /* May also use SOCK_DGRAM or */
                                  /* SOCK_STREAM */
memset(&ifr, 0, sizeof(struct ifreq));
strcpy(ifr.ifr_name, "enet0");
ifr.ifr_addr.sa_len = sizeof(struct sockaddr_in);
ifr.ifr_addr.sa_family = AF_INET;
((struct sockaddr_in *)&ifr.ifr_addr)->sin_addr.
s_addr = htonl(0xac1002e2);
error = ioctl(s, SIOCDIFADDR, (caddr_t)&ifr);
```

The interface table may also be retrieved using the `ioctl` call.

Example Usage:

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

int s;
error_code error;
struct ifconf ifc;
char buffer[256];
s = socket(AF_INET, SOCK_RAW, 0); /* May also use SOCK_DGRAM or */
                                  /* SOCK_STREAM */
ifc.ifc_len = 256;
ifc.ifc_buf = buffer;
error = ioctl(s, SIOCGIFCONF, (caddr_t)&ifc);
```

Getstats and Setstats Below SPIP

This section provides details about the getstats and setstats sent by `spip` to drivers below it.

SPF_SS_SETADDR

Drivers that need to know their IP address (such as `spenet` for ARP processing) should implement the `SPF_SS_SETADDR` setstat. If the number of addresses is limited, an `EOS_FULL` error should be returned when the limit is reached. A driver that does not need to know the protocol addresses may return `EOS_UNKSVC`.

Example Usage:

```
case SPF_SS_SETADDR: {
 struct sockaddr_in *sin = (struct sockaddr_in *)pb->param;
 error = add_ip_address(sin->sin_addr);
 return (error);
}
```

SPF_SS_DELADDR

If the SPF_SS_SETADDR setstat is supported to add addresses, the SPF_SS_DELADDR setstat must be supported to remove them. If an attempt is made to remove an unknown address, EADDRNOTAVAIL should be returned.

Example Usage:

```
case SPF_SS_DELADDR: {
 struct sockaddr_in *sin = (struct sockaddr_in *)pb >param;
 error = del_ip_address(sin->sin_addr);
 return (error);
}
```

IP_SS_IOCTL

The IP_SS_IOCTL setstat is used to notify interfaces to join or leave a particular multicast group. If the interface wishes to limit the number of multicast groups joined, a EOS_FULL error should be returned when the threshold has been exeeded.

Example Usage:

```
#include <sys/ioctl>
#include <net/if.h>
case IP_SS_IOCTL: {
    struct bsd_ioctl *arg_ptr;
    struct ifreq *ifr;

    arg_ptr = pb->param;
    ifr = arg_ptr->arg;
    switch(arg_ptr->cmd) {
        case SIOCADDMULTI:
            return (AddMulticast(ifr->ifr_addr));
        case SIOCDELMULTI:
            return (DeleteMulticast(ifr->ifr_addr));
    }
    return(EOPNOTSUPP);
}
```

SPF_GS_SYMBOLS

The SPF_GS_SYMBOLS getstat is used to retrieve the address of one or more symbols (variables) maintained by a driver. After obtaining this information, a program can examine the variable dynamically. This getstat assists porting programs that use the kvm_nlist or nlist functions on other systems, and relieves drivers of copying statistic information from driver space to user space.

When processing the SPF_GS_SYMBOLS getstat, a driver receives a pointer to an array of nlist structures. The nlist structure is defined in MWOS/SRC/DEFS/SPF/BSD/nlist.h as:

```
struct nlist {
   char         *n_name;  /* symbol name */
   unsigned long n_value; /* address/value of the symbol */
   unsigned char n_type;  /* type defines */
   unsigned char res[3];  /* reserved space */
};
```

You must initialize each n_name member to point to the name of the symbol to be retrieved, and zero the rest of the structure. On a successful return, the n_type member of each element found is non-zero (typically N_ABS). n_value contains the address of the symbol corresponding to the name you specified in n_name. The driver processes every member of the nlist array until it reaches an element where the n_name member is zero. After processing the getstat, the driver passes this getstat to the next driver in the path. This enables one system call to retrieve multiple symbols from multiple drivers.

The following code fragment shows how to use the SPF_GS_SYMBOLS getstat. It retrieves information for the two symbols _ipstat and _rtstat from spip.

```
#include <nlist.h>
#include <spf.h>
struct nlist  nl[5];     /* name list to be passed to IP */
struct nlist *nl_ptr;    /* pointer to walk through list after*/
                         /* getstat */
spf_ss_pb     pb;        /* SPF parameter block */
path_id       path;

memset(nl, sizeof(nl), 0); /* zero list */
nl[0].n_name = "_ipstat";     /* first element to get IP statistics */
nl[1].n_name = "_rtstat";     /* second element to get routing stats */
pb.code = SPF_GS_SYMBOLS;
pb.size = sizeof(nl);         /* or # elements * sizeof(element) */
pb.param = nl;
pb.updir = SPB_GOINGDWN;
if (_os_open("/ip0", FAM_READ, &path) == 0) {
       _os_getstat(path, SS_SPF, &pb);
       _os_close(path);
}
for (nl_ptr = nl; nl_ptr->n_name; nl_ptr++)
       if (nl_ptr->n_type) /* non-zero means symbol was found */
       printf("Address of %s is %X\n", nl_ptr->n_name, nl_ptr->n_value);
```

To access the memory pointed to by n_value, a process must be system-state. User state processes must use _os_permit.

Possible return codes:

EOS_BPADDR	The nlist pointed to by pb.param failed _os_chkmem.

SPF RAW (spraw) Protocol Driver

The SPF RAW protocol driver (spraw) provides a standard raw socket interface to the IP layer.

The following table lists the driver and descriptor provided for RAW.

Table 5-2. SPRAW Driver and Descriptor

Driver	Descriptor
spraw	raw0

Data Reception and Transmission Characteristics

The spraw driver handles input of all IP datagrams where the protocol field of the IP header is 1 (ICMP), 255 (RAW), or any other value that does not have a corresponding driver above spip. On reception of such a datagram, it is delivered to any application that has requested to receive that protocol number. If multiple processes have requested a particular protocol, the incoming mbuf is duplicated and a copy delivered to each path.

On transmission, spraw fills in the required IP header fields and passes the datagram to spip for delivery to the destination. If the IP_HDRINCL option is set for a path, the application has filled in the IP header and the datagram is passed to spip for delivery.

Default Descriptor Values for spraw

raw0 is the only spraw descriptor provided with LAN Communications. There should only be one raw0 descriptor for the machine. The following discussion explains how this descriptor is configured, and how you can change this descriptor by editing the spf_desc.h file in the SPRAW directory:

MWOS/SRC/DPIO/SPF/DRVR/SPRAW/DEFS

Configuring the raw0 Descriptor

To configure or change the raw0 descriptor, complete the following steps:

Step 1. Edit/update spf_desc.h in the /DEFS directory.

Step 2. Change to the root /SPRAW directory and run os9make.

This process creates an updated descriptor in the following directory:

MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF.

Refer to the *Using SoftStax* manual for more information about the contents and usage of the spf_desc.h file.

Consideration for Other Drivers

The spraw driver depends on functions located in spip and will only work on top of spip.

SPF Routing Domain (sproute) Protocol Driver

The sproute driver provides a BSD 4.4 style routing domain. This domain allows a process to send and receive routing messages with spip using the normal sockets API. A routing domain socket can be created by issuing the socket system call and specifying a family of AF_ROUTE and a socket type of SOCK_RAW.

The following table lists the driver and descriptor provided for sproute.

Table 5-3. SPROUTE Driver and Descriptor

Driver	Descriptor
sproute	route0

Data Reception and Transmission Characteristics

The routing domain enables an application to send a datagram containing an rt_msghdr structure to add, delete, or change routes within the system routing table.

The size of the routing table supported by sproute is limited by the amount of memory available. Each routing table entry is approximately 128 bytes.

An application reading from a routing domain socket receives datagrams containing rt_msghdr structures, indicating changes in the system routing table. It receives datagrams containing if_msghdr structures when interfaces go up and down, as well as ifa_msghdr structures when addresses are added to and deleted from the system.

Routing sockets do not require a connect or a bind. After creation they may be immediately written to and read from using basic socket API calls.

Default Descriptor Values for sproute

route0 is the only routing domain descriptor provided with LAN Communications. There should only be one route0 descriptor for the machine. The following discussion explains how this descriptor is configured, and how you can change this descriptor by editing the spf_desc.h file in the SPROUTE directory:

MWOS/SRC/DPIO/SPF/DRVR/SPROUTE/DEFS

Configuring the route0 Descriptor

To configure or change the route0 descriptor, complete the following steps:

Step 1. Edit/update spf_desc.h in the /DEFS directory.

Step 2. Change to the root /SPROUTE directory and run os9make.

This process creates an updated descriptor in the following directory:

MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF.

Consideration for Other Drivers

The sproute driver depends on and only works from functions in spip.

SPF TCP (sptcp) Protocol Driver

The SPF TCP (sptcp) protocol driver provides reliable data transfer service over IP.

The following table lists the driver and descriptor provided for TCP:

Table 5-4. TCP Driver and Descriptor

Driver	Descriptor
sptcp	tcp0

Data Reception and Transmission Characteristics

The sptcp driver receives incoming TCP packets from the spip driver and maps the TCP port and IP destination address to a particular path with matching socket address. If no matching path is found, a TCP reset is returned to the sender.

Upon transmission, sptcp repackages the data to the correct size for the transmitting interface, fills in the necessary header information, and passes the packet to spip for delivery to the destination.

Default Descriptor Values for sptcp

tcp0 is the TCP descriptor provided with LAN Communications. There should only be one descriptor for the machine. The following discussion explains how this descriptor is configured and how you can change the descriptor by editing the spf_desc.h file located in:

MWOS/SRC/DPIO/SPF/DRVR/SPTCP/DEFS

Configuring the tcp0 Descriptor

To configure or change the tcp0 descriptor, complete the following steps:

Step 1. Edit/update spf_desc.h in the /DEFS directory.

Step 2. Change to the root /SPTCP directory and run os9make.

This process creates an updated descriptor in the following directory: MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF.

Considerations for Other Drivers

The sptcp driver depends on and works on top of functions in spip.

SPF UDP (spudp) Protocol Driver

The SPF UDP protocol driver (spudp) provides datagram service over IP. The following table lists the driver and descriptor provided for UDP:

Table 5-5. UDP Driver and Descriptor

Driver	Descriptor
spudp	udp0

Data Reception And Transmission Characteristics

The spudp driver receives incoming UDP packets from the spip driver below and maps the UDP port and IP address to a particular path with matching socket address. The spudp driver creates an address mbuf with each incoming packet and chains the data to it using the m_pnext field of the mbuf header. If multiple paths match, as can happen with wildcard addresses or multicasts, the incoming mbuf is duplicated and a copy sent to each path. If no match is found, an ICMP port unreachable error is returned. On transmission, spudp fills in the necessary header information and passes the datagram to spip for delivery to the destination.

The amount of memory used by a single UDP connection is limited to avoid using all the available mbuf pool. When an application's read queue grows beyond the READSZ specified in the udp0 descriptor, an SPF_SS_FLOWON setstat is generated. When spudp receives this, it buffers up to the number of bytes specified in the RECVBUFFER descriptor variable. If more data is received, the packets are silently dropped. On transmit, spudp does not buffer the data and the mbuf pool may be exhausted if the bottom layer hardware drivers queue an unlimited number of packets.

Default Descriptor Values for spudp

udp0 is the only UDP descriptor provided with LAN Communications. There should only be one udp0 descriptor for the machine. The following discussion explains how this descriptor is configured, and how you can change this descriptor by editing the spf_desc.h file in the SPUDP directory:

MWOS/SRC/DPIO/SPF/DRVR/SPUDP/DEFS

Configuring the udp0 Descriptor

To configure or change the udp0 descriptor, complete the following steps:

Step 1. Edit/update spf_desc.h in the /DEFS directory.

Step 2. Change to the root /SPUDP directory and run os9make.

This process creates an updated descriptor in the following directory:

MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF.

Refer to the *Using SoftStax* manual for more information about the contents and usage of the spf_desc.h file.

Considerations for Other Drivers

The spudp driver depends on functions located in spip and, therefore, only works on top of spip.

SPF Ethernet (spenet) Protocol Driver

The spenet protocol driver sits between the hardware Ethernet drivers and spip. Its main function is to map between Ethernet addresses and IP addresses using the Address Resolution Protocol (ARP). Refer to RFC 826 for additional details. The spenet driver also adds and removes Ethernet headers for outgoing and incoming packets.

Spenet maintains an ARP table for mapping between Ethernet and IP addresses. When spenet receives a unicast packet from spip, and the destination address has no entry in the ARP table, spenet broadcasts an ARP request to discover the Ethernet address. The results are then added to the ARP table, and are removed after 20 minutes. The arp utility can be used to view or modify the ARP table.

Table 5-6. SPF Ethernet Driver and Descriptor

Driver	Descriptor
spenet	enet

Data Reception and Transmission Characteristics

The spenet driver receives incoming Ethernet packets from one or more drivers below it and maps the destination Ethernet address to the destination IP address. Before passing the packet to spip, the lu_pathdesc component of the logical unit statics is set to point to the appropriate path descriptor to enable spip to determine which interface received the packet.

For transmission, spenet adds the appropriate Ethernet hardware destination and source addresses to the packet. If the M_BCAST flag is set in the mbuf header, the packet is assumed to be a broadcast and the all 1's hardware broadcast address is used. If the M_MCAST flag is not set, the existance of the Ethernet hardware destination address is checked. If it does exist, that address should be used to create the Ethernet header. In any other case, the destination address in the IP header is converted to the appropriate link layer multicast address. In all other cases the ARP cache is searched using the IP address of the next hop destination. If no ARP entry exists, the ARP protocol is initiated. When the Ethernet header has been completed, the packet is sent to an interface driver below.

Default Descriptor Values for spenet

enet is the only descriptor for spenet provided with LAN Communications. This descriptor is generic for all Ethernet drivers and should not need to be updated.

This descriptor is configured to change fields by editing the spf_desc.h file in the SPENET directory:

MWOS/SRC/DPIO/SPF/DRVR/SPENET/DEFS

Configuring the enet Descriptor

To configure or change the enet descriptor, complete the following steps:

Step 1. Edit/update spf_desc.h in the /DEFS directory.

Step 2. Change to the root /SPENET directory and run os9make.

This process creates an updated descriptor in the following directory:

MWOS/<OS>/<PROCESSOR>/CMDS/BOOTOBJS/SPF.

Refer to the *Using SoftStax* manual for more information about the contents and usage of the spf_desc.h file.

Other Default Settings

The following variables can be configured in the enet descriptor. All others should not be changed.

MAXADDR_PER_IFACE	indicates the maximum number of protocol addresses that can be associated per hardware interface. LAN Communications supports more than one protocol address (IP) per hardware interface. The default value is four.
TIMER_INT	spenet runs a cyclic timer that is used to remove old arp entries. This value defines the timer interval in seconds. The default value is 60.
KILL_C	If a completed (received arp reply) entry is not used in this many timer intervals it is deleted. The default value is 20.
KILL_I	If an entry remains incomplete (no arp reply received) for this many timer intervals it is deleted. The default value is three.

Drivers Below spenet

When spip receives data from drivers below through its dr_updata entry point the data must be 4-byte aligned. This requirement ensures that the source and destination IP addresses may be accessed as 4-byte integers. When receiving data, the four bytes immediately preceding the mbuf data must contain a pointer to the device entry of the driver sending data up the stack. In addition, spip requires a 32-byte offset. If there is not enough space, packets will be pitched.

Getstats for SPENET

The structures used for getstats and setstats for spenet are found in the following locations:

MWOS/SRC/DEFS/SPF/BSD/net/if_arp.h

MWOS/SRC/DEFS/SPF/BSD/netinet/if_ether.h

The spenet driver provides the following getstats to programmers.

Refer to the *OS-9 Network Programming Reference Manual* for additional details on _os_getstat and the spf_ss_pb structure.

SPF_GS_ARPENT

This setstat retrieves a particular entry from the ARP table. The `param` member of the `spf_ss_pb` structure must point to a user allocated `arptab` structure. The `at_iaddr` member of the `arptab` structure must be set to the IP address (in network order) of the entry to retrieve. On success, `0` is returned. `EOS_PNNF` is returned if the entry cannot be found.

SPF_GS_ARPTBL

This setstat retrieves the entire ARP table. The `param` member of the `spf_ss_pb` structure points to a user allocated array of `arptab` structures. The size member must be set to the size of this array in bytes. On success, `0` is returned, indicating `spenet` copied as much of the table as possible to the users array, and set the size member of the `spf_ss_pb` to the actual size of the ARP table (in bytes).

It is recommended that you retrieve the ARP table using two getstats. The first getstat sets the size to zero. On return, size will indicate the size of the current ARP table. Dynamically allocate this much space (plus some additional space in case the table grows), and issue another getstat.

ENET_GS_STATS

This setstat retrieves the statistics maintained by `spenet`. The `param` member of the `spf_ss_pb` structure points to a user-allocated `enet_stat_pb` structure. On success, `0` is returned.

Setstats for SPENET

The `spenet` driver provides the following setstats to programmers.

Refer to the *OS-9 Network Programming Reference Manual* for additional details on `_os_getstat` and the `spf_ss_pb` structure.

To alter the ARP table, a process must have super user access.

SPF_SS_ADDARP

This setstat adds an entry to the ARP table. The `param` member of the `spf_ss_pb` structure points to a user allocated `arpreq` structure. You must initialize the `arp_pa` (in network order), `arp_ha`, and `arp_flags` members of this structure. See the file `if_arp.h` for settings of `arp_flags`. The `arp_pa` member should be treated as a `sockaddr_in` stucture, setting the `sin_family` to `AF_INET`. On success, `0` is returned.

SPF_SS_DELARP

This setstat removes an entry from the ARP table. The `param` member of the `spf_ss_pb` structure points to a user allocated `arpreq` structure. You must set the `arp_pa` member to the IP address (in network order) of the entry to be deleted. This member should also be treated as a `sockaddr_in` stucture, setting the `sin_family` to `AF_INET`. On success, 0 is returned. `EOS_PNNF` indicates the address was not found in the ARP table.

Refer to the `arp` command in *Chapter 7, Utilities* for additional details.

6 BOOTP Server

The Bootstrap Protocol (BOOTP) enables booting from the network. BOOTP clients require a BOOTP server on the connected network to support the BOOTP protocol as specified in Request for Comment (RFC) 951 (Croft/Gilmore) and Trivial File Transfer Protocol (TFTP) as specified in RFC 906 (Finlayson). It also requires the server to support the BOOTP Vendor Information Extensions are described in RFC 1048 and Request for Comment 1084 (Reynolds).

This chapter covers the following topics:

The BOOTP server is based on the Carnegie Mellon University implementation. Microware does not provide or support the BOOTP server for UNIX or other operating systems. Contact the University Computer Center at Carnegie Mellon for the availability of the BOOTP server on other operating systems.

Bootstrap Protocol

Bootstrap Protocol (BOOTP) is a client-server protocol. The system being booted is the client. The process includes the following steps.

1. The client system makes requests to a server system on the network or the same VME chassis over the backplane. The server or the client may or may not be an OS-9 system. OS-9 clients request the server to identify the following:
   - client IP address
   - server IP address
   - path to the bootfile
   - size of the bootfile

You can adjust the number of contact attempts the client makes to a server by editing the `config.des` file in the following directory:

`mwos\OS9000\<PROCESSOR>\PORTS\<TARGET>\ROM\CNFGDATA`

You must add the line `maxbootptry=<number>` to the `eb` section of the file. `<number>` can be from 1 to infinity. The default is eight.

2. The server subsequently transfers the bootfile across the network back to the client using the TFTP protocol.
3. The ROM boot code starts the network boot option (BOOTP) either through the menu selection or automatically without operator intervention. The client broadcasts the BOOTP request containing the client's hardware address (for example, Ethernet address) retrieved from SRAM. A server responds with the information listed above.
4. The client then sends a TFTP request for its bootfile to the server. The responding server calls the TFTP service to transfer the bootfile to the client. The client reads the bootfile as it is transferred across the network and copies it into local RAM in the same manner as other boot device drivers.
5. After the file is successfully read in by the client, control returns to the booting subsystem to complete the bootstrap and pass control to the OS-9 kernel.

Server Utilities

A BOOTP server includes the `bootptab` configuration file and the utility programs identified in Figure 6-1.

Table 6-1. BOOTP Server Utilities

Name	Description
`bootpd`	Responds to BOOTP client requests with BOOTP server responses.
`bootptest`	A simple utility to test `bootpd` server response.
`tftpd`	Responds to `tftp` read requests and forks `tftpdc` to handle the transfer.
`tftpdc`	Reads a bootfile for a client using the TFTP protocol.

Utility programs are located in the /MWOS/CMDS directory. The bootptab configuration file is usually located in the TFTPBOOT directory copied from MWOS/SRC/TFTBOOT. Below is the procedure for starting a BOOTP server and the associated utilities:

```
tftpd <>>>/nil &
bootpd /h0/TFTPBOOT/bootptab<>>>/nil &
```

The boot file name is dependent on the client BOOTP system. On OS-9, the boot file is called OS9boot.<hostname>. The OS9boot.<hostname> file should have public read permissions set to allow tftpd to access it. While in the /h0/TFTPBOOT directory, use the following command to turn on the public read permissions for all OS9boot files:

```
$ attr -pr os9boot.*
```

The bootpd server is derived from the Version 2.1 bootpd source code. This source code contains the following notice:

```
/*
 * Copyright (c) 1988 by Carnegie Mellon.
 *
 * Permission to use, copy, modify, and distribute this
 * program for any purpose and without fee is hereby
 * granted, provided that this copyright and permission
 * notice appear on all copies and supporting
 * documentation, the name of Carnegie Mellon not be used
 * in advertising or publicity pertaining to distribution
 * of the program without specific prior permission, and
 * notice be given in supporting documentation that
 * copying and distribution is by permission of Carnegie
 * Mellon and Stanford University. Carnegie Mellon makes
 * no representations about the suitability of this
 * software for any purpose. It is provided "as is"without
 * express or implied warranty.

 * Copyright (c) 1986, 1987 Regents of the University of
 * California.
 * All rights reserved.
 *
 * Redistribution and use in source and binary forms are
 * permitted provided that this notice is preserved and
 * that due credit is given to the University of California
 * at Berkeley. The name of the University may not be used
 * to endorse or promote products derived from this
 * software without specific prior written permission.
 * This software is provided as is'' without express or
 * implied warranty.
*/
```

bootptab Configuration File Setup

When bootpd is first started, it performs the following functions:

1. Read a configuration file to build an internal database of clients and desired boot responses for each.
2. Listen for BOOTP boot requests on UDP socket port 67 (bootps).
3. Check the file time stamp on the configuration file before processing a boot request. If the file time stamp changed since the last check, the client database is rebuilt.

The configuration file has a format similar to termcap in which two-character, case-sensitive tag symbols represent host parameters. These parameter declarations are separated by colons (:). The general format for the bootptab file is as follows:

```
hostname:tg=value...:tg=value...:tg=value:
```

hostname is the actual name of a BOOTP client and tg is a two-character tag symbol. Most tags must be followed by an equal sign and a value. Some tags may also appear in Boolean form with no value (:tg:).

bootpd recognizes the following tags:

Table 6-2. Bootp Tags

Tag	Description
bf	Bootfile
bs	Bootfile size in 512-octet (byte) blocks
ha	Host hardware address
hd	Bootfile home directory
hn	Send hostname
ht	Host hardware type
ip	Host IP address
sm	Host subnet mask
tc	Table continuation (points to similar "template" entry)
vm	Vendor magic cookie selector

There is also a generic tag, Tn, where n is an RFC-1048 vendor field tag number. This enables immediate use of future extensions to RFC-1048 without first modifying bootpd. Generic data may be represented as either a stream of hexadecimal numbers or as a quoted string of ASCII characters. The length of the generic data is automatically determined and inserted into the proper field(s) of the RFC-1048-style BOOTP reply.

The ip and sm tags each expect a single IP address. All IP addresses are specified in standard Internet dot notation and may use decimal, octal, or hexadecimal numbers (octal numbers begin with 0, hexadecimal numbers begin with 0x or 0X).

Hardware Type

The `ht` tag specifies the hardware type code as listed below:

- unsigned decimal, octal, or hexadecimal integer
- `ethernet` or `ether` for 10Mb Ethernet

Address

The `ha` tag takes a hardware address. The hardware address must be specified in hexadecimal. You can include optional periods and/or a leading `0x` for readability. The `ha` tag must be preceded by the `ht` tag (either explicitly or implicitly; see `tc`).

Host Name, Home Directory, and Bootfile

The host name, home directory, and bootfile are ASCII strings which can optionally be surrounded by double quotes (" "). The client's request and the values of the `hd` and `bf` symbols determine how the server fills in the bootfile field of the BOOTP reply packet.

- If the client specifies an absolute path name and that file exists on the server machine, that path name is returned in the reply packet.
- If the file cannot be found, the request is discarded and a reply is not sent.
- If the client specifies a relative path name, a full path name is formed by appending the value of the `hd` tag and testing for the file's existence.
- If the `hd` tag is not supplied in the configuration file or if the resulting bootfile cannot be found, the request is discarded. Because BOOTP clients normally supply `os9boot` as the bootfile name, the relative path name case is used. OS-9 BOOTP clients normally supply `sysboot` as the bootfile name.

Clients specifying null boot files elicit a reply from the server. The exact reply depends on the `hd` and `bf` tags.

- If the `bf` tag specifies an absolute path name and the file exists, that path name is returned in the reply packet.
- If the `hd` and `bf` tags together specify an accessible file, that file name is returned in the reply.
- If a complete file name cannot be determined or the file does not exist, the reply contains a zeroed-out bootfile field.

In each case, existence of the file means, in addition to actually being present, the public read access bit of the file must be set. `tftpd` requires this to permit the file transfer. Set the `hd` tag to `/h0/TFTPBOOT` or to the same directory as given on the `tftpd` command line.

All file names are first tried as `filename.hostname` and then as `filename`. This provides for individual per-host bootfiles.

The following table further illustrates the interaction between hd, bf, and the bootfile name received in the BOOTP request.

Table 6-3. BOOTP Request Matrix

Homedir Specified	Bootfile Specified	Client's file Specification	Action
No	No	Null	Send null file name
No	No	Relative	Discard request
No	Yes	Null	Send if absolute else discard request
No	Yes	Relative	Discard request
Yes	No	Null	Send null file name
Yes	No	Relative	Lookup with .host
Yes	Yes	Null	Send home/boot or bootfile
Yes	Yes	Relative	Lookup with .host

Bootfile Size

The bootfile size, bs, may be a decimal, octal, or hexadecimal integer specifying the size of the bootfile in 512-octet blocks, or the keyword auto. Specifying auto causes the server to automatically set the bootfile size to the actual size of the named bootfile at each request. Specifying the bs symbol as Boolean has the same effect as specifying auto as its value. OS-9 BOOTP clients require bs or bs=auto.

Sending a Host Name

The hn tag is strictly a Boolean tag. It does not take the usual equal sign and value. Its presence indicates the host name should be sent to RFC-1048 clients. bootpd attempts to send the entire host name as it is specified in the configuration file. If this does not fit into the reply packet, the name is truncated to just the host field (up to the first period, if present) and then tried. In no case is an arbitrarily truncated host name sent. If nothing reasonable fits, nothing is sent.

Sharing Common Values Between Tags

Often, many host entries share common values for certain tags (such as name servers). Rather than repeatedly specifying these tags, you can list a full specification for one host entry and shared by others using the tc (table continuation) tag. The template entry is often a dummy host which does not actually exist and never sends BOOTP requests. This feature is similar to the tc feature of termcap for similar terminals.

bootpd allows the tc tag symbol to appear anywhere in the host entry, unlike termcap which requires it to be the last tag.

Information explicitly specified for a host always overrides information implied by a tc tag symbol, regardless of its location within the entry. The tc tag may be the host name or IP address of any host entry previously listed in the configuration file.

Sometimes you need to delete a specific tag after it has been inferred with `tc`. To delete the tag, use the construction `tag@`. This removes the effect of the tag.

For example, to completely undo the host directory specification, use `:hd@:` at an appropriate place in the configuration entry. After removal with `@`, you can reset a tag using `tc`.

Blank lines and lines beginning with a pound sign (`#`) are ignored in the configuration file. Host entries are separated from one another by new lines. You can extend a single host entry over multiple lines if the lines end with a backslash (`\`). You can also have lines longer than 80 characters.

Tags may appear in any order, with the following exceptions:

- The host name must be the very first field in an entry.
- The hardware type must precede the hardware address.
- Individual host entries must not exceed 1024 characters.

bootptab File Example

An example `/h0/TFTPBOOT/bootptab` file follows:

```
# First, we define a global entry which specifies the stuff every host
uses.
#
# the bs tag is required for OS-9 BOOTP clients
# bf is set (to anything) to cause the bootfile.hostname lookup action
#
global.dummy:sm=255.255.255.0:hd=/h0/tftpboot:bs:
#
# individual hosts
#
boop:tc=global.dummy:ht=ethernet:ha=08003E205284:ip=192.52.109.96:
#
vite:tc=global.dummy:ht=ethernet:ha=08003e20c300:ip=192.52.109.57:
#
boesky:tc=global.dummy:ht=ethernet:ha=08003E202eae:ip=192.52.109.61:
```

7 Utilities

This chapter examines LAN Communications utilities provided with this package.

Overview

The following utilities are provided with LAN Communications.

Table 7-1. LAN Communications Utilities

Utility	Description
arp	Print and update the ARP table.
bootptest	Test the bootpd and tftpd daemons.
dhcp	DHCP client negotiation utility.
ftp	File Transfer Protocol. Transfer files to and from remote systems.
hostname	Prints or sets the string returned by the socket library gethostname() function.
idbgen	Internet Database Generation. idbgen builds the internet data module from the data files: host.conf, hosts, hosts.equiv, inetd.conf, networks, protocols, resolv.conf, interfaces.conf, routes.conf services, drpw, idbgen must be run each time any of these files are updated.
idbdump	Internet Database Display. idbdump dumps the current entries in the internet data module (inetdb).
ifconfig	Displays and modifies the interface table. ifconfig allows the addition of new interfaces, modification of IP addresses and broadcast addresses, and deletion of IP addresses.
ipstart	Initializes IP stack.
mbdump	Display current state of system mbuf pool.
ndbmod	Allows you to add, remove, or modify information stored in the inetdb and inetdb2 data modules.
netstat	Report network information and statistics.
ping, ping6	Send ICMP ECHO_REQUEST packets to host.
route	Add or delete routes.
routed, route6d	Dynamic routing daemon.
rtsol	Router solicitation.
telnet	Telnet user interface; telnet provides the ability to log on to remote systems.
tftp	Trivial File Transfer Protocol. tftp transfers files to and from remote systems.

All LAN Communications utilities and servers use the netdb shared module and the inetdb and inetdb2 data modules for name resolution.

Windows 95/NT verisions of the idbgen, idbdump, rpcdbgen, and rpcdump utilities are provided in addition to the OS-9 versions.

Daemon server programs and connection handlers are identified in the following table.

Table 7-2. Daemon Server Programs and Connection Handlers

Daemon	Description
bootpd	Bootp Server Daemon.
ftpd	FTP Server Daemon.
ftpdc	FTP Server Connection Handler (forked by ftpd or inetd).
inetd	Internet Services Master Daemon. inetd can be configured to fork a particular program to handle data on a particular protocol/port number combination. inetd can replace the ftpd and telnetd server daemons. telnetdc and ftpdc must still be available.
routed, route6d	Dynamic Routing Daemon.
telnetd	Telnet Server Daemon.
telnetdc	Telnet Server Connection Handler (forked by telnetd or inetd).
tftpd	TFTP Server Daemon.
tftpdc	TFTP Server Connection Handler (forked by tftpd).

Utilities

This section includes utility definitions in alphabetical order according to the following alpha sort rules:

1. Special characters (not letters, numbers, or underscores) are listed first.
2. Utilities are listed in alphabetic order next without regard for numbers and underscores.
3. If two utility names are identical using these rules, then they are alphabetized according to the following order:
4. Symbols
5. Underscores
6. Alphabetic characters
7. Numbers

Syntax Usage

Each utility description includes a syntactical description of the command line. These symbolic descriptions use the following notations:

`[ ]` Enclosed items are optional

`{ }` Enclosed items may be used 0, 1, or multiple times

`< >` Enclosed item is a description of the parameter to use. For example:

- `<path>` is a legal pathlist.
- `<devname>` is a legal device name.
- `<modname>` is a legal memory module name.
- `<procID>` is a process number.
- `<opts>` is one or more options specified in the command description.
- `<arglist>` is a list of parameters.
- `<text>` is a character string ended by end-of-line.
- `<num>` is a decimal number, unless otherwise specified.
- `<file>` is an existing file.
- `<string>` is an alphanumeric string of ASCII characters.

arp
Ethernet/IP Address Resolution Display and Control for IPv4

Syntax

```
arp [<opts>]
```

IP Functionality

IPv4 addresses only

Options

`<hostname>`
Display ARP entry for `<hostname>`.

`-a`
Display all of the ARP table entries.

`-d <hostname>`
Delete an entry for the host called `hostname`. This option can only be used by the super-user.

`-n`
This option can be used when specifying a single hostname, or with the `-a` option. It indicates that IP addresses should not be resolved to hostnames. A `"?"` will be printed instead of the hostname.

`-s <hostname> <eth_addr> [temp][pub]`

Create an ARP entry for the host called `<hostname>` with the Ethernet address `<ether_addr>`.

The Ethernet address is given as six hex bytes separated by colons. The entry is permanent unless the word `temp` is given in the command. If the word `pub` is given, the entry will be published. For instance, this system responds to ARP requests for `hostname` even though the hostname is not its own. This option can only be used by the super-user.

Description

The `arp` program displays and modifies the Internet-to-Ethernet address translation tables used by the address resolution protocol (ARP). This table is maintained by the `spenet` driver. The age field indicates the number of minutes the entry has been in the table. Non-permanent entries are removed after 20 minutes.

With no flags, the program displays the current ARP entry for `<hostname>`. The host may be specified by name or by number, using Internet dot notation.

Examples

Publish a temporary arp entry for a machine called `odin`.

```
arp -s odin 04:00:00:12:34:56 pub temp
```

This entry will expire after 20 minutes. To make it permanent, leave the `temp` qualifier off.

See Also

`ifconfig`

`netstat`

bootpd
BOOTP Request Server Daemon

Syntax

```
bootpd [<opts>] {<configfile>}
```

IP Functionality

IPv4 addresses only

Options

`-?`
Display the syntax, options, and command description of `bootpd`.

`-d`
Log debug information to `<stderr>`.

`-t <num>`
Exit after `<num>` minutes of no activity.

Description

`bootpd` is the server daemon handling client BOOTP requests. `bootpd` must be run as super user.

The `-d` option causes `bootpd` to display request activity that is useful to diagnose BOOTP client request problems. Each additional `-d` (up to three) appearing on the command line gives more debugging messages.

Each time a client request is received, `bootpd` checks to see if the `<configfile>` has been updated since the last request. This enables changes to `<configfile>` without restarting `bootpd`. By default, configfile is `/h0/TFTPBOOT/bootptab`.

`bootpd` is normally run in a LAN Communications startup file as follows:

```
bootpd /h0/TFTPBOOT/bootptab <>>>/nil&
```

`bootpd` looks in `inetdb` (using `getservbyname()`) to find the port numbers it should use. Two entries are extracted:

`bootps`	the bootp server listening port
`bootpc`	the destination port used to reply to clients

If the port numbers cannot be determined this way, the port numbers are assumed to be 67 for the server and 68 for the client.

- End the command line with an ampersand (`&`) to place `bootpd` in the background (example, `bootpd<>>>/nil&`).
- `bootpd` is used in conjunction with `tftpd`.

Refer to *Chapter 6, BOOTP Server* for how to set up the BOOTP server.

bootptest
Test Utility for BOOTP Server Response

Syntax

```
bootptest -h=<hostname> -e=<etheraddr>
-n=<filename> [<opts>]
```

IP Functionality

IPv4 addresses only

Options

`-h=<hostname>`
Target server IP address (name or dotted decimal).

`-e=<etheradr>`
Ethernet address in colon notation.

`-n=<filename>`
Bootfile name for `bootp` server.

`-f=<filename>`
Copy bootfile into `<filename>`.

Description

`bootptest` sends a BOOTP request to the network and waits for a response from a BOOTP server. If a response is received, `bootptest` attempts to read the bootfile from the server. `bootptest` provides a way to test a BOOTP server setup without using an actual diskless client.

The `-h`, `-e`, and `-n` options are required and must appear on the command line. `-h` accepts a name which is converted to an IP address using `gethostbyname()`. If a host name is unavailable, the IP address can be given in dotted decimal notation.

To broadcast, specify 0 or 255 as the host portion of the IP network address. This solicits a response from any BOOTP server on the named network.

The BOOTP client test utility may use all ones (255.255.255.255) for the server IP address when it boots if it does not yet know its IP address. An IP address of all ones is received as a broadcast by any IP host with a socket bound to the `bootps` port (UDP 67). `bootpd` uses the contents of the BOOTP message to indicate where the broadcast came. Otherwise, `bootptest` can use the IP address of the system.

The `bootptab` configuration file on the `bootpd` server must specify an entry for the system on which `bootptest` is running. `bootptest` cannot perform a proxy test for another host because the `bootpd` server directs the BOOTP response to the intended client's IP address, not the IP address from which `bootptest` is running.

The most useful test is a simple assurance test that `bootpd` is properly running on the server system. Run `bootptest` naming `loopback` or the host's own hostname and see if a response is received from `bootpd`. Use the `-d` option in `bootpd` to display log messages.

Example

The following is an example of `bootptest`:

```
bootptest -h=192.52.109.255 -e=8:0:3E:20:52:84 -n=os9boot
```

dhcp
DHCP Client Negotiation Utility

Syntax

```
dhcp [<eth_dev>] [<opts>]
```

IP Functionality

IPv4 addresses only

Options

`-v`
Verbose mode. Print additional information about what the program is doing.

`eth_dev`
Name of Ethernet device in `inetdb` module.

`-broadcast <address>`
Set the expected DHCP broadcast address.

`-nofork`
Do not fork child DHCP process.

`-override`
This option allows existing network configurations, such as DNS name servers, to be overridden by DHCP server-supplied information.

`-timeout <seconds>`
Number of seconds to wait for DHCP reply.

`-port <port>`
UDP port of DHCP server.

Description

`dhcp` is the DHCP client negotiation utility. The `eth_dev` parameter is the name of the Ethernet interface used with `ndbmod` or `idbgen`. This name should not be confused with the ethernet device descriptor or driver. If no device is specified on the command line, and only one interface was added with `ndbmod` or `idbgen`, `dhcp` uses it.

`broadcast`	Specify that the DHCP server is not correctly broadcasting IP packets to the correct address.
`timeout`	Enable the user to specify how long `dhcp` waits before retrying a failed request. The default is 10 seconds.
`-nofork`	Stop the `dhcp` client creating a child process and exiting. This is not normally used.

If the lease time on the IP address supplied by the DHCP server expires, DHCP removes the address, and reverts back to requesting an IP address.

If the DHCP Server supplies DNS information, `dhcp` attempts to add it to an `inetdb` module. Space for this is provided by creating a new `inetdb3` with the `ndbmod` command.

Examples

Sample output from `dhcp` using the verbose mode.

```
# dhcp -v
DHCP: Ethernet device name 'enet0'
DHCP: Ethernet SPF descriptor name '/spe30'
DHCP: Eth Address is 00:60:97:8C:28:7B
DHCP: Adding IP address: 0.0.0.0
DHCP: Adding broadcast address: 255.255.255.255
DHCP: Adding subnet mask: 0.0.0.0
Sending DHCPDISCOVER to 255.255.255.255
Sending DHCPREQUEST to 255.255.255.255
DHCP: Adding IP address: 192.168.3.200
DHCP: Adding broadcast address: 192.168.3.255
DHCP: Adding subnet mask: 255.255.255.0
DHCP: Received a lease for 3600 seconds
DHCP: adding default route to 192.168.3.225
DHCP: Offered domain name: dm1.radisys.com
DHCP: Offered DNS Server 172.16.1.32
DHCP: Adding DNS Server 172.16.1.32
```

See Also

`ifconfig`

`ndbmod`

ftp

File Transfer Manipulation/Remote Internet Site Communication

Syntax

```
ftp [-AadfginpRtvV] [-o <output>] [-P <port>] [-r <retry>]
{[<user@>]<host> [<port>]} {user@]<host>:[<path>][/} {<file>:///<path>}
{ftp://[<user>[:<password>]@]<host>[:<port>]/<path>[/]]

{http://[<user>[:<password>]@]<host>[:<port>]/<path>}


ftp [-u <url> file]
```

IP Functionality

IPv4 and IPv6 addresses

Options

Options may be specified at the command line or to the command interpreter.The following options exist for `ftp`.

`-A`

Force active mode `ftp`.

By default, `ftp` will try to use passive mode `ftp` and fall back to active mode if passive is not supported by the server. This option causes `ftp` to always use an active connection. It is only useful for connecting to very old servers that do not implement passive mode properly.

`-a`

Cause `ftp` to bypass normal login procedure, and use an anonymous login instead.

`-d`

Enable debug messages.

`-f`

Force a cache reload for transfers that go through the FTP or HTTP proxies.

`-g`

Disable file name globbing.

`-i`

Turn off interactive prompting during multiple file transfers.

`-n`

Restrain ftp from attempting “auto-login” upon initial connection.

If auto-login is enabled, `ftp` will check the `.netrc` file in the user's home directory (as defined by the environment variable `HOME`) for an entry describing an account on the remote machine. If no entry exists, `ftp` will prompt for the remote machine login name. (The default is the user identity on the local machine.) If necessary, it will also prompt for a password and an account with which to login.

`-o <output>`

Save the contents in `<output>` when auto-fetching files.

`<output>` is parsed according to the file naming conventions below. If output is not “-”' or doesnot start with “I”, only the first file specified will be retrieved into `<output>`; all other files will be retrieved into the basename of their remote name.

`-p`

Force passive mode operation and disables fall back to active mode. Use behind connection filtering firewalls.

Without this option, `ftp` attempts to use passive mode by default, falling back to active mode if the server does not support passive connections.

`-P <port>`

Set the port number to `<port>`.

`-r <wait>`

Retry the connection attempt if it failed, pausing for `<wait>` seconds.

`-R`

Restart all non-proxied auto-fetches.

Enable packet tracing (unimplimented).

`-u <url> <file> [...]`

Upload files on the command line to `<url>`, where `<url>` is one of the `ftp` URL types as supported by auto-fetch (with an optional target filename for single file uploads), and `<file>` is one or more local files to be uploaded.

`-v`

Enable verbose and progress.

This is the default if output is to a terminal (and in the case of progress, `ftp` is the foreground process). `ftp` is forced to show all responses from the remote server, as well as report on data transfer statistics.

`-V`

Disable verbose and progress, overriding the default of enabled when output is to a terminal.

`-?`

Display command line usage.

Description

`ftp` is the user interface to the Internet standard File Transfer Protocol. The program allows a user to transfer files to and from a remote network site. The last five arguments fetch a file using the FTP or HTTP protocols, or by direct copying, into the current directory. This is ideal for scripts.

Commands

Command arguments which have embedded spaces may be quoted with double quotation marks (“ ”). Commands that toggle settings can take an explicit on or off argument to force the setting appropriately.

Commands that take a byte count as an argument support an optional suffix on the argument which changes the interpretation of the argument. Supported suffixes are listed below:

b	Causes no modification. (optional)
k	Kilo; multiply the argument by 1024
m	Mega; multiply the argument by 1048576
g	Giga; multiply the argument by 1073741824

The client host with which `ftp` communicates may be specified on the command line. If this is done, `ftp` will immediately attempt to establish a connection to an FTP server on that host; otherwise, `ftp` will enter its command interpreter and await instructions from the user. When `ftp` is awaiting commands from the user, the prompt `ftp>` is provided to you. The following commands are recognized by `ftp`:

`! [<command> [<args>]]`

Invoke an interactive shell on the local machine.

If there are arguments, the first is taken to be a command to execute directly and the rest of the arguments act as its arguments.

`$ <macro-name> [<args>]`

Execute the macro macro-name that was defined with the `macdef` command.

Arguments are passed to the macro unglobbed.

`account [<passwd>]`

Supply a supplemental password required by a remote system for access to resources once a login has been successfully completed.

If no argument is included, you are prompted for an account password in a non-echoing input mode.

`append <local-file> [<remote-file>]`

Append a local file to a file on the remote machine.

If `<remote-file>` is left unspecified, the local file name is used in naming the remote file after being altered by any `ntrans` or `nmap` setting. File transfer uses the current settings for type, format, mode, and structure.

`ascii`
Set the file transfer type to ASCII network.

This is the default type.

`bell`
Arrange for a bell to sound after each file transfer command is completed.

`binary`
Set the file transfer type to support binary image transfer.

`bye`
Terminate the FTP session with the remote server and exit `ftp`.

An end of file will also terminate the session and exit.

`case`
Toggle remote computer file name case mapping during mget commands.

When case is on (default is off), remote computer file names with all letters in upper case are written in the local directory with the letters mapped to lower case.

`cd <remote-directory>`
Change the working directory on the remote machine to `<remote-directory>`.

`cdup`
Change the remote machine working directory to the parent of the current remote machine working directory.

`chd <remote-directory>`
This is a synonym for `cd`.

`chmod <mode> <remote-file>`
Change the permission modes of the file `<remote-file>` on the remote system to `<mode>`.

`close`
Terminate the FTP session with the remote server and return to the command interpreter.

Any defined macros are erased.

`cr`
Toggle carriage return stripping during ASCII-type file retrieval.

Records are denoted by a carriage return/linefeed sequence during ASCII-type file transfer. When `cr` is on (the default), carriage returns are stripped from this sequence to conform with the UNIX single linefeed record delimiter. Records on non-UNIX remote systems may contain single linefeeds; when an ASCII-type transfer is made, these linefeeds may be distinguished from a record delimiter
(only when `cr` is off).

`debug [<debug-value>]`
Toggle debugging mode.

If an optional `<debug-value>` is specified, it is used to set the debugging level. When debugging is on, ftp prints each command sent to the remote machine, preceded by the string "-->".

`delete <remote-file>`
Delete the file `<remote-file>` on the remote machine.

`dir [<remote-directory> [<local-file>]]`
Print a listing of the contents of a directory on the remote machine.

The listing includes any system-dependent information that the server chooses to include; for example, most UNIX systems will produce output from the command `ls -l`. If the `<remote-directory>` is left unspecified, the current working directory is used.

If interactive prompting is on, `ftp` will prompt you to verify that the last argument is indeed the target local file for receiving `dir` output. If no local file is specified, or if `<local-file>` is "-", the output is sent to the terminal.

`disconnect`
This is a synonym for `close`.

`epsv4`
Toggle the use of the extended `EPSV` and `EPRT` commands on IPv4 connections.

First, try `EPSV/EPRT`, then `PASV/PORT`, which is enabled by default. If an extended command fails, this option is temporarily disabled for the duration of the current connection, or until `epsv4` is executed again.

`exit`
This is a synonym for `bye`.

`fget <localfile>`
Retrieve the files listed in `<localfile>`, which have one line per filename.

`form <format>`
Set the file transfer form to `<format>`.

The default format is "non-print".

`ftp <host> [<port>]`
This is a synonym for `open`.

`gate [<host> [<port>]`
Toggle gate-ftp mode, which used to connect through the TIS FWTK and Gauntlet ftp proxies.

This will not be permitted if the gate-ftp server has not been set (either by you or the `FTPSERVER` environment variable). If host is given,
gate-ftp mode is enabled and the gate-ftp server is set to `<host>`. If `<port>` is also given, it used as the port to connect to on the gate-ftp server.

`get <remote-file> [<local-file>]`
Retrieve the `<remote-file>` and store it on the local machine.

If the local file name is not specified, it is given the same name it has on the remote machine, subject to alteration by the current `case`, `ntrans`, and `nmap` settings. The current settings for `type`, `form`, `mode`, and `structure` are used while transferring the file.

`glob`
Toggle filename expansion for `mdelete`, `mget` and `mput`.

If globbing is turned off with `glob`, the file name arguments are taken literally and not expanded. For `mdelete` and `mget`, each remote file name is expanded separately on the remote machine and the lists are not merged.

Expansion of a directory name is likely to be different from expansion of the name of an ordinary file: The exact result depends on the foreign operating system and ftp server, and can be previewed with `mls remote-files -`.

`mget` and `mput` are not meant to transfer entire directory subrees of files. This can be done by transferring a `tar()` archive of the subtree (in binary mode).

`hash [<size>]`
Toggle hash-sign (“#”) printing for each data block transferred.

The size of a data block defaults to 1024 bytes. This can be changed by specifying `<size>` in bytes. Enabling `hash` disables `progress`.

`help [<command>]`
Print an informative message about the meaning of `<command>`.

If no argument is given, `ftp` prints a list of the known commands.

`idle [<seconds>]`
Set the inactivity timer on the remote server to seconds seconds.

If `<seconds>` is omitted, the current inactivity timer is printed.

`image`
This is a synonym for `binary`.

`lcd [<directory>]`
Change the working directory on the local machine. If no directory is specified, your home directory is used.

`lpwd`
Print the working directory on the local machine.

`ls [<remote-directory> [<local-file>]]`
This is a synonym for `dir`.

`macdef <macro-name>`
Define a macro.

Subsequent lines are stored as the macro `macro-name`; a null line (consecutive newline characters in a file or carriage returns from the terminal) terminates macro input mode. There is a limit of 16 macros and 4096 total characters in all defined macros.

Macros remain defined until a close command is executed. The macro processor interprets "$" and "\" as special characters. A "$" followed by a number (or numbers) is replaced by the corresponding argument on the macro invocation command line.

A "$" followed by an "i" signals that macro processor that the executing macro is to be looped. On the first pass, "$i" is replaced by the first argument on the macro invocation command line; on the second pass it is replaced by the second argument. A "\" followed by any character is replaced by that character. Use the "\" to prevent special treatment of the "$".

`mdelete [<remote-files>]`
Delete the remote-files on the remote machine.

`mdir <remote-files> <local-file>`
This is similar to `dir`; however, with this option, multiple remote files may be specified and the `<local-file>` must be specified.

If interactive prompting is on, `ftp` will prompt you to verify that the last argument is indeed the target local file for receiving mdir output. A `<local-file>` of "-" sends the output to `stdout`.

`mget <remote-files>`
Expand the remote-files on the remote machine and do a get for each file name thus produced.

Refer to `glob` for details on the filename expansion. Resulting file names are processed according to `case`, `ntrans`, and `nmap` settings. Files are transferred into the local working directory, which can be changed with `lcd directory`; new local directories can be created with `! makdir directory`.

`mkdir <directory-name>`
Make a directory on the remote machine.

`mls <remote-files> <local-file>`
This is similar to `ls`; however, with this option multiple remote files may be specified and the `<local-file>` must be specified.

If interactive prompting is on, `ftp` prompts you to verify that the last argument is the target local file for receiving `mls` output. A local-file of "-" sends the output to `stdout`.

`mode <mode-name>`
Set the file transfer mode to mode-name.

The default mode is `stream` mode.

`modtime <remote-file>`
Show the last modification time of the file on the remote machine.

`mput <local-files>`
Expand wild cards in the list of local files given as arguments and do a put for each file in the resulting list.

Refer to `glob` for details of filename expansion. Resulting file names will then be processed according to `ntrans` and `nmap` settings.

`msend <local-files>`
This is a synonym for `mput`.

`newer <remote-file> [<local-file>]`
Get the file only if the modification time of the remote file is more recent that the file on the current system.

If the file does not exist on the current system, the remote file is considered newer. Otherwise, this command is identical to get.

`nlist [<remote-directory> [<local-file>]]`
This is a synonym for `ls`; however, this option sends the `NLST` command, rather than the `LIST` command, to the remote server.

`nmap [<inpattern outpattern>]`
Set or unset the filename mapping mechanism.

If no arguments are specified, the filename mapping mechanism is unset. If arguments are specified, remote filenames are mapped during mput commands and put commands issued without a specified remote target filename. If arguments are specified, local filenames are mapped during mget commands and get commands issued without a specified local target filename.

This command is useful when connecting to a remote computer with different file naming conventions or practices. The mapping follows the pattern set by inpattern and outpattern.

`[<Inpattern>]`

This is a template for incoming filenames (which may have already been processed according to the `ntrans` and `case` settings). Variable templating is accomplished by including the sequences "$1", "$2", and all others through "$9" in `inpattern`. Use "\" to prevent this special treatment of the "$" character. All other characters are treated literally and are used to determine the nmap `[inpattern]` variable values. For example, given inpattern "$1.$2" and the remote file name `mydata.data`, "$1" would have the value `mydata`, and "$2" would have the value `data`.

`[<outpattern>]`

This determines the resulting mapped filename. The sequences "$1", "$2", and all others through "$9" are replaced by any value resulting from the inpattern template. The sequence "$0" is replace by the original filename. Additionally, the sequence `[seq1, seq2]` is replaced by `[seq1] if seq1` is not a null string; otherwise, it is replaced by `seq2`.

For example, the command below yields the output filename `myfile.data` for input filenames `myfile.data` and `myfile.data.old`; `myfile.file` for the input filename `myfile`; and `myfile.myfile` for the input filename `.myfile`:

```
nmap $1.$2.$3 [$1,$2].[$2,file]
```

Spaces may be included in outpattern, as in the following example:

```
nmap $1 sed "s/  *$//" > $1
```

Use the "\" character to prevent special treatment of the "$","[',']", and "," characters.

`ntrans [<inchars>[<outchars>]]`

Set or unset the filename character translation mechanism.

If no arguments are specified, the filename character translation mechanism is unset. If arguments are specified, characters in remote filenames are translated during mput commands and put commands issued without a specified remote target filename. If arguments are specified, characters in local filenames are translated during `mget` commands and get commands issued without a specified local target filename.

This command is useful when connecting to a remote computer with different file naming conventions or practices. Characters in a filename matching a character in inchars are replaced with the corresponding character in outchars. If the character's position in `inchars` is longer than the length of `outchars`, the character is deleted from the file name.

`open <host> [<port>]`

Establish a connection to the specified host FTP server.

An optional port number may be supplied, in which case, `ftp` will attempt to contact an ftp server at that port. If the autologin option is on (default), `ftp` will also attempt to automatically log the user in to the FTP server.

`passive [auto]`

Toggle passive mode (if no arguments are given).

If `auto` is given, act as if `FTPMODE` is set to "auto". If passive mode is turned on (default), `ftp` will send a `PASV` command for all data connections instead of a `PORT` command. The `PASV` command requests that the remote server open a port for the data connection and return the address of that port. The remote server listens on that port and the client connects to it.

When using the more traditional `PORT` command, the client listens on a port and sends that address to the remote server, who connects back to it. Passive mode is useful when using `ftp` through a gateway router or host that controls the directionality of traffic. (Note that though FTP servers are required to support the `PASV` command by RRC 1123, some do not.)

`preserve`

Toggle preservation of modification times on retrieved files.

`progress`

Toggle display of transfer progress bar.

The progress bar will be disabled for a transfer that has local-file as "-" or a command that starts with "|". Enabling `progress` disables `hash`. Refer to FILENAMING CONVENTIONS for more information.

`prompt`

Toggle interactive prompting.

Interactive prompting occurs during multiple file transfers to allow the user to selectively retrieve or store files. If prompting is turned off (default is on), any `mget` or `mput` will transfer all files, and any `mdelete` will delete all files.

When prompting is on, the following commands are available at a prompt:

`a`	Answer "yes" to the current file and automatically answer "yes" to any remaining files for the current command.
`n`	Answer "no" and do not transfer the file.
`p`	Answer "yes" to the current file and turn off prompt mode (as is "prompt off" had been given).
`q`	Terminate the current operation.
`y`	Answer "yes" and transfer the file.
`?`	Display a help message.

Any other reponse will answer "yes" to the current file.

`proxy <ftp-command>`

Execute an `ftp` command on a secondary control connection.

This command allows simultaneous connection to two remote FTP servers for transferring files between the two servers. The first proxy command should be an "open" command so that it can establish the secondary control connection. Enter the command `proxy ?` to see other FTP commands executable on the secondary connection.

The following commands behave differently when prefaced by proxy:

`open`	Will not define new macros during the auto-login process
`close`	Will not erase existing macro definitions
`get` and `mget`	Transfer files from the host on the primary control connection to the host on the secondary control connection
`put`, `mput`, and `append`	Transfer files from the host on the secondary control connection to the host on the primary control connection.

Third party file transfers depend upon support of the FTP protocol `PASV` command by the server on the secondary control connection.

`put <local-file> [<remote-file>]`
Store a local file on the remote machine.

If remote-file is left unspecified, the local-file name is used after processing according to any ntrans or nmap settings in naming the remote file. File transfer uses the current settings for type, format, mode, and structure.

`pwd`
Print the name of the current working directory on the remote machine.

`quit`
This is a synonym for `bye`.

`quote <arg1> <marg2> ...`
The arguments specified are sent, verbatim, to the remote FTP server.

`recv <remote-file> [<local-file>]`
This is a synonym for `get`.

`reget <remote-file> [<local-file>]`
`reget` acts like `get`; however, if `local-file` exists and is smaller than `remote-file`, `local-file` is presumed to be a partially transferred copy of `remote-file` and the transfer is continued from the apparent point of failure.

This command is useful when transferring very large files over networks that are prone to dropping connections.

`rename [<from> [<to>]]`
Rename the file `from` on the remote machine to the file `to`.

`reset`
Clear reply queue.

This command re-synchronizes command/reply sequencing with the remote FTP server. Resynchronization may be necessary following a violation of the FTP protocol by the remote server.

`restart <marker>`
Restart the immediately following get or put at the indicated marker.

On UNIX systems, marker is usually a byte offset into the file.

`rhelp [<command-name>]`
Request help from the remote FTP server.

If a command-name is specified it is supplied to the server as well.

`rmdir <directory-name>`
Delete a directory on the remote machine.

`rstatus [<remote-file>]`
With no arguments, show status of remote machine.

If `remote-file` is specified, show status of `remote-file` on remote machine.

`runique`
Toggle storing of files on the local system with unique filenames.

If a file already exists with a name equal to the target local filename for a get or mget command, a ".1" is appended to the name. If the resulting name matches another existing file, a ".2" is appended to the original name. If this process continues up to ".99", an error message is printed and the transfer does not take place. The generated unique filename will be reported. Note that runique will not affect local files generated from a shell command (see below). The default value is off.

`send <local-file> [<remote-file>]`
This is a synonym for `put`.

`sendport`
Toggle the use of `PORT` commands.

By default, `ftp` will attempt to use a `PORT` command when establishing a connection for each data transfer. The use of `PORT` commands can prevent delays when performing multiple file transfers. If the `PORT` command fails, `ftp` will use the default data port. When the use of `PORT` commands is disabled, no attempt will be made to use `PORT` commands for each data transfer. This is useful for certain FTP implementations that ignore `PORT` commands but, incorrectly, indicate that they have been accepted.

`set [<option> <value>]`
Set option to value.

If `<option>` and `<value>` are not given, display all of the options and their values. The currently supported options are listed below:

`anonpass` Defaults to `$FTPANONPASS`.

`ftp_proxy` Defaults to `$ftp_proxy`.

`http_proxy` Defaults to `$http_proxy`.

`no_proxy` Defaults to `$no_proxy`.

`prompt` Defaults to `$FTPPROMPT`.

`rprompt` Defaults to `$FTPRPROMPT`.

`size <remote-file>`
Return size of `remote-file` on remote machine.

`site <arg1> <arg2>...`
The arguments specified are sent, verbatim, to the remote FTP server as a `SITE` command.

`status`
Show the current status of `ftp`.

`struct <struct-name>`
Set the file transfer structure to `struct-name`.

By default `file` structure is used.

`sunique`
Toggle storing of files on remote machine under unique file names.

The remote FTP server must support FTP protocol `STOU` command for successful completion. The remote server will report unique name. Default value is off.

`system`
Show the type of operating system running on the remote machine.

`tenex`
Set the file transfer type to that needed to talk to `TENEX` machines.

`trace`
Toggle packet tracing. (unimplemented)

`type [<type-name>]`
Set the file transfer type to `type-name`.

If no type is specified, the current type is printed. The default type is network ASCII.

`umask [<newmask>]`
Set the default umask on the remote server to `newmask`. If `newmask` is omitted, the current umask is printed.

`unset <option>`
Unset option. Refer to `set` for more information.

`usage <command>`
Print the usage message for command.

`user <user-name> [<password> [<account>]]`
Identify yourself to the remote FTP server.

If the password is not specified and the server requires it, `ftp` will prompt the user for it (after disabling local echo). If an account field is not specified, and the FTP server requires it, you will be prompted for it.

If an account field is specified, an account command will be relayed to the remote server after the login sequence is completed if the remote server did not require it for logging in. Unless `ftp` is invoked with `auto-login` disabled, this process is done automatically on initial connection to the FTP server.

`verbose`
Toggle `verbose` mode.

In `verbose` mode, all responses from the FTP server are displayed to the user. In addition, if `verbose` is on, when a file transfer completes, statistics regarding the efficiency of the transfer are reported. By default, `verbose` is on.

`? [<command>]`
This is a synonym for `help`.

Command arguments which have embedded spaces may be quoted with quote “”marks. Commands which toggle settings can take an explicit on or off argument to force the setting appropriately.

Commands which take a byte count as an argument support an optional suffix on the argument which changes the interpretation of the argument. Supported suffixes are:

b	Cause no modification. (optional)
k	Kilo; multiply the argument by 1024
m	Mega; multiply the argument by 1048576
g	Giga; multiply the argument by 1073741824

Correct execution of many commands depends on proper behavior by the remote server.

Auto-Fetching Files

In addition to standard commands, this version of `ftp` supports an auto-fetch feature. To enable auto-fetch, simply pass the list of hostnames/files on the command line.

The following formats are valid syntax for an auto-fetch element:

[user@]host:[path][/]

Classic FTP format.

If path contains a glob character and globbing is enabled, (refer to `glob`), then the equivalent of `mget path` is performed. If the directory component of path contains no globbing characters, it is stored locally with the basename of `path`, in the current directory. Otherwise, the full remote name is used as the local name, relative to the local root directory.

ftp://[user[:password]@]host[:port]/path[/][;type=X]

An FTP URL, retrieved using the FTP protocol if `set ftp_proxy` is not defined. Otherwise, transfer using HTTP via the proxy defined in `set ftp_proxy`. If `set ftp_proxy` is not defined and user is given, login as user. In this case, use password if supplied, otherwise prompt the user for one.

In order to be compliant with RFC 1738, `ftp` strips the leading “`/`” from path, resulting in a transfer relative from the default login directory of the user. If the “`/`” directory is required, use a leading path of “`%2F`”. If a user's home directory is required (and the remote server supports the syntax), use a leading path of “`%7Euser/`”. For example, to retrieve `/etc/motd` from `localhost` as the user `myname` with the password `mypass`, use `tp://myname:mypass@localhost/%2fetc/motd`.

If a suffix of “`;type=A`” or “`;type=I`” is supplied, then the transfer type will take place as ascii or binary (respectively). The default transfer type is binary.

http://[user[:password]@]host[:port]/path

An HTTP URL, retrieved using the HTTP protocol. If `set http_proxy` is defined, it is used as a URL to an HTTP proxy server. If HTTP authorization is required to retrieve path, and `user` (and optionally `password`) is in the URL, use them for the first attempt to authenticate.

file:///path

A local URL, copied from `/path`.

Unless noted otherwise above, and `-o` output is not given, the file is stored in the current directory as the basename of path. If a classic format or a FTP URL format has a trailing "/", then `ftp` will connect to the site and `cd` to the directory given as the path, and leave the user in interactive mode ready for further input.

Direct HTTP transfers use HTTP 1.1. Proxied FTP and HTTP transfers use HTTP 1.0. If -R is given, all auto-fetches that do not go via the FTP or HTTP proxies will be restarted. For FTP, this is implemented by using reget instead of get. For HTTP, this is implemented by using the `Range: bytes=` HTTP/1.1 directive.

If WWW or proxy WWW authentication is required, you will be prompted to enter a username and password to authenticate with. When specifying IPv6 numeric addresses in a URL, you need to surround the address in square brackets. (For example: `ftp://super@[::1]:21/`.) This occurs because colons are used in IPv6 numeric address, as well as being the separator for the port number.

Aborting a File Transfer

To abort a file transfer, use the terminal interrupt key (usually Ctrl-C). Sending transfers will be immediately halted. Receiving transfers will be halted by sending an FTP protocol ABOR command to the remote server, and discarding any further data received. The speed at which this is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, the prompt will not appear until the remote server has completed sending the requested file.

If the terminal interrupt key sequence is used while `ftp` is awaiting a reply from the remote server for the ABOR processing, then the connection will be closed. This is different from the traditional behavior (which ignores the terminal interrupt during this phase), but is considered more useful.

File Naming Conventions

Files specified as arguments to ftp commands are processed according to the following rules.

1. If the file name `-' is specified, the `stdin` (for reading) or `stdout` (for writing) is used.
2. If the first character of the file name is `|', the remainder of the argument is interpreted as a shell command. `ftp` then forks a shell, using popen() with the argument supplied, and reads (writes) from the `stdout` (`stdin`). If the shell command includes spaces, the argument must be quoted (i.e. "`| ls -lt`"). A particularly useful example of this mechanism is "`dir . |more`".
3. Failing the above checks, if "globbing" is enabled, local file names are expanded according to the rules. If the ftp command expects a single local file (i.e. `put`), only the first filename generated by the "globbing" operation is used.
4. For `mget` commands and `get` commands with unspecified local file names, the local filename is the remote filename, which may be altered by a `case`, `ntrans`, or `nmap` setting. The resulting filename may then be altered if `runique` is on.
5. For `mput` commands and `put` commands with unspecified remote filenames, the remote filename is the local filename, which may be altered by a `ntrans` or `nmap` setting. The resulting filename may then be altered by the remote server if `sunique` is on.

File Transfer Parameters

The FTP specification specifies many parameters which may affect a file transfer. The `type` may be one of `ascii`, `image` (binary), `ebcdic`, and `local byte size` (for PDP-10's and PDP-20's mostly). `ftp` supports the ascii and image types of file transfer, plus local byte size eight for `tenex` mode transfers. `ftp` supports only the default values for the remaining file transfer parameters: `mode`, `form`, and `struct`.

The .netrc File

The `.netrc` file contains login and initialization information used by the auto-login process. It resides in your home directory. The following tokens are recognized; they may be separated by spaces, tabs, or new-lines:

`machine <name>`

Identify a remote machine name.

The auto-login process searches the `.netrc` file for a machine token that matches the remote machine specified on the ftp command line or as an open command argument. Once a match is made, the subsequent `.netrc` tokens are processed, stopping when the end of file is reached or another machine or a default token is encountered.

`default`
: This is the same as machine name except that default matches any name.

  There can be only one default token, and it must be after all machine tokens. This is normally used as: default login anonymous password user@site, thereby giving you an automatic anonymous FTP login to machines not specified in `.netrc`. This can be overridden by using the `-n` flag to disable auto-login.

`login name`
: Identify a user on the remote machine.

  If this token is present, the auto-login process will initiate a login using the specified name.

`password string`
: Supply a password.

  If this token is present, the auto-login process will supply the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the `.netrc` file for any user other than anonymous, `ftp` will abort the auto-login process if the `.netrc` is readable by anyone besides the user.

`account string`
: Supply an additional account password.

  If this token is present, the auto-login process will supply the specified string if the remote server requires an additional account password, or the auto-login process will initiate an ACCT command if it does not.

`macdef name`
: Define a macro.

  This token functions like the ftp `macdef` command functions. A macro is defined with the specified name; its contents begin with the next .netrc line and continue until a blank line (consecutive new-line characters) is encountered. If a macro named `init` is defined, it is automatically executed as the last step in the auto-login process.

Command Line Prompt

By default, `ftp` displays a command line prompt of `ftp>`. This can be changed with the `set prompt` command. A prompt can be displayed on the right side of the screen (after the command input) with the `set rprompt` command.

The following formatting sequences are replaced by the given information:

`%/`
: The current remote working directory.

`%c[[0]n], %.[[0]n]`
: The trailing component of the current remote working directory, or n trailing components if a digit `n` is given.

  If `n` begins with "0", the number of skipped components precede the trailing component(s) in the format `/<skipped>trailing` (for "%c") or `...trailing` (for "%.").

`%M`
: The remote host name.

`%m`
: The remote host name, up to the first ".".

`%n`
: The remote user name.

`%%`
: A single "%"".

Environment

`ftp` uses the following environment variables.

`FTPANONPASS`
: Password to send in an anonymous FTP transfer. Defaults to `whoami @`.

`FTPMODE`
: Overrides the default operation mode. Support values are:

`active`	active mode FTP only
`auto`	automatic determination of passive or active (this is the default)
`gate`	gate-ftp mode
`passive`	passive mode FTP only

`FTPPROMPT`
: Command-line prompt to use. Defaults to `ftp>`.

`FTPRPROMPT`
: Command-line right side prompt to use.

`FTPSERVER`
: Host to use as gate-ftp server when gate is enabled.

`FTPSERVERPORT`
: Port to use when connecting to gate-ftp server when gate is enabled. Default is port returned by a `getservbyname()` lookup of `ftpgate/tcp`.

`HOME`
: For default location of a `.netrc` file, if one exists.

`SHELL`
: For default shell.

`ftp_proxy`
: URL of FTP proxy to use when making FTP URL requests. If not defined, use the standard FTP protocol.

`http_proxy`
: URL of HTTP proxy to use when making HTTP URL requests.

  If proxy authentication is required and there is a username and password in this URL, they will automatically be used in the first attempt to authenticate to the proxy.

  Note that the use of a username and password in `ftp_proxy` and `http_proxy` may be incompatible with other programs that use it.

`no_proxy`
: A space or comma separated list of hosts (or domains) for which proxying is not to be used. Each entry may have an optional trailing `:port`, which restricts the matching to connections to that port.

See Also

`ftpd`

`tftp`

ftpd
Incoming FTP Server Daemon

Syntax

```
ftpd [-dHlqQrsuUwWX] [-a <anondir>] [-c <confdir>]
[-C <user>] [-e <emailaddr>][-h <hostname>]
[-P <dataport>] [-V <version>]]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-6`
Listen on an IPv6 socket. Default is IPv4.

`-a <anondir>`
Define `<anondir>` as the directory to `chroot()` into for anonymous logins.

Default is the home directory for the `ftp` user. This can also be specified with the `ftpd.conf chroot` directive. <`anondir>` must be a fully qualified path; the same as returned by the `pd` shell command.

`-c <confdir>`
Change the root directory of the configuration files from `/dd/sys` to `confdir`.

This changes the directory for the following files: `ftpchroot`, `ftpusers`, `ftpwelcome`, `motd`, and the file specified by the `ftpd.conf` limit directive.

`-C <user>`
Check whether or not a `<user>` can be granted access under the restrictions given in `ftpusers` and exit without attempting a connection. `ftpd` exits with an exit code of 0 if access would be granted, or 1 otherwise. This can be useful for testing configurations.

Example: `ftpdc -DDD anonymous`

`-d`
Debugging information is written to `stderr`.

`-e <emailaddr>`
Use `<emailaddr>` for the `%E` escape sequence. (Refer to the Display File Escape Sequences.)

`-h <hostname>`
Explicitly set the host name to advertise to `<hostname>`.

The default is the hostname associated with the IP address on which that `ftpd` is listening. This ability (with or without `-h`), in conjunction with `-c <confdir>`, is useful when configuring “virtual” FTP servers, each listening on separate addresses as separate names. Refer to `inetd` for more information on starting services to listen on specific IP addresses.

`-H`
: Explicitly set the hostname to advertise to the "standard hostname" for the current processor. The default is the hostname associated with the IP address on which `ftpd` is listening.

`-l`
: Log each successful and failed FTP session to `stderr`. If this option is specified more than once, the retrieve (get), store (put), append, delete, make directory, remove directory and rename operations and their file name arguments are also logged.

`-P <dataport>`
: Use `<dataport>` as the data port, overriding the default of using the port one less than the port on which `ftpd` is listening.

`-r`
: Permanently drop root privileges once you are logged in. The use of this option may result in the server using a port other than the `<listening-port -1>` for PORT style commands, which is contrary to the RFC 959 specification, but in practice very few clients rely upon this behaviour.

`-u`
: Enable unauthenticated access as super user without password file.

`-V <version>`
: Use `<version>` as the version to advertise in the login banner and in the output of STAT and SYST instead of the default version information. If `<version>` is empty or "-", do not display any version information.

`-X`
: Log wu-ftpd style `xferlog` entries to `stderr`, prefixed with `xferlog:`.

`-?`
: Display command line usage.

Description

`ftpd` is the Internet File Transfer Protocol server process. The server uses the TCP protocol and listens at the port specified in the FTP service specification.

The file `/dd/sys/nologin` can be used to disable FTP access. If the file exists, `ftpd` displays it and exits. If the file `/dd/sys/ftpwelcome` exists, `ftpd` prints it before issuing the "ready" message. If the file `/dd/sys/motd` exists (under the chroot directory if applicable), `ftpd` prints it after a successful login. This may be changed with the `ftpd.conf` directive `motd`.

The `ftpd` server currently supports the following FTP requests. The case of the requests is ignored.

Table 7-3. FTP Requests

Request	Description
ABOR	Abort previous command.
ACCT	Specify account (ignored).
ALLO	Allocate storage (vacuously).
APPE	Append to a file.
CDUP	Change to parent of current working directory.
CWD	Change working directory.
DELE	Delete a file.
EPSV	Prepare for server-to-server transfer.
EPRT	Specify data connection port.
FEAT	List extra features that are not defined in RFC 959.
HELP	Give help information.
LIST	Give list files in a directory (“dir#-ea”).
LPSV	Prepare for server-to-server transfer.
LPRT	Specify data connection port.
MLSD	List contents of directory in a machine-processable form.
MLST	Show a pathname in a machine-processable form.
MKD	Make a directory.
MDTM	Show last modification time of file.
MODE	Specify data transfer mode.
NLST	Give name list of files in directory.
NOOP	Do nothing.
OPTS	Define persistent options for a given command.
PASS	Specify password.
PASV	Prepare for server-to-server transfer.
PORT	Specify data connection port.
PWD	Print the current working directory.
QUIT	Terminate session.
REST	Restart incomplete transfer.
RETR	Retrieve a file.
RMD	Remove a directory.
RNFR	Specify rename-from file name.
RNTO	Specify rename-to file name.
SITE	Non-standard commands (see next section).
SIZE	Return size of file.
STAT	Return status of server.
STOR	Store a file.
STOU	Store a file with a unique name.

Table 7-3. FTP Requests

Request	Description
STRU	Specify data transfer structure.
SYST	Show operating system type of server system.
TYPE	Specify data transfer type.
USER	Specify user name.
XCUP	Change to parent of current working directory (deprecated).
XCWD	Change working directory (deprecated).
XMKD	Make a directory (deprecated).
XPWD	Print the current working directory (deprecated).
XRMD	Remove a directory (deprecated).

The following non-standard or UNIX specific commands are supported by the SITE request.

Table 7-4. UNIX Requests

Request	Description
CHMOD	Change mode of a file. Example: `SITE CHMOD 755 filename`
HELP	Give help information.
IDLE	Set idle-timer. Example: `SITE IDLE 60`
UMASK	Change umask. Example: `SITE UMASK 002`

The following FTP requests (as specified in RFC 959) are recognized, but are not implemented: ACCT, SMNT, and REIN. MDTM and SIZE are not specified in RFC 959, but will appear in the next updated FTP RFC.

The `ftpd` server will abort an active file transfer only when the ABOR command is preceded by a Telnet Interrupt Process (IP) signal and a Telnet Synch signal in the command Telnet stream, as described in Internet
RFC 959. If a STAT command is received during a data transfer, preceded by a Telnet IP and Synch, transfer status will be returned.

User Authentication

`ftpd` authenticates users according to five rules.

1. The login name must be in the password data base, `/dd/sys/password`, and not have a null password. In this case, a password must be provided by the client before any file operations may be performed.
2. If a `/dd/sys/ftpusers` file exists, the login name must be allowed based on the information in the `ftpusers` file.
3. The user must have a standard (`shell` or `mshell`). If the user's shell field in the password database is empty, the shell is assumed to be `mshell`.

4. If directed by the file `/dd/sys/ftpchroot` the session's root directory will be changed by `chroot()` to the directory specified in the `ftpd.conf` chroot directive (if set), or to the home directory of the user. However, the user must still supply a password. This feature is intended as a compromise between a fully anonymous account and a fully privileged account. The account should also be set up as for an anonymous account.
5. If the user name is "anonymous"' or "ftp", an anonymous FTP account must be present in the password file (user "ftp"). In this case the user is allowed to log in by specifying any password. (By convention, an email address for the user should be used as the password.)

   The server performs a `chroot()` to the directory specified in the `ftpd.conf` chroot directive (if set), the `-a <anondir>` directory (if set), or to the home directory of the "ftp" user. The server then performs a `chdir()` to the directory specified in the `ftpd.conf` homedir directive (if set), otherwise to "/".

   If other restrictions are required (such as disabling of certain commands and the setting of a specific umask), then appropriate entries in `ftpd.conf` are required.

   If the first character of the password supplied by an anonymous user is "-", then the verbose messages displayed at login and upon a CWD command are suppressed.

OS-9 does not support `chroot()`. The FTP server uses a virtual `chroot` that is not as secure. Anonymous logins should not be used in an insecure environment.

Display File Escape Sequences

When ftpd displays various files back to the client (such as `/dd/sys/ftpwelcome` and `/dd/sys/motd`), various escape strings are replaced with information pertinent to the current connection.

The supported escape strings are shown below:

`%c`	Class name.
`%C`	Current working directory.
`%E`	Email address given with `-e`.
`%L`	Local hostname.
`%M`	Maximum number of users for this class. Displays "unlimited" if there's no limit.
`%N`	Current number of users for this class.
`%R`	Remote hostname.
`%s`	If the result of the most recent `%M` or `%N` was not "1", print an "s".
`%S`	If the result of the most recent `%M` or `%N` was not "1", print an "S".
`%T`	Current time.
`%U`	User name.
`%%`	A `%` character.

FILES

`/dd/sys/ftpchroot`	List of normal users who should use `chroot()`.
`/dd/sys/ftpd.conf`	Configure file conversions and other settings.
`/dd/sys/ftpusers`	List of unwelcome or restricted users.
`/dd/sys/ftpwelcome`	Welcome notice before login.
`/dd/sys/motd`	Welcome notice after login.
`/dd/sys/nologin`	If it exists, displayed and access is refused.
`/dd/temp/ftpd.pids`	CLASS State file of logged-in processes for the ftpd class CLASS.

ftpusers

The `ftpusers` file provides user access control for `ftpd` by defining which users may login. If the file does not exist, the `ftpuser` validation is not performed.

The "\" symbol is the escape character; it can be used to escape the meaning of the comment character, or if it is the last character on a line, extends a configuration directive across multiple lines. The "#" symbol is the comment character and all characters from it to the end of line are ignored (unless it is escaped with the escape character).

The syntax of each line is shown below:

```
<userglob>[:<groupid>][@<host>] [<directive> [<class>]]
```

`<userglob>`
: Matched against the user name, using `fnmatch()` glob matching (such as "f*").

`<groupid>`
: Matched against the group `id`, of which the user is a member.

`<host>`
: Either a CIDR address (refer to `inet_net_pton`) to match against the remote address (such as "1.2.3.4/24"), or a glob to match against the remote hostname (such as "*.netbsd.org").

`<directive>`
: If the directive is "allow" or "yes", the user is allowed access. If it is "deny" or "no", or the directive is not given, the user is denied access.

<class>
Defines the class to use in ftpd.conf.

If class is not given, it defaults to one of the following options:

chroot	if there is a match in /dd/sys/ftpchroot for the user
guest	if the user name is “anonymous” or “ftp”
real	if neither of the above is true

No further comparisons are attempted after the first successful match. If no match is found, the user is granted access.

If a user requests a guest login, the ftpd server checks to see that both “anonymous” and “ftp” have access, so if you deny all users by default, you will need to add both “anonymous allow” and “ftp allow” to /dd/sys/ftpusers in order to allow guest logins. An example file is installed on the host system in the following location:

/mwos/os9000/src/sys/ftpusers.example

ftpchroot

The file /dd/sys/ftpchroot is used to determine which users will have their session's root directory changed (using chroot()), either to the directory specified in the ftpd.conf chroot directive (if set), or to the home directory of the user. If the file does not exist, the root directory change is not performed.

The syntax is similar to ftpusers, except that the class argument is ignored. If there is a positive match, the session's root directory is changed. No further comparisons are attempted after the first successful match. An example file is installed on the host system in the following location:

mwos/os9000/src/sys/ftpchroot

ftpd.conf

The ftpd.conf file specifies various configuration options for ftpd that apply once a user has authenticated their connection. ftpd.conf consists of a series of lines, each of which may contain a configuration directive, a comment, or a blank line. Directives that appear later in the file override settings by previous directives. This allows wildcard entries to define defaults and have class-specific overrides.

A directive line has the following format: command class [arguments]

A “\” symbol is the escape character; it can be used to escape the meaning of the comment character, or if it is the last character on a line, extends a configuration directive across multiple lines. A “#” symbol is the comment character, and all characters from it to the end of line are ignored (unless it is escaped with the escape character).

Each authenticated user is a member of a <class>, which is determined by ftpusers. A <class> is used to determine which ftpd.conf entries apply to the user. The following special classes exist when parsing entries in ftpd.conf:

all	Matches any <class>.
none	Matches no <class>.

Each class has a type, which may be one of the following options:

GUEST	Guests (as per the "anonymous" and "ftp" logins). A chroot() is performed after login.
CHROOT	Users who use chroot() (as per ftpchroot). A chroot() is performed after login.
REAL	Normal users.

The ftpd STAT command will return the class settings for the current user as defined by ftpd.conf. Each configuration line may be one of the following options:

advertise <class> <host>

Set the address to advertise in the response to the PASV and LPSV commands to the address for host (which may be either a host name or IP address).

This may be useful in some firewall configurations, although many ftp clients may not work if the address being advertised is different to the address to which they have connected. If <class> is none or no argument is given, disable this option.

checkportcmd <class> [<off>]

Check the PORT command for validity.

The PORT command will fail if the IP address specified does not match the FTP command connection, or if the remote TCP port number is less than IPPORT_RESERVED. It is strongly encouraged that this option be used, espcially for sites concerned with potential security problems with FTP bounce attacks. If class is none or off is given, disable this feature, otherwise enable it.

chroot <class> [<pathformat>]

If <pathformat> is not given or class is none, use the default behavior. Otherwise, <pathformat> is parsed to create a directory to create as the root directory with chroot() into upon login. pathformat can contain the following escape strings:

%c	class name
%d	home directory of user
%u	user name
%%	a % character

The default root directory is:

CHROOT:	This is the user's home directory.
GUEST:	If -a <anondir> is given, use it; otherwise, use the home directory of the ftp user.
REAL:	By default, no chroot() is performed.

classtype class type

Set the class type of class to type.

`conversion class suffix [type disable command]`

Define an automatic in-line file conversion. If a file to retrieve ends in suffix, and a real file (without suffix) exists, then the output of command is returned instead of the contents of the file.

`suffix`

The suffix to initiate the conversion.

`type`

A list of valid filetypes for the conversion. Valid types are: "f" (file) and "d" (directory).

`disable`

The name of file that will prevent conversion if it exists. A file name of "." will prevent this disabling action. (The conversion is always permitted.)

`command`

The command to run for the conversion. The first word should be the full path name of the command, as `execv()` is used to execute the command. All instances of the word `%s` in command are replaced with the requested file (without suffix).

Conversion directives specified later in the file override earlier conversions with the same suffix.

Example: `conversion all .tar df .notar /dd/CMDS/tar -cf - %s`

This allows an ftp command to remotely execute a tar command. For instance, to retreive directory "FOO" and all of its contents, type the following at the command line:

```
ftp> bin
ftp> get FOO.tar
```

`display class [file]`

If file is not given or class is `none`, disable this. Otherwise, each time the user enters a new directory, check if file exists. If it does, display its contents to the user. Escape sequences are supported; refer to Display file escape sequences in `ftpd` for more information.

`homedir class [<pathformat>]`

If `pathformat` is not given or class is `none`, use the default behavior. Otherwise, `pathformat` is parsed to create a directory to change into upon login, and to use as the home directory of the user for tilde expansion in pathnames. `pathformat` is parsed per the `chroot` directive. The default home directory is the home directory of the user fo `REAL` users, and "/" for `GUEST` and `CHROOT` users.

`limit class count [<file>]`

Limit the maximum number of concurrent connections for class to count, with "0" meaning unlimited connections.

User limits are currently not implemented.

`maxfilesize class size`
Set the maximum size of an uploaded file to size. If class is `none` or no argument is given, disable this.

`maxtimeout class time`
Set the maximum timeout period that a client may request, defaulting to two hours. This cannot be less than 30 seconds, or the value for timeout. If class is `none` or time is not specified, set to default of 2 hours.

`modify class [<off>]`
If class is `none` or off is given, disable the following commands: `CHMOD`, `DELE`, `MKD`, `RMD`, `RNFR`, and `UMASK`. Otherwise, enable them.

`motd class [<file>]`
If file is not given or class is `none`, disable this. Otherwise, use file as the message of the day file to display after login. Escape sequences are supported. If file is a relative path, it will be searched for in `/dd/sys` (which can be overridden with `-c confdir`).

`notify class [<fileglob>]`
If `fileglob` is not given or class is `none`, disable this. Otherwise, each time the user enters a new directory, notify the user of any files matching `fileglob`.

`passive class [<off>]`
If class is `none` or off is given, disallow passive (`PASV`, `LPSV`, and `EPSV`) connections. Otherwise, enable them.

`portrange class min max`
Set the range of port number which will be used for the passive data port. `max` must be greater than `min`, and both numbers must be be between `IPPORT_RESERVED` (1024) and 65535. If class is `none` or no arguments are given, disable this.

`rateput class rate`
Set the maximum put (`STOR`) transfer rate throttle for class to rate bytes per second, which is parsed as per rateget rate. If class is `none` or no arguments are given, disable this.

`sanenames class [<off>]`
If class is `none` or off is given, allow uploaded file names to contain any characters valid for a file name. Otherwise, only permit file names which do not start with a "." and only comprise of characters from the set `[_A-Za-z0-9]`.

`template class [<refclass>]`
Define `refclass` as the "template" for class; any reference to refclass in following directives will also apply to members of class. This is useful to define a template class so that other classes which are to share common attributes can be easily defined without unnecessary duplication. There can be only one template defined at a time. If `refclass` is not given, disable the template for class.

`timeout class time`
: Set the inactivity timeout period. (The default is fifteen minutes). This cannot be less than 30 seconds, or greater than the value for maxtimeout. If class is `none` or time is not specified, set to the default of 15 minutes.

`umask class umaskval`
: Set the umask to umaskval. If class is `none` or umaskval is not specified, set to the default of 027.

`upload class [<off>]`
: If class is `none` or off is given, disable the following commands: `APPE`, `STOR`, and `STOU`, as well as the modify commands: `CHMOD`, `DELE`, `MKD`, `RMD`, `RNFR`, and `UMASK`. Otherwise, enable them.

Defaults

The following defaults are used:

```
checkportcmd  all
classtype     chroot CHROOT
classtype     guest  GUEST
classtype     real   REAL
display       none
limit         all    -1     # unlimited connections
maxtimeout    all    7200   # 2 hours
modify        all
motd          all    motd
notify        none
passive       all
timeout       all    900    # 15 minutes
umask         all    027
upload        all
modify        guest  off
umask         guest  0707
```

An example file is installed on the host system in the following location:

`mwos/os9000/src/sys/frpd.conf.example`

nologin

`nologin` displays a message that an account is not available and exits non-zero. It is intended as a replacement shell field for accounts that have been disabled.

See Also

`ftp`
`ftpdc`

ftpdc
FTP Server Connection Handler

Syntax

```
ftpdc [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Description

`ftpdc` is the incoming communications handler for `ftp`. `ftpd` and `inetd` can fork `ftpdc`.

Do not run this from the command line. Only `ftpd` and `inetd` can invoke this utility.

See Also

`ftp`

`ftpd`

hostname

Display or Set Internet Name of Host

Syntax

```
hostname [<name>]
```

Options

`-?`

Displays the description, options, and command syntax for `hostname`.

Description

`hostname` prints or sets the string returned by the socket library `gethostname()` function. By default, `gethostname()` returns the `net_name` string appearing in the `inetdb` data module. Use `hostname` to override the default. The length of the string is limited to the length of the current string in the `inetdb` data module.

idbdump
Display Internet Database Entries

Syntax

```
idbdump [<opts>] [<inetdb_file>] [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
: Display the description, options, and command syntax for `idbdump`.

`-d[=]<file>`
: Only display `<file>` entries. Allowable files are `host.conf`, `hosts`, `hosts.equiv`, `inetd.conf`, `networks`, `protocols`, `resolv.conf`, `services`, `interfaces.conf`, `hostname`, `routes.conf`, or `rpc`.

`-m`
: Use the `inetdb` data module in memory. (OS-9 resident systems only)

Description

`idbdump` displays a formatted listing of the entries in the internet database. If options are not specified, `idbdump` displays all entries in the database. Specific options display the appropriate type of database entries.

By default, `idbdump` looks for the internet database as the file `inetdb` in the current data directory. This default can be overridden by a command line path list to the file, or by the `-m` option. This command is also available under the Windows 95/NT operating systems.

Entries from the `routes.conf` file and the `interfaces.conf` file are stored in the `inetdb2` module. To see these entries, specify the module name at the command line:

```
$ idbdump inetbd2
```

See Also

`idbgen`

idbgen
Generate Network Database Module(s)

Syntax

```
idbgen [<opts>] [<filename>] [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
Display the description, options, and command syntax for `idbgen`.

`-d=<path>`
Specify the directory to find the network database files (default is current directory). This option can be repeated to search multiple directories.

`-r=<num>`
Set the module revision to `<num>`.

`-to[=]<target>`
Specify the target operating system. (The default is `OS9000`.)

Table 7-5. Target Operating System <target>

<target>	Operating System
`osk`	OS-9 for Motorola 68000 family processors
`os9000` or `osk` (default)	OS-9

`-tp=<target>`
Specify the target processor and options. (Default is PPC on cross-hosted machines.)

When using `-to=<target>` or `-tp=<target>`, `idbgen` does not recognize the target options in lowercase. For example, `osk`, is not recognized; `OSK` is.

The following table lists target processors and their options.

Table 7-6. Target Processor and Options

<target>	Processor
`68k` or `68000`	Motorola 68000/68010/68070
`CPU32`	Motorola 68300 family
`020` or `68020`	Motorola 68020/68030/68040
`040` or `68040`	Motorola 68040
`386` or `80386`	Intel 80386/80486/Pentium
`PPC` (default)	Generic PowerPC

Table 7-6. Target Processor and Options (Continued)

<target>	Processor
403	PPC 403
601	MPC 601
603	MPC 603
ARM	Generic ARM processor
ARMV3	ARM V3 processor
ARMV4	ARM V4 processor

`-x`
Place the modules in the execution directory (only for OS-9 for 68K/OS-9 resident systems).

`-z[=]<file>`
Read additional command line arguments from `<file>`. (The default is standard input.)

Description

`idbgen` generates the OS-9 data modules `inetdb` and `inetdb2` from the network database files: `host.conf`, `hosts`, `hosts.equiv`, `inetd.conf`, `networks`, `protocols`, `resolv.conf`, `interfaces.conf`, `routes.conf`, `services` and `rpc`. Any time a change is made to any of these files, `idbgen` must be used to generate new data modules.

By default, the output internet database modules are named `inetdb` and `inetdb2` and are placed in the current directory. An optional command line file name can override this default by specifying the pathlist to the output files.

By default, `idbgen` looks for the network database files in the current directory. However, the `-d` option can be used to specify the directories containing the files.

Refer to *Appendix A, Configuring LAN Communications* for definitions and descriptions of the configuration files.

This command is also available under the Windows 95/NT operating systems.

See Also

`idbdump`

ifconfig
Configure Network Interface

Syntax

```
ifconfig [-L] <interface> [af] [<opts>]
ifconfig -a [-d] [-u] [af]
ifconfig -l [-d] [-u]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`<IP Address <dest_addr | broadcast> [alias]`
Change IP address and broardcast address. (Add an alias.)

`netmask <subnet mask>`
Change subnet mask.

`up`
Mark interface as “up”.

`down`
Mark interface as “down”.

`stop`
Mark an interface to stop.

`start`
Mark an interface to start.

`delete`
Delete IP address or alias.

`binding <device>`
Bind device to interface name.

`iff_nopointopoint`
Override driver `iff_pointopoint` flag.

`iff_nobroadcast`
Override driver `iff_broadcast` flag.

`iff_nomulticast`
Override driver `iff_multicast` flag.

`af address_family`
Current supported families are IPv4 address family `inet` and IPv6 address family `inet6`.

`-alias`
Remove the specified network address `alias` from the interface.

`anycast`
Set the IPv6 anycast address bit.

`-anycast`
Clear the IPv6 anycast address bit.

`tentative`
Set the IPv6 tentative address bit.

`-tentative`
Clear the IPv6 tentative address bit.

`pltime n`
Set preferred lifetime for the address.

`vltime n`
Set valid lifetime for the IPv6 address.

`prefixlen n`
Set the prefix length at the IPv6 address. The effect is similar to netmask.

`broadcast <addr>`
Change broadcast address. (`inet` only)

`metric n`
Set the routing metric of the interface to n.This is used by the routing protocol. Higher metrics makes a route less favorable. The default metric is 0.

`mtu n`
Set the maximum transmission unit of the interface.

`-a flag`
Show all interfaces in the system.

`-d flag`
Only list interfaces that are down.

`-u flag`
Only list interfaces that are up.

`-l flag`
List all available interfaces in the system with no other additional information.

`-L flag`
Enable displaying lifetime information of the address.

`unbind`
Remove interface binding.

`rebind`

Bind stopped interface.

Description

`ifconfig` assigns an address to a network interface and/or configures network interface settings. It can also dynamically add an interface to the system.

`address`
: The address is either a host name present in the host name data base, or an Internet address expressed in the standard Internet dot notation.

`alias`
: Establish an additional network address for this interface. Useful when changing network numbers and accepting packets addressed to the old interface.

`binding <device>`
: Dynamically add a new interface to the system. The device parameter specifies the device/protocol descriptors(s) which will be opened and associated with the specified interface name (for example, `/spe30/enet`).

`stop`
: Mark an interface to stop. This causes the hardware device associated with the specified interface to be closed. The interface remains in SPIP's internal interface table and may be restarted using the start command.

`start`
: Mark an intreface to start. This is used to enable an interface after using `stop`.

`unbind`
: This is similar to `stop`, except that it also causes all information about the interface to be removed from SPIP's interface table.

`rebind`
: This is used with a stopped interface to associate it with a different hardware driver stack.

`broadcast <addr>`
: Specify the address to use to represent broadcasts to the network. The default broadcast address has a host part of all 1s.

`delete`
: Remove the network address specified. Used when an alias is incorrectly specified, or no longer needed.

`dest_address`
: Specify the address of the peer on the other end of a point-to-point link.

`down`
: Mark an interface down. When an interface is marked down, the system does not attempt to transmit messages through that interface. Incoming packets on

the interface are discarded. This action does not automatically disable routes using the interface.

`up`
Mark an interface up. Used to enable an interface after an ifconfig down. It happens automatically when setting the first address on an interface.

`iff_nobroadcast`
Drivers that are capable of sending broadcasts will automatically set the `iff_broadcast` flag. If you do not want to send broadcasts from a particular ethernet card this option could be used to prevent the flag from being set. This option requires the `binding` option.

`iff_nopointopoint`
Override the `iff_pointopoint` flag that is set automatically by the slip and ppp drivers. This option requires the `binding` option.

`iff_nomulticast`
Override the `iff_multicast` flag set by multicast capable interfaces. This option is useful if you do not wish to send or receive any multicast packets and are on a network with a high volume of multicast traffic.

`interface`
The interface parameter is a string of the form `nameX` where X is the integer unit number (for example, `enet0`). The name is used internally by IP and is not the name of a file or module.

`netmask <mask>`
Specify how much of the address to reserve for subdividing networks into sub-networks. The mask includes the network part of the local address and the subnet part, which is taken from the host field of the address.

The mask can be specified as a single hexadecimal number with a leading `0x`, with a dot-notation Internet address, or with a pseudo-network name listed in the `inetdb` network table "networks". The mask contains 1s for the bit positions in the 32-bit address which are to be used for the network and subnet parts, and 0s for the host part. The mask contains at least the standard network portion, and the subnet field is contiguous with the network portion.


You must specify an IP address to configure netmask.

`metric n`
Set the routing metric of the interface to n, default 0. The routing metric is used by the routing protocol . It is counted as addition hops to the destination network or host. Higher metric makes a route less favorable.

`prefixlen len`
Specify that len bits are reserved for subdividing networks into sub-networks, defulat 64. The len must be integer between 0 to 128. It is almost always 64 under the current IPv6 assignment rule. (Inet6 only)

`tentative`

An tentative address means it has not passed Duplicate Address Detection (DAD) yet. Such an address has not yet been assigned to the interface in a true sense. The address is associated with the interface only for the purposes of performing DAD. (Inet6 only)

`anycast`
: An identifier for a set of interfaces. A packet sent to an anycast address is routed to the "nearest" interface having that address, depending on the distance of a routing path.

  A routing path between certain nodes may vary with changes in the network configuration. Therefore, a packet using anycast address may not be sent to the same node when any changes on the routing path occur. As a result, a point-to-point communication in which its end point is specified by an anycast address does not work well, especially with connection-oriented protocols. (Inet6 only)

`ifconfig` displays the current configuration for a network interface when no options are supplied. Only the super-user can modify the configuration of a network interface.

Examples

Create a new interface `enet0`:

```
ifconfig enet0 172.16.1.1 binding /spe30/enet
```

Since no netmask or broadcast address was specified, the appropriate class A, B, or C addresses will be used. In this case, the netmask `255.255.0.0` and a broadcast address of `172.16.255.255` will be used. To override the default netmask:

```
ifconfig enet0 172.16.1.1 netmask 255.255.255.0 binding /spe30/enet
```

This will also change the broadcast address to `172.16.1.255`.

Create a new interface, but disable its multicast capability:

```
ifconfig enet0 odin iff_nomulticast binding /spe30/enet
```

This example also uses a machine name instead of an IP address. For this to work, the name must either be resolvable in your local `inetdb` or by a DNS name server reachable by an interface other than the one being added.

Change the address of an existing interface:

```
ifconfig enet0 loki
```

If the hardware driver supports it, you can add an alias to an already existing interface:

```
ifconfig enet0 thor alias
```

Remove the alias:

```
ifconfig enet0 thor -alias
```

Create the point-to-point interface `ppp0` without any address information:

```
ifconfig ppp0 binding /ipcp0
```

Set the local and destination address of a point-to-point link:

```
ifconfig slip0 192.168.8.175 192.168.8.174
```

Show the address information for interface `enet0`:

```
ifconfig enet0
```

The output is similar to the following:

```
enet0: flags=8003<UP,BROADCAST,MULTICAST> mtu 1500
inet 172.16.1.1 netmask 0xffff0000 broadcast 172.16.255.255
```

Show all interfaces:

```
ifconfig -a
```

List IPv6 on the interface:

```
ifconfig enet0 inet6
```

Add IPv6 address to interface:

```
ifconfig enet0 inet6 fec0::2436:1
```

Delete IPv6 address from the interface:

```
ifconfig enet0 inet6 fec0::24361:1 delete
```

Set anycast bit:

```
ifconfig enet0 inet6 fec0::2436:1 anycast
```

Clear anycast bit

```
ifconfig enet0 inet6 fec0::2436:1 -anycast
```

Set valid lifetime for the address:

```
ifconfig enet0 inet6 inet6 fec0::2436:1 vltime 1100000000
```

Show the valid lifetime for the address:

```
ifconfig -L enet0 inet6
```

Set routing metric of the interface:

```
ifconfig enet0 metric 50
```

Mark an interface down:

```
Ifconfig enet0 down
```

Mark an interface stop:

```
Ifconfig enet0 stop
```

See Also

netstat

inetd
Internet Master Daemon

Syntax

```
inetd [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
Display the description, options, and command syntax for `inetd`.

`-i[...]`
Internal `inetd` options.

Description

`inetd` is a master internet daemon process, that can be configured to listen for incoming TCP or UDP connections on up to 25 separate ports. When `inetd` detects an incoming connection, it forks a child process (for example `telnetdc` or `ftpdc`) to handle the connection.

The `inetd.conf` file specifies the configuration for the `inetd` process. Each non-comment or non-white space line in this file specifies a service which `inetd` provides. Each line has the following syntax:

```
<ServiceName> <SocketType> <Protocol> <Flags> <User> <ServerPathname>
[<Args>]
```

`ServiceName`
Specify the internet service to be provided. This service name must also be specified in the `<services>` section of the `inetdb` data module. The port number for the specified service is referenced through the `<services>` section of `inetdb`.

`SocketType`
Specify whether the socket type is a `stream` (for TCP services) or `dgram` (for UDP services).

`Protocol`
Specify the protocol type to use for this service. This protocol name must also be specified in the `<protocols>` section of the `inetdb` data module.

`Flags`
Specify special actions `inetd` to take when processing incoming connections for this service. `Wait` is the only valid `Flags` action.

`User`

Specify the group/user under which the forked child process should run—either in the format `<group>.<user>` (such as "12.136"), the keyword `root` (resolving to user "0.0"), or the keyword `nobody` (resolving to user "1.0").

`ServerPathname`

Specify the child process to fork when an incoming connection for the specified service is detected. This may be just the program name if it is in memory or in the current path, or it may be a full path name to the program file. Eight special services, `echo` (`dgram` & `stream`), `discard` (`dgram` & `stream`), `daytime` (`dgram` & `stream`), and `chargen` (`dgram` & `stream`) are handled internally by `inetd` by specifying the server path name `internal`.

`Args`

Specify additional arguments to use when forking the child process.

When `inetd` forks a child process to handle an incoming service request, the child is forked with three paths: `stdin` and `stdout` are the connected socket paths and `stderr` is the `stderr` path with which `inetd` was forked. If the service was on a `udp/dgram` socket, a `connect()` is performed before forking the child. If the service was on a `tcp/stream` socket, an `accept()` is performed before forking the child.

Super user account is required to run `inetd`.

ipstart
Initialize IP Stack

Syntax

```
ipstart
```

Options

None.

Description

`ipstart` initializes the IP stack at startup. Run this utility in the foreground.

mbdump

Display System mbuf statistics

Syntax

```
mbdump [<opts>]
```

Options

`-a`
: Display mbuf pool allocation table.

`-m`
: Display current mbuf pool bit map.

`-?`
: Display description, options, and command syntax for mbdump.

Description

`mbdump` displays the current state of the system mbuf pool. This information can be used to determine whether or not the size of the mbuf pool should be altered.

By default, `mbdump` displays the size of the pool, where it is located, how much is in use, and the minimum allocation size.

ndbmod

Dynamically Update the Internet Data Module

Syntax

```
ndbmod <[opts]>
```

Subcommand-Specific Options

`-?`

Display the description, options, and command syntax for `ndbmod`.

`hostname <hostname>`

Set hostname of the system.

`interface <options>`

Add or delete an interface.

Options are defined as follows:

```
add <intname> [address inet <addr>[netmask <addr>] \
[broadcast|destaddr <addr>]]
[inet6 <ipv6addr/prefixlength>] binding <device>
[mtu <mtu>] \ [metric <metric>] [up|down]
[iff_broadcast][iff_pointopoint][iff_nomulticast]
[iff_nobroadcast] [iff_nopointopoint]del <intname>
```

The `[iff_broadcast]`and `[iff_broadcast]` flags are for compatibility with older version of LAN Communications. The preferred method is to let the driver set these flags and to use the `[iff_noXXX]` flags to override the driver defaults.

`resolve <arglist>`

Add DNS search and resolver information.

Options are defined as follows:

```
<domainname [server <addr>]* [search <srch-1> <srch-2> ...]
```

`host <options>`

Add or delete a host entry.

Options are defined as follows:

```
add <addr> <name> [<alias-1> <alias-2> ...]
del <name|alias>
```

`route <options>`

Add IPv4 or IPv6 static route.

Options are defined as follows:

```
add <dst-route> <gateway> [<netmask>]
add [default]<gateway>[<netmask>]
```

`create <modname> <num-files> <size-1> <size-2>... <size-n>`

Create a dynamic Internet data module having `<num-files>` files.

`<size-1>` indicates bytes to reserve for file 1.

`<size-2>` indicates bytes to reserve for file 2.

All file sizes must be specified; 0 indicates no space is reserved for that file.

`<modname>` is one of `inetdb2`, `inetdb3`, or `inetdb4`.

File Numbers

1	`/dd/etc/hosts` (aprox. 25 bytes per host)
2	`/dd/etc/hosts.equiv` (not used)
3	`/dd/etc/networks` (aprox. 40 bytes per network)
4	`/dd/etc/protocols` (aprox. 25 bytes per protocol)
5	`/dd/etc/services` (aprox. 25 bytes per service)
6	`/dd/etc/inetd.conf` (aprox. 50 bytes per entry)
7	`/dd/etc/resolv.conf` (aprox. 100 bytes)
8	host config (not used)
9	host interfaces (aprox 300 bytes per interface)
10	hostname (>= length of hostname + 1, recommended 65)
11	static routes (aprox. 105 bytes per entry)
12-32	reserved

Description

The `ndbmod` utility allows the user to add, remove, or modify information stored in the `inetdb` data module such as host names, IP addresses, and DNS Server fields dynamically. It also enables the creation of expansion `inetdb` modules in the event that the `inetdb` data module is full or in ROM.

These expansion modules can also hold additional host-specific information such as the machines `hostname`, and interface settings such as IP address and network masks.

`ndbmod` must be run before the IP stack is initialized.

Example 1

To create an alternate `inetdb` module called `inetdb2` with 100 bytes for new host information, 0 for `hosts.equiv`, 30 for new network entries:

```
ndbmod create inetdb2 12 100 0 30 30 0 50 200 100 0 150 64
```

Example 2

To set the hostname for a system name alpha:

```
ndbmod hostname alpha
```

Example 3

To add a host entry for system beta with alias of beta1 and beta2:

```
ndbmod host add 192.1.1.3 beta beta1 beta2
ndbmod host add feco::1111:1 alpha alpha1 alpha2
```

Example 4

To remove the host entry for beta:

```
ndbmod host del beta
```

Example 5

To add a SLIP interface to an `inetdb2` data module:

```
ndbmod interface add  slip0 address 192.1.1.1 \ destaddr 192.1.1.2 binding
/spsl0
```

Example 6

To delete a SLIP interface from the `inetdb2` data module:

```
ndbmod interface del slip0
```

Example 7

To add an ethernet interface and disable the multicast capabilities:

```
ndbmod interface add enet0 address 172.16.1.1 \ iff_nomulticast binding
/spe30/enet
```

Example 8

To add an Ethernet interface wiht an IPv4 address and netmask and IPv6 addresses with prefix length:

```
ndbmod interface add enet0 address inet 172.16.1.1 netmask 255.255.255.255
inet6 fec0::1111:1/64 binding /spe30/enet
```

Example 9

To add an Ethernet interface with IPv6 addresses:

```
ndbmod interface add enet0 address inet6 fec0::1111:1 binding /spe30/enet
```

Example 10

To add a new DNS resolve entry to the `inetdb2` data module:

```
ndbmod resolve testdns.com server 192.1.1.3 search
testdns.com misc.org
```

Example 11

To add a new DNS resolve entry with an IPv6 server:

```
ndbmod resolve testdns.com server fec0::1111:1 search ipv6.dm1.radisys.com
```

Example 12

To add a static network route to the `inetdb2` data module:

```
ndbmod route add network 192.2.2.0 192.1.1.2
```

See Also

`idbgen`

netstat
Show Network Status

Syntax

```
netstat [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-A`
: With the default display, show the address of any protocol control blocks associated with sockets; used for debugging.

`-a`
: With the default display, show the state of all sockets; normally sockets used by server processes are not shown.

`-b`
: Show interface packet statistics.

`-d`
: With either interface display (option `-i` or an interval), show the number of dropped packets.

`-f <address_family>`
: Limit statistics or address control block reports to those of the specified `<address_family>`. The following address families are recognized: `inet`, for AF_INET; `inet6`, for AF_INET6.

`-g`
: Show information related to multicast (group address) routing. By default, show the IP Multicast virtual-interface and routing tables. If the `-s` option is also present, show multicast routing statistics.

`-I <interface>`
: Show information for all interfaces. This is used with a `<wait>` interval. If the `-f <address_family>` option (with the `-s` option) or the `-p <protocol>` option is present, show per-interface statistics on the interface for the specfied `<address_family>` or `<protocol>`, respectively.

`-i`
: Show the state of interfaces which have been auto-configured (interfaces statically configured into a system, but not located at boot time are not shown). If the `-a` option is also present, multicast addresses currently in use are shown for each Ethernet interface and for each IP interface address.

  Multicast addresses are shown on separate lines following the interface address with which they are associated. If the

-f <address_family> option (with the -s option) or the -p <protocol> option is present, show per-interface statistics on all interfaces for the specfied <address_family> or <protocol>, respectively.

-L
Do not show link-level routes.

-M
Extract values associated with the name list from the specified core instead of the default </dev/kmem>.

-m
Show statistics recorded by the memory management routines. (The network manages a private pool of memory buffers.)

-N
Extract the name list from the specified system instead of the default /netbsd.

-n
Show network addresses as numbers (normally netstat interprets addresses and attempts to display them symbolically). This option may be used with any of the display formats.

-P <pcbaddr>
Dump the contents of the protocol control block (PCB) located at the kernel virtual address <pcbaddr>. This address may be obtained using the -A flag. The default protocol is TCP, but may be overridden using the -p flag.

-p <protocol>
Show statistics about protocol, which is either a well-known name for a protocol or an alias for it. A null response typically means that there are no interesting numbers to report. The program complains if the protocol is unknown or if there is no statistics routine for it.

-s
Show per-protocol statistics. If this option is repeated, counters with a value of zero are suppressed.

-r
Show the routing tables. When -s is also present, show routing statistics instead.

-v
Show extra (verbose) detail for the routing tables (-r), or avoid truncation of long addresses.

-w <wait>
Show network interface statistics at intervals of <wait> seconds.

Description

The netstat command symbolically displays the contents of various network-related data structures. There are a number of output formats, depending on the

options for the information presented. The first form of the command displays a list of active sockets for each protocol. The second form presents the contents of one of the other network data structures according to the option selected. Using the third form, with a wait interval specified, netstat will continuously display the information regarding packet traffic on the configured network interfaces. The fourth form displays statistics about the named protocol. The fifth and sixth forms display per interface statistics for the specified protocol or address family.

The default display, for active sockets, shows the local and remote addresses, send and receive queue sizes (in bytes), protocol, and the internal state of the protocol. Address formats are of the form “host.port” or “network.port” if a socket's address specifies a network but no specific host address. When known the host and network addresses are displayed symbolically according to the data bases `/etc/hosts` and `/etc/networks`, respectively. If a symbolic name for an address is unknown, or if the `-n` option is specified, the address is printed numerically, according to the address family.

The interface display provides a table of cumulative statistics regarding packets transferred, errors, and collisions. The network addresses of the interface and the maximum transmission unit (``mtu'') are also displayed.

The routing table display indicates the available routes and their status. Each route consists of a destination host or network and a gateway to use in forwarding packets. The flags field shows a collection of information about the route stored as binary choices.

The individual flags are discussed in more detail in the `route` section.

The mapping between letters and flags is shown in Table 7-7.

Table 7-7. Mapping Between Letters and Flags

1	`RTF_PROTO2`	Protocol specific routing flag
2	`RTF_PROTO1`	Protocol specific routing flag
B	`RTF_BLACKHOLE`	Just discard pkts (during updates)
C	`RTF_CLONING`	Generate new routes on use
D	`RTF_DYNAMIC`	Created dynamically (by redirect)
G	`RTF_GATEWAY`	Requires forwarding by intermediary
H	`RTF_HOST`	Host entry (net otherwise)
L	`RTF_LLINFO`	Valid protocol to link address translation.
M	`RTF_MODIFIED`	Modified dynamically (by redirect)
R	`RTF_REJECT`	Host or net unreachable
S	`RTF_STATIC`	Manually added
U	`RTF_UP`	Route usable
X	`RTF_XRESOLVE`	External daemon translates proto-to-link address
c	`RTF_CLONED`	This is a cloned route.

Direct routes are created for each interface attached to the local host; the gateway field for such entries shows the address of the outgoing interface. The refcnt field

gives the current number of active uses of the route. Connection oriented protocols normally hold on to a single route for the duration of a connection while connectionless protocols obtain a route while sending to the same destination. The use field provides a count of the number of packets sent using that route. The mtu entry shows the mtu associated with that route. This mtu value is used as the basis for the TCP maximum segment size. The 'L' flag appended to the mtu value indicates that the value is locked, and that path mtu discovery is turned off for that route. A "-" indicates that the mtu for this route has not been set, and a default TCP maximum segment size will be used. The interface entry indicates the network interface utilized for the route.

When `netstat` is invoked with the `-w` option and a `<wait>` interval argument, it displays a running count of statistics related to network interfaces. An obsolescent version of this option used a numeric parameter with no option, and is currently supported for backward compatibility. This display consists of a column for the primary interface (the first interface found during autoconfiguration) and a column summarizing information for all interfaces. The primary interface may be replaced with another interface with the `-I` option. The first line of each screen of information contains a summary since the system was last rebooted. Subsequent lines of output show values accumulated over the preceding interval.

ping
Send ICMP ECHO_REQUEST Packets to Host

Syntax

```
ping [<opts>] host
```

IP Functionality

IPv4 addresses only

Options

`-s <packetsize>`
: Number of data bytes to send.

`-?`
: Print help.

Description

`ping` sends an ICMP echo request to a specified host and waits for a reply. With the `-s` option, you can specify the size of data to send to the host. Upon success, `ping` displays the number of bytes received from the host and transmission time.

You can ping broadcast and multicast addresses; however, only the first machine to respond will be printed.

ping6

Send ICMPv6 Packets to Network Hosts

Syntax

```
ping6 [-dfHnNqRtvw] [-a addrtype] [-b bufsiz] [-c count] [-h hoplimit] [-
I interface] [-i wait] [-l preload] [-p pattern] [-P policy] [-S
sourceaddr]
[-s packetsize] [hops...] host
```

IP Functionality

```
IPv6 addresses only
```

Options

`-a addrtype`

Generate ICMPv6 Node Information Node Addresses query, rather than echo-request. addrtype must be a string constructed of the following characters.

`a`	requests all the responder's unicast addresses. If the charater is ommited, only those addresses which belong to the interface which has the responder's address are requests.
`c`	requests responder's IPv4-compatible and IPv4-mapped addresses.
`g`	requests responder's global-scope addresses.
`s`	requests responder's site-local addresses.
`l`	requests responder's link-local addresses.
`A`	requests responder's anycast addresses. Without this character, the responder will only return unicast addresses. The specification does not state how to get the responder's anycast addresses. (experimental option)

`-b bufsiz`

Set socket buffer size.

`-c count`

Stop after sending (and receiving) count `ECHO_RESPONSE` packets.

`-d`

Set the `SO_DEBUG` option on the socket being used.

`-f`

Flood ping. Output packets as fast as they come back or one hundred times per second, whichever is more.

For every ECHO_REQUEST sent a period "." is printed, while for every ECHO_REPLY received a backspace is printed. This provides a rapid display of how many packets are being dropped. Only the super-user may use this option. This can be very hard on a network and should be used with caution.

-H
Specifies to try reverse-lookup of IPv6 addresses. The ping6 command does not try reverse-lookup unless the option is specified.

-h hoplimit
Set the IPv6 hoplimit.

-I interface
Source packets with the given interface address. This flag applies if the ping destination is a multicast address, or link-local/site-local unicast address.

-i wait
Wait seconds between sending each packet. The default is towait for one second between each packet. This option is incompatible with the -f option.

-l preload
If preload is specified, ping6 sends that many packets as fast as possible before falling into its normal mode of behavior. Only the super-user may use this option.

-n
Numeric output only. No attempt will be made to lookup symbolic names for host addresses.

-N
Probe node information multicast group (ff02::2:xxxx:xxxx). host must be string hostname of the target (must not be a numeric IPv6 address). Node information multicast group will be computed based on given host, and will be used as the final destination. Since node information multicast group is a link-local multicast group, destination link needs to be specified by -I option.

-p pattern
You may specify up to 16 pad bytes to fill out the packet you send. This is useful for diagnosing data-dependent problems in a network. For example, -p ff will cause the sent packet to be filled with all ones. -Q flag, ping6 prints out any ICMP error messages caused by its own ECHO_REQUEST messages.

-q
Quiet output. Nothing is displayed except the summary lines at startup time and when finished.

-S sourceaddr
Specify the source address of request packets. The source address must be one of the unicast addresses of the sending node. If the outgoing interface is specified by the -I option as well, sourceaddr needs to be an address assigned to the specified interface.

`-s packetsize`
: Specify the number of data bytes to be sent. The default is 56, which translates into 64 ICMP data bytes when combined with the 8 bytes of ICMP header data. You may need to specify `-b` as well to extend socket buffer size.

`-t`
: Generate ICMPv6 Node Information supported query types query, rather than echo-request. `-s` has no effect if `-t` is specified.

`-v`
: Verbose output. ICMP packets other than `ECHO_RESPONSE` that are received are listed.

`-w`
: Generate ICMPv6 Node Information DNS Name query, rather than echo-request. `-s` has no effect if `-w` is specified.

`-W`
: Same as `-w`, but with old packet format based on 03 draft. This option is present for backward compatibility. `-s` has no effect if `-w` is specified.

`hops`
: IPv6 addresses for intermediate nodes, which will be put into type 0 routing header.

`host`
: IPv6 adddress of the final destination node. When using ping6 for fault isolation, it should first be run on the local host, to verify that the local network interface is up and running. Then, hosts and gateways further and further away should be pinged. Round-trip times and packet loss statistics are computed.

  If duplicate packets are received, they are not included in the packet loss calculation, although the round trip time of these packets is used in calculating the round-trip time statistics. When the specified number of packets have been sent (and received) or if the program is terminated with a SIGINT, a brief summary is displayed, showing the number of packets sent and received, and the minimum, maximum, mean, and standard deviation of the round-trip times.

Description

`ping6` uses the ICMPv6 protocol's mandatory `ICMP6_ECHO_REQUEST` datagram to elicit an `ICMP6_ECHO_REPLY` from a host or gateway. `ICMP6_ECHO_REQUEST` datagrams have an IPv6 header and ICMPv6 header.

This program is intended for use in network testing, measurement and management. Because of the load it can impose on the network, it is unwise to use `ping6` during normal operations or from automated scripts.

See Also

`netstat`

```
ifconfig

ping

routed
```

route
Add/Delete Routes

Syntax

```
route <opts>
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-f`
: Remove all routes (per `flush`). If used in conjunction with the `add`, `change`, `delete` or `get` commands, `route` removes the routes before performing the command.

`-n`
: Bypasses attempts to print host and network names symbolically when reporting actions. (The process of translating between symbolic names and numerical equivalents can be quite time consuming, and may require correct operation of the network; thus it may be expedient to forgo this, especially when attempting to repair networking operations).

`-q`
: Suppress all output.

`-v`
: (verbose) Print additional details.

The `route` utility also provides several commands:

`add`
: Add a route.

  Syntax: `route [-n] <command> [-net | -host] <destination gateway>` where `<destination>` is the destination host or network, and `<gateway>` is the next-hop intermediary by which packets should be routed.

`flush`
: Remove all routes.

  If `flush` is specified, `route` will "flush" the routing tables of all gateway entries. When the address family is specified by any of the `-osi`, `-xns`, `-atalk`, `-inet`, or `-inet6` modifiers, only routes having destinations with addresses in the delineated family will be deleted.

  Syntax: `route [-n] flush [<family>]`

`delete`
: Delete a specific route.

Syntax: `route [-n] <command> [-net | -host] <destination gateway>` where `<destination>` is the destination host or network, and `<gateway>` is the next-hop intermediary by which packets should be routed.

`change`
Chnage aspects of a route (such as its gateway).

Syntax: `route [-n] <command> [-net | -host] <destination gateway>` where `<destination>` is the destination host or network, and `<gateway>` is the next-hop intermediary by which packets should be routed.

`get`
Look up and display the route for a destination.

Syntax: `route [-n] <command> [-net | -host] <destination gateway>` where `<destination>` is the destination host or network, and `<gateway>` is the next-hop intermediary by which packets should be routed.

`show`
Print out the routing table.

Syntax: `route [-n] <command> [-net | -host] <destination gateway>` where `<destination>` is the destination host or network, and `<gateway>` is the next-hop intermediary by which packets should be routed.

`monitor`
Continuously report any changes to the routing information base, routing lookup misses, or suspected network partitionings.

Syntax: `route [-n] monitor`

Description

`route` is a utility used to manually manipulate the network routing tables. It normally is not needed, as a system routing table management daemon such as `routed()`, should tend to this task.

The `route` utility supports a limited number of general options, but a rich command language, enabling the user to specify any arbitrary request that could be delivered via the programmatic interface discussed in `route()`.

Routes to a particular host may be distinguished from those to a network by interpreting the Internet address specified as the destination argument. The optional modifiers `-net` and `-host` force the destination to be interpreted as a network or a host, respectively. Otherwise, if the destination has a "local address part" of INADDR_ANY, or if the destination is the symbolic name of a network, then the route is assumed to be to a network; otherwise, it is presumed to be a route to a host. For example, "128.32" is interpreted as `-host 128.0.0.32`; "128.32.130" is interpreted as `-host 128.32.0.130`; "-net 128.32" is interpreted as `128.32.0.0`; and "-net 128.32.130" is interpreted as `128.32.130.0`.

If the destination is directly reachable via an interface requiring no intermediary system to act as a gateway, the `-interface` modifier should be specified; the

gateway given is the address of this host on the common network, indicating the interface to be used for transmission.

The optional `-netmask` qualifier is intended to achieve the effect of an OSI ESIS redirect with the netmask option, or to manually add subnet routes with netmasks different from that of the implied network interface
(as would otherwise be communicated using the OSPF or ISIS routing protocols). An additional ensuing address parameter is specified (interpreted as a network mask). The implicit network mask generated in the AF_INET case can be overridden by making sure this option follows the destination parameter. `-prefixlen` is also available for similar purpose, in the case of IPv6.

Routes have associated flags which influence operation of the protocols when sending to destinations matched by the routes. These flags may be set (or sometimes cleared) by indicating the following corresponding modifiers:

Table 7-8.

`-cloning`	RTF_CLONING	generate a new route on use
`-xresolve`	RTF_XRESOLVE	emit mesg on use (for external lookup)
`-iface`	~RTF_GATEWAY	destination is directly reachable
`-static`	RTF_STATIC	manually added route
`-nostatic`	~RTF_STATIC	pretend route added by kernel or daemon
`-reject`	RTF_REJECT	emit an ICMP unreachable when matched
`-blackhole`	RTF_BLACKHOLE	silently discard pkts (during updates)
`-proto1`	RTF_PROTO1	set protocol specific routing flag
`-proto2`	RTF_PROTO2	set protocol specific routing flag
`-llinfo`	RTF_LLINFO	validly translate proto addr to link addr

The optional modifiers `-rtt`, `-rttvar`, `-sendpipe`, `-recvpipe`, `-mtu`, `-hopcount`, `-expire`, and `-ssthresh` provide initial values to quantities maintained in the routing entry by transport level protocols, such as TCP or TP4. These may be individually locked by preceding each such modifier to be locked by the `-lock meta-modifier`, or one can specify that all ensuing metrics may be locked by the -lockrest meta-modifier.

In a change or add command where the destination and gateway are not sufficient to specify the route (as in the ISO case where several interfaces may have the same address), the `-ifp` or `-ifa` modifiers may be used to determine the interface or interface address.

All symbolic names specified for a destination or gateway are looked up first as a host name using `gethostbyname()`. If this lookup fails, `gethostbyname()` is then used to interpret the name as that of a network.

`route` uses a routing socket and the new message types RTM_ADD, RTM_DELETE, RTM_GET, and RTM_CHANGE. As such, only the super-user may modify the routing tables.

See Also

netstat

routed

route6d

route6d
RIP Routing Daemon Over IPv6

Syntax

```
route6d [<opts>]
```

IP Functionality

IPv6 addresses

Description

The `route6d` is a routing daemon, which supports RIP over IPv6.

`route6d` requires that routing domain support (`sproute` and `route0`) be loaded onto the system.

Options

`-a`

Enables aging of the statically defined routes. With this option, any statically defined routes will be removed unless corresponding updates arrive as if the routes are received at thestartup of `route6d`.

`-R <routelog>`

This option makes the `route6d` to log the route change (add/delete) to the file `<routelog>`.

`-A <prefix/preflen,if1[,if2...]>`

This option is used for aggregating routes. `<prefix/preflen>` specifies the prefix and the prefix length of the aggregated route. When advertising routes, `route6d` filters specific routes covered by the aggregate, and advertises the aggregated route `<prefix/preflen>`, to the interfaces specified in the comma-separated interface list, `<if1[,if2...]>`. `route6d` creates a static route to `<prefix/preflen>` with RTF_REJECT flag, into the kernel routing table.

`-d`

Enables output of debugging message. This option also instructs `route6d` to run in foreground mode (does not become daemon).

`-D`

Enables extensive output of debugging message. This option also instructs `route6d` to run in foreground mode (does not become daemon).

`-h`

Disables the split horizon processing.

-l

By default, route6d will not exchange site local routes for safety reasons. This is because semantics of site local address space is rather vague (specification is still in being worked), and there is no good way to define site local boundary. With -l option, route6d will exchange site local routes as well. It must not be used on site boundary routers, since -l option assumes that all interfaces are in the same site.

-L <prefix/preflen,if1[,if2...]>
Filter incoming routes from interfaces <if1,[if2...]>. route6d will accept incoming routes that are in <prefix/preflen>. If multiple -L options are specified, any routes that match one of the options is accepted. ::/0 is treated specially as default route, not "any route that has longer prefix length than, or equal to 0". If you would like to accept any route, specify no -L option. For example, with "-L 3ffe::/16,if1 -L ::/0,if1", route6d will accept default route and routes in 6bone test address, but no others.

-N <if1[,if2...]>
Do not listen to, or advertise, route from/to interfaces specified by <if1,[if2...]>.

-O <prefix/preflen,if1[,if2...]>
Restrict route advertisement toward interfaces specified by<if1,[if2...]>. With this option route6d will only advertise routes that matches <prefix/preflen>.

-q
Make route6d in listen-only mode. No advertisement is sent.

-s
Make route6d to advertise the statically defined routes which exist in the kernel routing table when route6d invoked. Announcements obey the regular split horizon rule.

-S
This option is the same as -s option except that no split horizon rule applies.

-T <if1[,if2...]>
Advertise only default route, toward <if1[,if2...]>.

-t <tag>
Attach route tag <tag> to originated route entries. <tag> can be decimal, octal prefixed by 0, or hexadecimal prefixed by 0x.

Upon receipt of signal SIGINT or SIGUSR1, route6d will dump the current internal state into /var/run/route6d_dump.

See Also

route

routed

routed
Dynamic Routing Daemon

Syntax

```
routed [<opts>]
```

IP Functionality

IPv4 addresses only

Options

`-?`
Print help.

`-s`
Force `routed` to send routing updates. This is the default if more than one network interface is configured and IP forwarding has been enabled in `spip`.

`-q`
Do not send any routing updates.

`-h`
Do not advertise host routes if a network route to the host also exists.

`-m`
Advertise a host route for the machine's primary address. The primary address is found by resolving the machine name returned by `gethostname()`.

`-t`
Increase the debugging level causing more trace information to be written to `stdout`. This option can be specified up to four times to select the highest debug level. The debug level can also be increased or decreased by sending a `SIGUSR1` or `SIGUSR2` signal to the `routed` process.

`-A`
Ignore authenticated RIPv2 packets if authentication is disabled. This option is required for RFC 1723 conformance although it can cause valid routes to be ignored.

`-T <file>`
Output trace information to `<file>` instead of `stderr`.

`-P <parm>{[,<parm>]}`
Equivalent to adding `<parm>{[,<parm>]}` to a single line in the `/h0/SYS/gateways` file.

`-F net[/mask][,metric]`
Only send a "fake" default route to this network. This option is used to minimize RIP traffic but can cause routing loops if the route is propagated.

Description

`routed` requires that routing domain support (`sproute` and `route0`) be loaded in the system. `Routed` is a network routing daemon used to maintain routing tables. It supports the Routing Information Protocol (RIP) versions 1 and 2, and the Internet Router Discovery Protocol as defined in RFC's 1058, 1723, and 1256.

The daemon opens a UDP socket and listens for any routing packets sent to the route port (normally 520) and updates the routing table maintained by `spip`. If the system is configured as a router, copies of the routing table are periodically sent to all directly connected hosts and networks.

When `routed` is started, all non-static routes are removed from the routing table. Any static routes present (such as those added by the `route` utility) are preserved and included in RIP responses if they have a valid RIP metric. Also, if the file `/h0/SYS/gateways` exists, it is read for additional configuration options.

The following options are supported with the `-P` command line option or from `/h0/SYS/gateways`.

`if=<ifname>`
: All other options on this line apply to the interface `<ifname>`.

`passive`
: Do not advertise this interface, and do not use this interface for RIP and router discovery.

`no_rip`
: Disable RIP processing on the specified interface. If Router Discovery Advertisements are not enabled with the `-s` option, `routed` acts as a client router discovery daemon.

`no_ripv1_in`
: Ignore all RIP version 1 responses.

`no_ripv2_in`
: Ignore all RIP version 2 responses.

`no_rdisc`
: Disables the Internet Router Discovery Protocol

`no_solicit`
: Disable the transmission of router discovery solicitations.

`no_rdisc_adv`
: Disable transmission of router discovery advertisements.

`rdisc_pref=<num>`
: Set the preference in router discovery advertisements to `<num>`.

`rdisc_interval=<num>`
: Set the interval that router discovery advertisements are sent to `<num>` seconds and their lifetime to 3 * `<num>`.

See Also

route

rtsol
Router Solicitation Over IPv6

Syntax

```
rtsol [-dD]<interface>
```

IP Functionality

IPv6 addresses

Options

`-d`
Enable debugging.

`-D`
Enable more debugging including to print internal timer information.

Description

Send ICMPv6 Router Solicitation message to the specified interfaces.

telnet

Provide Internet Communication Interface

Syntax

```
telnet [<opts>]
[<hostname> [<portnum>|<servicename>]][<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
Display the description, option, and command syntax for `telnet`.

`-e=<ctrl-character> or "-e=^<character>"`
Set escape to user defined character.

`-n`
No escape available.

`-o`
Show options processing.

Description

`telnet` communicates with another host using the TELNET protocol. Executing `telnet` without parameters defaults to command mode. This is indicated by the prompt `telnet`. In this mode, `telnet` accepts and executes the commands listed below. If executed with parameters, `telnet` performs an `open` command with those parameters.

Once a connection is opened, `telnet` enters input mode. In this mode, typed text is sent to the remote host. To issue `telnet` commands from input mode, precede them with the `telnet` escape character. The escape character is initially set to control-right-bracket (`^]`) but can be redefined either from the environmental variable `TELNETESC` or the command line option `-e`. In command mode, normal terminal editing conventions are available.

Commands

The following commands are available. Type enough of each command to uniquely identify it.

```
capture [<param>]
```

Capture I/O of a `telnet` session to a specified file. `capture` supports four parameters as identified in the following table.

Table 7-9. Capture Parameters

Parameter	Description
`<file>`	Specify a new file in which to write the I/O of a `telnet` session. If `<file>` already exists, `capture` returns an error. When `<file>` is specified, `capture` creates and opens the file and turns on capture mode.
`on`	Turn on capture mode; begins to write I/O to the current specified capture file.
`off`	Turn off capture mode; stops writing I/O to the specified capture file. NOTE: This does not close the file.
`close`	Close the capture file.

`close`
Close the current connection and returns to telnet command mode.

`display`
Display the current `telnet` operating parameters. Refer to `toggle`.

`help [<command>]`
Print help information for specified `?[<command>]` command. If no command is specified, `help` lists them all.

`mode <param>`
Enter line-by-line or character-at-a-time mode.

Table 7-10. Mode Parameters

Parameter	Description
`character`	character at a time
`line`	line-by-line

`open [<host>][<port>]`
Open a connection to the specified `<host>`. If `<host>` is not specified, `telnet` prompts for the host name. `<host>` may be a host name or an internet address specified in dot notation. If the port number is unspecified, telnet attempts to contact a TELNET server at the default port.

`quit`
Close any open telnet connection and exits telnet.

`send [<chars>]`

Transmit special characters to telnet as identified in the following table.

Table 7-11. Send Parameters

Parameter	Description
ao	Abort Output
ayt	"Are You There"
brk	Break
ec	Erase Character
el	Erase Line
escape	Current Escape Character
ga	"Go Ahead" Sequence
ip	Interrupt Process
nop	"No Operation"
synch	"Synch Operation" Command
?	Display send parameters.

`set <param><value>`

Set `<telnet>` operating parameters by setting local characters to specific telnet character functions.

Once set, the local character sends the respective character function to the telnet utility. Specify control characters as a caret (`^`) followed by a single letter. For example, `control-X` is `^X`. `set` supports the parameters identified in the following table:

Table 7-12. Set Parameters

Parameter	Description
`echo <local char>`	Set character to toggle local echoing on and off.
`erase <local char>`	Set the `telnet` erase character.
`flushoutput <local char>`	Abort output.
`interrupt <local char>`	Set the `telnet` interrupt character. This character sends an Interrupt Process.
`kill <local char>`	Set the `telnet` kill character. This character sends an Erase Line.
`quit <local char>`	Set the `telnet` quit character. This character sends a Break.
`?`	Display help information.

`status`

Show the current telnet status. This includes the connected peer and the state of debugging.

`toggle <param>`

Toggle telnet operating parameters, listed in the following table.

Table 7-13. Toggle Parameters

Parameter	Description
`crmod`	Toggles the mapping of received carriage returns. When `crmod` is enabled, any carriage return characters received from the remote host are mapped into a carriage return and a line feed. This mode does not affect characters typed, only those received. This mode is sometimes required for some hosts asking the user to perform local echoing.
`localchars`	Toggles the effects of the set using the `set` command.
`debug`	Toggles debugging mode. When `debug` is on, it opens connections with socket level debugging on. Turning `debug` on does not affect existing connections.
`netdata`	Toggles the printing of hexadecimal network data in debugging mode.
`options`	Toggles the viewing of options processing in debugging mode. Displays options sent by `telnet` as SENT; displays options received from the `telnet` server as RCVD.
`?`	Display help information.

`z`

Suspend the current `telnet` session and forks a shell.

`$`

Suspend the current `telnet` session and forks a shell.

See Also

`telnetd`

telnetd
Incoming Telnet Server Daemon

Syntax

```
telnetd [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
Display the description, options, and command syntax for `telnetd`.

`-6`
Support IPv6 address. Default is IPv4.

`-d`
Print the debug information to standard error.

`-f=<program>`
Program to fork (default = “login”).

`-l`
Print login information to standard error.

`-t`
Idle connection timeout value (in minutes). Default = 0 (no connection timeout).

Description

`telnetd` is the incoming `telnet` daemon process. It must be running to handle incoming `telnet` connection requests. `telnetd` forks the `telnetdc` communications handler each time a connection to the `telnet` service is made.

To save this information for later use, redirect the standard error path and standard output path to an appropriate file on the command line:

```
telnetd -d </nil >>>-/h0/SYS/telnetd.debug&
```

or

```
telnetd -l </nil >>>-/h0/SYS/telnetd.log&
```

If neither option is used, redirect the standard error path to the null driver (along with the standard in/out paths):

```
telnetd <>>>/nil&
```

- Super user account is required to run `telnetd`.
- End the command line with an ampersand (`&`) to place telnetd in the background (example, `telnetd<>>>/nil&`).

The `-f` option is useful for remote access to a diskless embedded system. The need for a RAM disk with a password file is eliminated with the following command line:

```
telnetd -f=shell <>>>/nil/&
```

(`mshell` can be used in place of `shell`)

A telnet to this machine goes straight to a shell prompt wtihout a login prompt.

See Also

`telnet`

`tftpdc`

telnetdc
Telnet Server Connection Handler

Syntax

```
telnetdc [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Description

`telnetdc` is the incoming connection handler for `telnet`. `telnetdc` can only be forked from `telnetd` or `inetd`.

Do not run this from the command line. Only `telnetd` and `inetd` can fork this utility.

The psuedo keyboard modules `pkman`, `pkdrv`, and `pk` are required. OS-9 for 68K also requires `pks`.

See Also

`telnet`

`telnetd`

tftp
Trivial File Transfer Program

Syntax

```
tftp [<host>[ <port>]]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`<host>`
Optional default host for future transfers.

`<port>`
Optional default port for future transfers.

Description

`tftp` is the user interface to the Internet TFTP (Trivial File Transfer Protocol), which allows users to transfer files to and from a remote machine. The remote host may be specified on the command line, in which case `tftp` uses host as the default host for future transfers (see the connect command below).

Commands

Once `tftp` is running, it issues the prompt "`tftp>`" and recognizes the following commands:

`? <command-name> ...`
Print help information.

`ascii`
Shorthand for "`mode ascii`"

`binary`
Shorthand for "`mode binary`"

`blksize <blksize>`
Set the block size for the transfer.

`connect <host> [<port>]`
Set the `<host>` (and optionally `<port>`) for transfers. Note that the TFTP protocol, unlike the FTP protocol, does not maintain connections between transfers; thus, the `connect` command does not actually create a connection, but remembers what host is to be used for transfers. You do not have to use the `connect` command; the remote host can be specified as part of the `get` or `put` commands.

`get <filename>`
`get <remotename> <localname>`
`get <file1> <file2> ... <fileN>`

Get a file, or set of files, from the specified sources. Source can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form `<hosts>:<filename>` to specify both a host and filename at the same time. If the latter form is used, the last hostname specified becomes the default for future transfers.

`mode <transfer-mode>`

Set the mode for transfers; `<transfer-mode>` may be one of `ascii` or `binary`. The default is `ascii`.

`put <file>`
`put <localfile> <remotefile>`
`put <file1> <file2> ... <fileN> <remote-directory>`

Put a file, or set of files, to the specified remote file or directory. The destination can be in one of two forms: a filename on the remote host, if the host has already been specified, or a string of the form `<hosts>:<filename>` to specify both a host and filename at the same time. If the latter form is used, the hostname specified becomes the default for future transfers. If the `<remote-directory>` form is used, the remote host is assumed to be an OS-9 machine. If you need to specify IPv6 numeric address to `<hosts>`, wrap them using square bracket like `[hosts]:filename` to disambiguate the colon.

`quit`

Exit `tftp`. An end of file also exits.

`rexmt <retransmission-timeout>`

Set the per-packet retransmission timeout, in seconds.

`rtimeout <timeout-interval>`

Set the server-side time-out.

`status`

Show current status.

`timeout <total-transmission-timeout>`

Set the total transmission timeout, in seconds.

`trace`

Toggle packet tracing.

`tsize`

Toggle the tsize option.

This will display the transfer size prior to receiving a file, or send the transfer size prior to sending a file.

`verbose`

Toggle verbose mode.

SECURITY CONSIDERATIONS

Because there is no user-login or validation within the TFTP protocol, the remote site will probably have some sort of file-access restrictions in place. The exact methods are specific to each site and therefore difficult to document here.

See Also

`ftp`

`tftpd`

tftpd

Respond to tftpd Boot Requests

Syntax

```
tftpd [<opts>] [<dirname>] [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Options

`-?`
: Display the syntax, options, and command description of `tftpd`.

`-6`
: Support IPv6 address. Default is IPv4.

`-d`
: Log debug information to `<stderr>`.

`-t[=]<num>`
: Set timeout to `<num>` seconds.

Description

`tftpd` is the server daemon handling the client Trivial File Transfer Protocol (TFTP) requests. Once a BOOTP client has received the BOOTP response, it knows the name of its bootfile. The client then issues a TFTP "read file request" back to the same server machine from which it received the BOOTP response. `tftpd` forks `tftpdc` to perform the actual file transfer.

`tftpd` in any system is a security problem because the TFTP protocol does not provide a way to validate or restrict a transfer request since login procedures do not exist. To provide some level of security, `tftpd` only transfers files from a single directory. You can specify this directory on the `tftpd` command line. The default is `/h0/TFTPBOOT`.

Refer to *Chapter 6, BOOTP Server* for more information about the BOOTP Server.

See Also

`tftp`

`tftpdc`

tftpdc
TFTP Server Connection Handler

Syntax

```
tftpdc [<opts>]
```

IP Functionality

IPv4 and IPv6 addresses

Description

`tftpdc` is the incoming communications handler for TFTP. Each time a TFTP service connection is made, `tftpd` forks this process. **See Also**

`tftpdc` is intended to be run only by `tftpd`.

`tftpd`

8 Programming

This chapter covers discusses how to use sockets, broadcasting, and multicasting. The following sections are included:

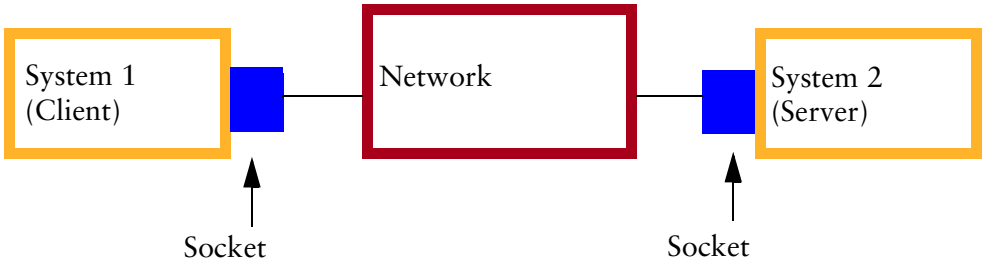
Programming Overview

LAN Communications is based on sockets. Sockets have the following characteristics:

- Sockets are abstractions providing application programs with access to the communication protocols.
- Sockets serve as endpoints of a communication path between processes running on the same or different hosts.
- Sockets enable one system to send and receive information from other systems on the network.
- Sockets also enable programmers to use complicated protocols, such as TCP/IP, with little effort.

The following illustrates how a socket might look in a network:

Figure 8-1. Network



Refer to Appendix A: Example Programs and Test Utilities of the *OS-9 Network Programming Reference*. It contains example programs for various types of sockets. You can use these programs as templates for writing your own programs.

Socket Types

Each socket has a specific address family, a specific protocol, and an associated type. Before establishing a socket, familiarize yourself with the types of sockets available and decide which is best for your needs.

The following address families are supported:

Table 8-1. Address Families

Address Family	Description
`AF_INET`	ARPA internet address family.
`AF_INET6`	Internet address version 6 family.
`AF_ROUTE`	BSD 4.4 style routing domain.

Two high-level protocols, the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP), are supported.

The following types of sockets are supported. Both SOCK_STREAM and SOCK_DGRAM imply a specific protocol to use. When using raw sockets the protocol is indicated when opening the socket.

Table 8-2. Socket Type/Protocol

Socket Type	Description	Implied Protocol
SOCK_STREAM	Stream sockets	TCP
SOCK_DGRAM	Datagram sockets	UDP
SOCK_RAW	RAW Sockets	none

Stream Sockets

Stream sockets are full-duplex byte streams, similar to pipes. Stream sockets must be in a connected state, and only two sockets can be connected at a time. A socket in the connected state has been bound to a permanent destination. Each socket in the connected pair is a peer of the other.

Stream sockets use the TCP protocol. TCP ensures data is not lost or duplicated. If data for which the peer protocol has buffer space cannot be successfully transmitted within a reasonable length of time, the connection is considered broken.

Datagram Sockets

Datagram sockets provide bi-directional flow of data packets called messages. A datagram socket state can be either connected or unconnected.

Datagram sockets use the UDP protocol. UDP does not guarantee sequenced, reliable, or unduplicated message delivery. However, UDP has a reduced protocol overhead and is adequate in many cases.

Raw Sockets

Raw sockets provide an unreliable datagram service. They enable an application to create its own protocol headers and are used to send and receive ICMP messages as well as implementing protocol types not supported directly by IP. The use of raw sockets is restricted to super user processes.

Establishing a Socket

You can establish sockets in several ways. The sequence required to establish connected sockets and unconnected sockets is different. Further, connected sockets are established differently on the client and server sides.

The following describes how to establish sockets and the necessary LAN Communications system calls.

Stream Sockets

Stream sockets use TCP for a transport protocol and must be connected before sending or receiving data. The sequence for connecting a socket is slightly different between the server and client sides.

Server Steps

Complete the following sequence on the server side:

Step 1. Create the socket.

Sockets are created with the `socket()` function. `socket()` requires three parameters:

- address format (`AF_INET` or `AF_INET6`)
- type (`SOCK_STREAM`)
- protocol (0)

`socket()` returns a path number to the created socket. This path number is a handle to the socket, similar to what the `open()` or `create()` functions return.

The following syntax is used to create a stream socket:

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

-OR-

```
s=socket(AF_INET6,SOCK_STREAM,0);
```

`s` is the returned path number of the socket.

Step 2. Bind to the socket.

When a socket is created with `socket()`, it has no association to local or destination addresses. This means a local port number is not assigned to the socket. Server processes operating on a well-known port must specify this port to the system by using the `bind()` LAN Communications function. The `bind()` call binds the port to the address.

The structure used for the address is `sockaddr_in` or `sockaddr_in6`, which is defined in the `in.h` or `in6.h` header file, respecitively.

Normally a server process accepts connections that arrive on any of the machine's interfaces. In this case the IP address passed to `bind()` in the `sockaddr_in` or `sockaddr_in6` structure is the wildcard address 0.0.0.0 or "::", which are represented by `INADDR_ANY` or `IN6ADDR_ANY_INIT`, respectively. If the server process wishes to restrict incoming connections to a single interface, the IP address associated with that interface may be used in place of `INADDR_ANY` or `IN6ADDR_ANY_INIT`.

The following syntax is valid for `bind()` on **IPv4**:

```
#define PORT 27000              /* port number to bind socket to define*/
struct sockaddr_in name;        /* 'name' as a sockaddr_in structure */
memset(&name,0,size of(name));  /*initialize structure to zero*/
name.sin_family = AF_INET;      /* address family is AF_INET */
name.sin_port = htons(PORT);/* assign port number in network byte order */
name.sin_addr.s_addr = INADDR_ANY;
                                /* allow any client to access socket */
bind(s, (struct sockaddr*) &name, sizeof(name));
              /* bind the port to the socket. The 's' parameter is the
              path number for the socket and was returned by socket() */
```

The following syntax is valid for `bind()` on **IPv6**:

```
const struct in6_addr in6addr_any=IN6ADDR_ANY_INIT;
#define PORT 2700 /*port number to bind socket*/
struct sockaddr_in6 name /*define variable "name" as a socket structure */
memset (&name,0,sizeof(name));/*initialize structure to zero */
name.sin6_family=AF_INET6;/*address family is AF_INET6 *?
name.sin6_port=htons (PORT);/*assign port number in network byte order */
name.sin6_addr=inet6addr_any;/*allow any client to access socket*/
bind (s, (struct sockaddr *)&name,sizeof(name));
              /*bind the port to the socket, "s" parameter is path number
              for the socket and was returned by socket*/
```

Step 3. Listen for a Connecting Socket

If you have a stream socket (type `SOCK_STREAM`), the `listen()` function sets the socket in a passive state and allows servers to prepare a socket for incoming connections. It also informs the system to queue multiple client requests which arrive at a socket very close in time. This queue length must be specified in the `listen()` call. After client requests fill this queue, any further requests are rejected, and the client socket is notified with an `ECONNREFUSED` error.

`listen()` requires two parameters:

- the socket's path number
- the client's requested queue size

For example, the following syntax is valid for `listen()`:

```
listen(s,4);
```

In this example, four client requests are allowed to queue. When the queue is full, additional requests are refused.

Step 4. Accept a connecting socket.

Once `listen()` sets up a passive socket, the `accept()` function is used to accept connections from clients. `accept()` blocks the process if pending connections are not presently queued.

If the socket is set up as non-blocking and an `accept()` call is made with no pending connections, `accept()` returns with an `EWOULDBLOCK` error.

`accept()` requires three parameters:

- the socket's path number
- a pointer to a variable with type `sockaddr_in` or `sockaddr_in6`
- the size of the `sockaddr_in` or `sockaddr_in6` structure

The following syntax is valid for `accept()` on **IPv4**:

```
struct sockaddr_in from;
int ns, size;

size=sizeof(struct sockaddr_in);
ns=accept(s, (struct sockaddr*) &from, &size);
```

The following syntax is valid for `accept()` on **IPv6**:

```
struct sockaddr_in6 from;
int ns, size;

size=sizeof(struct sockaddr_in6);
ns=accept(s, (struct sockaddr*) &from,

              (socklen.t*) &size);
```

Once `accept()` returns without error, a connection has been made to a client process and the following events occur:

1. The client request is removed from the `listen` queue.
2. The client foreign address was put into the `from` variable. This allows the server to see who connected to it.
3. `accept()` returns a new socket path number (ns).

This new socket transfers data to and from the client process. It represents a connected path to the client process. The original socket path number returned by `socket()` can then be used to accept further connections or it can be closed, determined by the application.

At this point in the server process, the socket has been established and communication between the two processes can begin.

Client Steps

Complete the following steps on the client side:

Step 1. Create the socket.

Sockets are created with the `socket()` function. `socket()` requires three parameters:

- an address format (`AF_INET` or `AF_INET6`)
- a type (`SOCK_STREAM` or `SOCK_DGRAM`)
- a protocol (usually `0`)

`socket()` returns a path number to the created socket. This path number is a handle to the socket, similar to what the `open()` or `create()` functions return.

The following syntax is used to create a stream socket:

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

-OR-

```
s=socket(AF_INET6,SOCK_STREAM,0);
```

`s` is the returned path number to the socket.

Step 2. Connect to a listening socket.

To connect to a listening socket, use the `connect()` function call.

`connect()` requires three parameters:

- the path descriptor returned by `socket()`
- a pointer to a structure containing the name
- the length of the name

Using Connect

The following example demonstrates the use of `connect()` on **IPv4**:

```
#define PORT 27000/* Port number of server process */
struct sockaddr_in ls_addr;
memset(&ls_addr, 0, sizeof(ls_addr));/* Initialize structure to zero */
ls_addr.sin_family = AF_INET;
ls_addr.sin_port = htons(PORT);/* Assign port in network byte order */
ls_addr.sin_addr.s_addr = inet_addr("thor");
                    /* IP address we wish to connect to */
connect (s, (struct sockaddr *)&ls_addr, sizeof(ls_addr));
```

The following example demonstrates the use of `connect()` on **IPv6**:

```
#define PORT 27000/* Port number of server process */
struct sockaddr_in6 ls_addr;
memset(&ls_addr, 0, sizeof(ls_addr));/* Initialize structure to zero */
ls_addr.sin6_family = AF_INET6;
ls_addr.sin6_port = htons(PORT);/* Assign port in network byte order */
inet_pton(AF_INET6, "thor", & ls_addr.sin6_addr);
          /*IP address to connect to */
connect (s, (struct sockaddr *)&ls_addr, sizeof(ls_addr));
```

The `connect()` call automatically binds the socket to a local port number if `bind()` was not explicitly called. `bind()` is not usually called by clients unless the server restricts the ports from which it will accept connections.

Datagram Sockets

Datagram sockets use UDP as a transport protocol and can be connected or connectionless.

The steps for creating connected and connectionless sockets are the same for both the server and client sides:

1. Create the socket

The following syntax is used to create a datagram socket:

```
int s;
s=socket(AF_INET,SOCK_DGRAM,0);
```

-OR-

```
s=socket(AF_INET6,SOCK_DGRAM,0);
```

2. Bind a port and/or address to the socket

As with TCP, the client does not need to bind a socket.

The following syntax is valid for `bind()` on **IPv4**:

```
struct sockaddr_in ls_addr;
bind(s,(struct sockaddr*) &ls_addr, sizeof (ls_addr);
```

The following syntax is valid for `bind()` on **IPv6**:

```
struct sockaddr_in6 ls_addr;
bind(s,(struct sockaddr*) &ls_addr, sizeof (ls_addr);
```

Connect a Socket

Either the client or server can optionally call `connect()` on a UDP socket. Unlike TCP sockets, calling `connect()` on a UDP socket does not cause any information to be sent to the peer. It simply stores the peer address in the local socket and allows the application to use `send()` and `recv()` instead of `sendto()` and `recvfrom()`.

A process may "disconnect" a UDP socket by calling `connect()` with `INADDR_ANY` or `INADDR6_ANY_INIT` as the IP address. The application can then no longer use `send()` and `receive()`, but can still use `sendto()` and `recvfrom()`.

The `connect()` function call requires three parameters:

- the path descriptor returned by `socket()`.
- a pointer to a `sockaddr_in` or `sockaddr_in6` structure containing the peer port and IP address.
- the size of the `sockaddr_in` or `sockaddr_in6` structure.

The following syntax is used to connect a datagram socket on **IPv4**:

```
struct sockaddr_in ls_addr;
connect(s, (struct sockaddr *)&ls_addr, sizeof(ls_addr));
```

The following syntax is used to connect a datagram socket on **IPv6**:

```
struct sockaddr_in6 ls_addr;
connect(s, (struct sockaddr *)&ls_addr, sizeof(ls_addr));
```

Header Files

The following table describes header files associated with sockets.

Table 8-3. Header Files Associated with Berkeley Sockets

Header File	Description
`netinet/in.h`	Provide structures and functions for the socket family `AF_INET`.
`netinet6/in6.h`	Provide structures and functions for the socket family `AF_INET6`.
`netdb.h`	Provide structures for hosts, networks, services, protocols, and functions calling socket address structures and socket family `AF_INET` structures.
`sys/socket.h`	Provide a definition to the socket address structure.

Header files are found in `MWOS/SRC/DEFS/SPF/BSD`.

The main internet application structure is `sockaddr_in` and `sockaddr_in6`, defined in the `in.h` and `in6.h` header files.

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

struct in_addr  {
   u_long s_addr;
};


struct sockaddr_in6 {
   u_int8_t         sin6_len;
   u_int8_t         sin6_family;
   u_int16_t        sin6_port;
   u_int32_t        sin6_port;

   struct in6_addr  sin6_addr;
   u_int32_t        sin6_scope_id;
};

struct in6_addr  {
   union {

      u_int8_t  __u6_addr8[16];

      u_int16_t __u6_addr16[8];

      u_int32_t __u6_addr32[4];

   } __u6_addr;
};
```

The `hostent` structure is in the `netdb.h` header file. It is used to get address information about any host in the `inetdb` database:

```
struct hostent  {
    char *h_name;/* pointer to host name */
    char **h_aliases;/* pointer to the pointer */
                /* to the alias for the host */
    int  h_addrtype;/* host address type */
    int  h_length;/* length of host */
    char **h_addr_list;/* list of addresses from */
                /* name server */
    #define h_addr   h_addr_list[0]
                /* backwards compatibility:
                 pointer to the address of

                 the host */

}
```

Reading Data Using Sockets

Four functions are available for reading data:

Table 8-4. Reading Functions

Function	Description

`_os_read()`
`read()`
`recv()`
	Returns the data available on the specified socket path, up to the amount requested. For packet oriented protocols, such as UDP, only a single datagram is returned, even if more are available. These calls are only valid on connected sockets.

`recvfrom()`	Functions similar to `recv()` except that it also returns a `sockaddr_in` or `sockaddr_in6` structure containing the address information of the sender. `recvfrom()` works with unconnected sockets.
`recvmsg()`	Similar to `recvfrom()`; however, this uses an `msghdr` structure to minimize the number of directly supplied parameters.

For more specific information on `recv()` `recvfrom()`, and `recvmsg()` refer to the *OS-9 Network Programming Reference*.

For more specific information on `_os_read()` and `read()`, refer to the *Ultra C Library Reference*.

Writing Data Using Sockets

Four similar functions are available for writing data.

Table 8-5. Writing Functions

Function	Description

`_os_write()`
`write()`
`send()`
	Writes data from a buffer to the socket path. These functions are only valid with connected sockets.

`sendto()`	Functions the same as `send()` except it also takes a `sockaddr_in` or `sockaddr_in6` parameter specifying where the data is to be sent. `sendto()` works with unconnected sockets.
`sendmsg()`	Similar to `sento()`; however, this uses an `msghdr` structure to minimize the number of directly supplied parameters.

If a socket path has been set to nonblocking and there is not enough buffer space available, the call can succeed, but only part of the data may be sent. Check the return value to see how many bytes were sent.

For more specific information on `send()` and `sendto()`, refer to the *OS-9 Network Programming Reference*.

For more specific information on `_os_write()` and `write()`, refer to the Hawk™ On-line Help System from the Hawk interface.

Setting up Non-Blocking Sockets

When a program tries to perform some socket functions, the program can block. The following are situations in which this would happen:

- Read from a socket that has no data (`read/recv/recvfrom`).
- Write to a socket that does not have enough space to satisfy size of write (`write/send/sendto`).
- Accept a connection with no waiting connection available (`listen/accept`).
- Calling connect on a TCP socket.

LAN Communications enables you to create a non-blocking socket to prevent the above problems. When your program tries to access a non-blocking socket in one of the above listed blocking conditions, the socket returns the error `EWOULDBLOCK`. The connect call is the exception and returns `EINPROGRESS`.

The following function can be used to set a socket to non-blocking or blocking:

`spath` is the socket path.

`blockflag` is either set to `IO_SYNC` for blocking or `IO_ASYNC` for nonblocking.

```
#include <SPF/spf.h>

error_code setblock(path_id spath, u_int8 blockflag)
{
   struct spf_popts sopts;
   u_int32 soptsz=sizeof(sopts)

   if ((errno=_os_gs_popt(spath,&soptsz,&sopts)) !=SUCCESS) {
      return(errno);
   }
   sopts.pd_ioasync=blockflag;
   return (_os_ss_popt(spath,soptsz,&sopts));
} /*end of setblock*/
```

The common UNIX function `ioctl()` can also be used.

```
#include <UNIX/ioctl.h>
error_code setblock(path_id spath, u_int8 blockflag){
   if (ioctl(spath, FIONBIO, dblockflag)<SUCCESS) {
   return (errno);
   }
   return(SUCCESS);
}
```

Broadcasting

Broadcasting support allows the network to deliver one copy of a packet to all attached hosts. Due to the extra overhead involved, multicasting is much preferred over broadcasting as a way to deliver data to multiple recipients. Also, broadcast packets are only sent to the local network and can not be forwarded by routers.

Stream sockets cannot be used for broadcasting messages. The `SO_BROADCAST` option must be enabled before broadcasting on a datagram socket.

Broadcasting Process

The broadcasting process sends its message to the pre-selected port number.

The broadcasting process uses the internet address for the address of the network. Usually this address is the same as the sender's broadcast address. For example, on a class C network, a system with address `192.7.44.105` has a broadcast address of `192.7.44.255`.

The broadcasting process must open the socket as a datagram socket. A `bind()` call is required before broadcasting. The broadcasting process must use the `sendto()` function.

Do not use `send()` or `_os_write()`.

Receiving Process

The receiving processes must receive from the pre-selected port number the broadcaster is using.

The receiving socket must be opened as a datagram socket. A `bind()` call is required before attempting to receive. The receiving processes use the `recvfrom()` function.

Multicasting

Multicasting support allows the delivery of a single packet to multiple hosts. The main difference between multicasting and broadcasting is that with multicasts, only the machines interested in receiving the packets see them. Also, multicast packets can be routed across networks, while broadcasts are only seen on the local network.

Currently, LAN Communications does not support routing multicast packets. It can only be used as an end node to send or receive them. Also, only datagram sockets can be used for multicasting.

Sending Multicasts

The same mechanism used to send unicast datagrams can be use to send multicasts by simply using a multicast destination IP address. However, to provide more control, 3 additional socket options may be used.

- `IP_MULTICAST_TTL`—This option limits the number of routers a multicast packet can pass through before being dropped. The default value is 1, which means multicasts will not travel beyond the locally attached network. Link local multicasts (addresses 224.0.0.0 - 224.0.0.255) are never routed, regardless of the TTL value.
- `IP_MULTICAST_LOOP`—The default is for multicasts not to be sent to the loopback interface. If you want to see the multicast packets you send, or another application on your hosts needs to see them, this option needs to be enabled.
- `IP_MULTICAST_IF`—When sending multicast packets, the output interface is selected via the normal routing procedure. This option allows an application to override this selection and pick the appropriate interface.

The normal routing lookup is performed first and fails, the `IP_MULTICAST_IF` option is not checked. This means a route to the selected multicast address must exist in the routing table. Normally this consists of either a network route such as 224.0.0.0 or, more commonly, the presence of a default route.

Receiving Multicasts

In order for an application to receive multicast packets, it must join the appropriate multicast group. After the group has been joined, receiving multicast packets is no different than normal unicast packets. Two socket options are provided for joining and leaving multicast groups.

- `IP_ADD_MEMBERSHIP`—This option informs IP that an application wants to receive packets for the given multicast group. If this is the first application on the host to join the group on the requested interface, IP will notify the interface to begin receiving multicast packets for group. Additionally, any multicast routers present on the subnet will be notified that a host has joined the group.
- `IP_DROP_MEMBERSHIP`—This option informs IP that packets for the indicated group should no longer be sent to the application. If this is the last application receiving this group on the indicated interface, IP will notify the interface to no longer receive the group.

Refer to the *OS-9 Network Programming Reference* for more information on using `setsockopt`.

Controlling Socket Operations

Socket level options control the socket's operation. These options are defined in `socket.h`. `setsockopt()` and `getsockopt()` are used to set and get options.

Table 8-6. Socket Level Options

Option	Description
`SO_DEBUG` *	Turn on recording of debugging information. This allows debugging in the underlying protocol modules.
`SO_REUSEADDR`	Indicate the rules used in validating addresses supplied in a `bind()` call should allow reuse of local addresses.
`SO_KEEPALIVE`	Allow the periodic transmission of messages on a connected socket. If a connected party fails to respond to these messages, the connection is considered broken and the socket is closed.
`SO_DONTROUTE`	Indicate 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_LINGER`	Control the actions taken when unsent messages are queued on a socket and a `close()` is performed. If the socket promises reliable delivery of data and `SO_LINGER` is set, the system blocks the process on the close attempt until it is able to transmit the data or until it decides it is unable to deliver the information. A time-out period, referred to as the *linger interval*, is specified by `setsockopt()` when `SO_LINGER` is requested.
`SO_BROADCAST`	Enable an application to send broadcast messages. Valid for datagram sockets only.
`SO_OOBINLINE`	Place any incoming out-of-band data in the normal input queue.
`SO_SNDBUF`	Set or retrieve the size of the socket send buffer.
`SO_RCVBUF`	Set or retrieve the size of the socket receive buffer.
`SO_SNDLOWAT`	Indicate the minimum amount of space that must be available in the send buffer before data is accepted for sending. If this much space is not available the process blocks, or if the socket is non-blocking an `EWOULDBLOCK` error is returned.
`SO_RCVLOWAT`*	Indicate the minimum amount of data that must be available to be read before `select` considers a path "readable".
`SO_SNDTIMEO`*	Control the maximum amount of time a process will block waiting for buffer space when sending data.
`SO_RCVTIMEO`*	Control the maximum amount of time a process will block waiting for incoming data.
`SO_TYPE`	Return the type of a socket such as `SOCK_STREAM` or `SOCK_DGRAM`. This option is only valid for `getsockopt()`.
`SO_ERROR`	Retrieve the current socket error if one exists. This option is only valid for `getsockopt()`.
`SO_USELOOPBACK`	This option is only valid for sockets in the routing domain (`AF_ROUTE`). It controls whether a process receives a copy of everything it sends.

* Unsupported option

A Configuring LAN Communications

This appendix explains how to configure and start LAN Communications for Internet access. It includes the following sections:

Configuring Network Modules

You can configure the host, network, DNS client, routing, and interface information by updating text files found in MWOS/SRC/ETC.

Step 1: Updating Files

You may need to update the following files for your system.

- hosts
- networks
- resolv.conf (if using DNS Client)
- routes.conf (if using static routing)
- interfaces.conf (located in MWOS/<OS>/<CPU>/PORTS/<BOARD>/SPF/ETC or MWOS/SRC/ETC)

Step 2: Creating Modules

Create the inetdb/inetdb2/rpcdb modules by running os9make on cross-development systems. This executes the idbgen and rpcdbgen utilities.

The makefile is found in MWOS/SRC/ETC. If you are working from a ports directory, the makefile is found in the MWOS/<OS>/<CPU>/PORTS/<TARGET>/SPF/ETC directory in the port.

The inetdb/inetdb2/rpcdb data modules will be placed in MWOS/<OS>/<CPU>/CMDS/BOOTOBJS/SPF, or the local ports CMDS/BOOTOBJS/SPF.

The idbgen reads files in the local SPF/ETC directory and/or files in MWOS/SRC/ETC. The rpcdbgen utility reads files from MWOS/SRC/ETC.

The idbdump utility can be used to print out the contents of the inetdb/inetdb2 data modules. The rpcdump utility can be used to print out the contents of the rpcdb data module.

The rpcdbgen utility and rpcdb data module are described in the *Using Network File System/Remote Procedure Call* manual.

Contents of inetdb

Below is an example of the contents of an inetdb data module.

```
> idbdump inetdb
Dump of OS-9/PowerPC INETDB network database module [inetdb]
  Compatability :  1
  Version Number:  9

Host Entries:
Host Name    Host Address     Host Aliases
----------   ---------------  -------------
localhost              127.0.0.1         me
```

```
Hosts Equivalent Entries:

Network Entries:
Network Name    Network Address   Network Aliases
-------------   ---------------   ---------------
loopback              127
private-A             10
private-B             172.1
private-C             192.1.2
localnet              10.0.0

Protocol Entries:
Protocol   Number   Protocol Aliases
--------   ------   ----------------
ip          0         IP
icmp        1         ICMP
igmp        2         IGMP
tcp         6         TCP
udp         17        UDP

Service Entries:
Service Name   Port/Protocol  Service Aliases
------------   -------------  ---------------
ndp               13312/tcp          ndpd
npp               13568/tcp          nppd
echo                  7/tcp
echo                  7/udp
discard               9/tcp
discard               9/udp
daytime              13/tcp
daytime              13/udp
chargen              19/tcp
chargen              19/udp
ftp-data             20/tcp
ftp-data             20/udp
ftp                  21/tcp
ftp                  21/udp
telnet               23/tcp
telnet               23/udp
nameserver           42/tcp
nameserver           42/udp
bootps               67/tcp
bootps               67/udp
bootpc               68/tcp
bootpc               68/udp
tftp                 69/tcp
tftp                 69/udp
touyr               520/udp
```

```
InetD Configuration Entries:
Service Name  Socket Type  Protocol  Flags   Owner  Server Arguments
------------  -----------  --------  ------  -----  ----------------
ftp         SOCK_STREAM  tcp      WAIT    0.0    ftpdc
telnet      SOCK_STREAMtcp      WAIT    0.0    telnetdc
echo        SOCK_STREAMtcp      WAIT    0.0    internal
echo        SOCK_DGRAMudp      WAIT    0.0    internal

Resolve Configuration Entries:
   Domain Name:  alpha.com
   Nameserver List:
      1:  10.0.0.1
      2:  10.0.0.2
      3:  10.0.0.3
   Search List:
      1:  alpha.com

Host Configuration Entries:

Interface Configuration Entries:

Hostname Configuration Entries:

Route Configuration Entries:
Destination Gateway    NetmaskType
----------- -------- -------- ----

RPC Entries:
Program            NumberAliases
---------------    -------   -------
portmapper         100000     rpcinfo
rstatd             100001     rup
rusersd            100002     rusers
nfs                100003     nfsrbf
ypserv             100004     yp
mountd             100005     mount  showmount
ypbind             100007
walld              100008     rwall  shutdown
yppasswdd          100009     yppasswd
etherstatd         100010     etherstat
rquotad            100011     rquotaprog  quota  rquota
sprayd             100012     spray
rje_mapper         100014
selection_svc      100015     selnsvc
database_svc       100016
```

```
rexd                100017     on
llockmgr            100020
nlockmgr            100021
statmon             100023
status              100024
bootparam           100026
ypupdated           100028ypupdate
keyserv             100029keyserver
dird                76    rdir
msgd                99    msg
sortd               22855 rsort
```

Contents of inetdb2

Below is an example of the `inetdb2` data module.

```
>idbdump inetdb2
Dump of OS-9000/PowerPC INETDB network database module [inetdb2]
  Compatability :  1
  Version Number:  9
Host Entries:
Host Name   Host Address   Host Aliases
---------   ------------   -------------

Hosts Equivalent Entries:
Network Entries:
Network Name   Network Address   Network Aliases
------------   ---------------   ----------------

Protocol Entries:
Protocol   Number   Protocol Aliases
---------  ------   -----------------

Service Entries:
Service Name    Port/Protocol   Service Aliases
-------------   -------------   ---------------

InetD Configuration Entries:
Service Name Socket Type  Protocol   Flags   Owner  Server Arguments
------------ -----------  ---------  ------  ------ ----------------
Resolve Configuration Entries:
Host Configuration Entries:
```

```
Interface Configuration Entries:
Interface:  enet0
  Binding:  /spe30/enet
    Flags:  0x1 <UP>
 MW_Flags:  0x8000 <NO_MULTICAST>
      MTU:  0
   Metric:  0
   Address          Netmask          Broadcast
   172.16.4.32      0.0.0.0          0.0.0.0
Interface:  ppp0
  Binding:  /ipcp0
    Flags:  0x1 <UP>
      MTU:  0
   Metric:  0
   Address          Netmask          Broadcast
   None
Hostname Configuration Entries:
Hostname: Beta
Route Configuration Entries:
Destination  Gateway Netmask Type
------------ ------- ------- ----
RPC Entries:
Program            Number    Aliases
```

Step 3: Configure the Interface Descriptor

To set port-specific parameters such as baud rate or interrupt number, you must configure the interface descriptor for PPP, SLIP, or Ethernet. To do this, perform the following steps:

Step 1. Update the `spf_desc.h` descriptor file in the local ports directory. For Ethernet, this file is found in the port directory for your target under `MWOS/<OS>/<CPU>/PORTS/<TARGET>/SPF/<driver>/DEFS`.

Step 2. To make, run `os9make` in the `SPF/<driver>` directory.

Step 4: Load LAN Modules

The minimum modules required to test connectivity with `ping` are included in the following list. Uncomment or add these files to your bootlist.

```
inetdb              inetdb2
sysmbuf             mbinstall
spip                ip0
spudp               udp0
sptcp               tcp0
netdb_dns           spf
spraw               raw0
ipstart             ping
```

- For SLIP, PPP, or Ethernet support, uncomment the appropriate driver(s) and descriptor(s).
- To include other utilities in your boot, such as telnet, uncomment the appropriate entry.

An example bootlist follows. Depending on the OS software version you are using, you may need a relative path of ../../../../../../<CPU> or `../../../<CPU>`.

```
*
* SysMbuf P2 Module:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/SysMbuf
*
* SysMbuf utilities:
*
../../../../../../<CPU>/CMDS/mbinstall
*../../../../../../<CPU>/CMDS/mbdump
*
* SPF file manager:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spf
*
* LAN protocol drivers and descriptors:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/sptcp
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/tcp0
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spudp
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/udp0
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spip
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/ip0
```

```
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spraw
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/raw0
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/sproute
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/route0
*
* Pseudo Key Board File Manager and Driver required for telnetdc)
*
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/pkman
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/pkdvr
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/pk
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/pks <OS-9 only>
*
* LAN trap handler and configuration data module:
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/netdb_dns
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/inetdb
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/inetdb2
*
* LAN SLIP drivers and descriptors:
*
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spslip
*../../../CMDS/BOOTOBJS/SPF/spsl0
*
* LAN PPP client drivers and descriptors:
*
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spipcp
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/splcp
*../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/sphdlc
*../../../CMDS/BOOTOBJS/SPF/spipcp0
*../../../CMDS/BOOTOBJS/SPF/splcp0
*../../../CMDS/BOOTOBJS/SPF/sphdlc0
*
* LAN PPP client utilities:
*
*../../../../../../<CPU>/CMDS/pppd
*../../../../../../<CPU>/CMDS/pppauth
*
* Ethernet Protocol Driver (required for hardware ethernet drivers)
*
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/spenet
```

```
../../../../../../<CPU>/CMDS/BOOTOBJS/SPF/enet
*
* LAN ethernet driver and descriptor:
*
../../../CMDS/BOOTOBJS/SPF/<ethernet driver>
../../../CMDS/BOOTOBJS/SPF/<ethernet descriptor>
*
* LAN utilities:
*
*../../../../../../<CPU>/CMDS/arp
*../../../../../../<CPU>/CMDS/bootpd
*../../../../../../<CPU>/CMDS/bootptest
*../../../../../../<CPU>/CMDS/dhcp
*../../../../../../<CPU>/CMDS/ftp
*../../../../../../<CPU>/CMDS/ftpd
*../../../../../../<CPU>/CMDS/ftpdc
*../../../../../../<CPU>/CMDS/hostname
*../../../../../../<CPU>/CMDS/idbdump
*../../../../../../<CPU>/CMDS/idbgen
*../../../../../../<CPU>/CMDS/ifconfig
*../../../../../../<CPU>/CMDS/inetd
../../../../../../<CPU>/CMDS/ipstart
*../../../../../../<CPU>/CMDS/ndbmod
*../../../../../../<CPU>/CMDS/netstat
../../../../../../<CPU>/CMDS/ping
*../../../../../../<CPU>/CMD/ping6
*../../../../../../<CPU>/CMDS/route
*../../../../../../<CPU>/CMDS/routed
*../../../../../../<CPU>/CMD/routed6
*../../../../../../<CPU>/CMD/rtsol
*../../../../../../<CPU>/CMDS/telnet
*../../../../../../<CPU>/CMDS/telnetd
*../../../../../../<CPU>/CMDS/telnetdc
*../../../../../../<CPU>/CMDS/tftpd
*../../../../../../<CPU>/CMDS/tftpdc
*
```

Also available is the example `loadspf` script in `MWOS/SRC/SYS`. This file provides an example of the modules to load in order to start the protocol stack.

If you are using a disk-based system, the following file can be used as an example or modified to match your system. This file can be copied from `MWOS/SRC/SYS`.

```
*
* loadspf for SPF LAN Communication Package Release
*
*
* Load SPF System Modules
*
chd CMDS/BOOTOBJS/SPF
*
load -d inetdb  inetdb2    ;* Load system specific inetdb
                              modules
load -d SysMbuf            ;* System Mbuf module
                           ;*(sets size of mbuf pool on
                              OS-9)
load -d pkman pkdvr pk     ;* Pseudo keyboard modules
                    (required for telnetdc)
*load -d pks;              ;* Pseudo keyboard addtional
                              for 68K
*
*
load -d spf                ;* SPF file manager
load -d spip ip0           ;* IP driver and descriptor
load -d sptcp tcp0         ;* TCP driver and descriptor
load -d spudp udp0         ;* UDP driver and descriptor
load -d spraw raw0         ;* RAW IP driver and descriptor
*load -d sproute route0    ;* Dynamic Routing driver and
                              descriptor
*
* Load SPF Trap library and Commands
* Load one of the following Netdb name resolution trap
* handlers
*
load -d netdb_dns         ;* Load trap handler for  DNS
                             * name resolution
*
* Load SPF Ethernet Drivers and Descriptors
*
*load -d spenet enet       ;* Ethernet Protocol driver and descriptor
(required by hardware Ethernet drivers)
*
```

```
* Load SPF Drivers and Descriptors ... Uncomment those
* needed
*
* <Ethernet drivers and descriptors>
*
* Serial Drivers and Descriptors
*
*load -d spslip   spsl0    ;* Slip /t1
*load -d spipcp   ipcp0    ;* PPP IPCP
*load -d splcp    lcp0     ;* PPP LCP
*load -d sphdlc   hdlc0    ;* PPP HDLC
*(chd ../..; load -d chat pppd ppplog pppauth; chd BOOTOBJS/SPF)    ;
*PPP Utilities
*
*
* Chd up to CMDS directory
*
chd ../..
load -d mbinstall                ;* Load mbinstall memory
                                  * handler (or can be done
                                  * within init
load -d ipstart                  ;* Load ipstart stack
                                  * initializer
*
*
*load -d routed routed6d          ;* Dynamic routing daemon
*load -d telnet telnetd telnetdc ;* Telnet support
                                  * modules
*load -d ftp ftpd ftpdc           ;* FTP support modules
*load -d tftpd tftpdc bootpd      ;* Bootp/TFTP support
                                  * modules
*load -d inetd                    ;* Super-Server Daemon
*load -d idbgen idbdump ndbmod    ;* Development tools
*load -d route hostname ifconfig arp  ;* Runtime tools
load -d netstat ping ping6 rtsol      ;* Statistics, verification tools */
```

Starting the Protocol Stack

To begin using the protocol stack, complete the following steps:

Step 1. Install the network or SPF memory handler.

After loading the modules, the first step is to install the network or SPF memory buffer handler (`SysMbuf`). This can be done with the `mbinstall` utility.

```
shell> mbinstall
```

Step 2. Start the TCP/IP protocol stack by executing the `ipstart` utility.

```
shell> ipstart
```

If you did not define your interfaces in `inetdb2`, you can add them now using `ifconfig`.

Running `devs` and `procs` shows the protocol drivers initialized and SPF receive thread process in the process table.

Step 3. Use the `ping` utility to verify that the network components are working correctly.

```
shell> ping localhost
```

The following appears on your screen:

```
PING localhost (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: ttl=255 time=0 ms
```

Next test to another system.

```
shell> ping <hostname>
```

Where `<hostname>` is the name of a computer in the `inetdb` module (hosts section), or a name that can be resolved by the DNS server specified in the `resolve.conf` section of `inetdb`. `<hostname>` can also be an IP address.

Something similar to the following appears on your screen:

```
PING delta.microware.com (172.16.1.40): 56 data bytes
64 bytes from 172.16.1.40: ttl=255 time=10 ms
```

Example Configuration

If you are using a disk-based system, the following `startspf` file can be used as an example or modified to match your system. This file can be copied from `MWOS/SRC/SYS`.

```
* startspf
* Shell Script to Start SPF System
*
* Set default directories before starting daemon programs
*
chd /h0
chx /h0/cmds
*
* Load SPF modules
*
SYS/loadspf
*
```

```
* Load and start mbuf handler  (May be done via p2 list in init module)
*   Allow for error returned in case sysmbuf is already initialized.
*
-nx
mbinstall
-x
*
* Start SPF system using ipstart
*
ipstart
*
* Add interfaces not specified in inetdb2
*
*ifconfig enet0 <my_address> binding /<dev>/enet
*ifconfig ppp0 binding /ipcp0
*
* Add any static routes. Even if running routed it may be useful
*   to add multicast routes.
*
*route add -net 224.0.0.0 <my_address>
*
* Start service daemons
*   routed: Dynamic routing server
*   inetd: FTP/Telnet and other protocols server
*   telnetd: Remote terminal server
*   ftpd: Remote file-transfer server (FTP)
*   bootpd: Network boot protocol server
*   tftpd: Trivial file transfer protocol server
*
routed <>>>/nil&
inetd <>>>/nil&
*telnetd <>>>/nil &
*ftpd <>>>/nil &
*bootpd /h0/TFTPBOOT/bootptab <>>>/nil&
*tftpd /h0/TFTPBOOT <>>>/nil &
*
*   spfndpd: Hawk User state debugging daemon
*   spfnppd: Hawk Profiling daemon
*
spfndpd <>>>/nil &
*spfnppd <>>>/nil &
```

Configuration Files

Files identified in Table A-1 reside in `MWOS/SRC/ETC` and provide the protocol stack with pertinent information. Utilities available to create, modify, or dump the `inetdb` modules are: `idbgen`, `ndbmod`, and `idbdump`, respectively.

Table A-1. Configuration Files

File	Description
`hosts`	This contains a list of hosts known to your system. If using DNS, this file may not need to be updated. Otherwise, add an entry for each of your hosts (including the host you are using) to the file.
`networks`	This contains a list of networks analogous to hosts.
`protocols`	This contains a list of protocols available.
`services`	This contains a list of services available.
`inetd.conf`	This contains a list of server daemon routines `inetd` supports (for example, telnet and FTP).
`resolv.conf`	This contains configuration information for a Domain Name System (DNS).
`interfaces.conf`	This contains configuration information for hostname and network interfaces to initialize when the stack is brought up.
`routes.conf`	This contains a list of static routes to add when the stack is brought up.
`rpc`	This contains a list of RPC support services, program number, and the client program.

Each file contains single-line entries consisting of one or more fields and (optionally) comments. Fields are separated by any number of spaces and/or tab characters. A pound sign (`#`) indicates the beginning of a comment. The comment includes all characters up to the end of the line. All files are described in the following sections.

Hosts

The `hosts` file contains information regarding the known hosts on the network. For each host, a single-line entry must be present. Each entry contains the following:

- internet address
- official host name
- aliases (optional)

Internet addresses are specified in the conventional dot notation for IPv4 or the "::" notation for IPv6. Host names can contain any printable character other than a field delimiter, new line, or comment character.

The following example `hosts` entries consist of an address, name, and comment. Addresses may be in either IPv4 or IPv6 style.

```
IPv4:   192.1.1.1 balin #documentation

IPv6:   fec0::2396 thorin #moria
```

Networks

The `networks` file contains information regarding the known networks composing the internet. A single line entry must be present for each network. Each entry consists of the following information:

- official network name
- network number
- aliases (optional)

Network numbers are specified in the conventional dot notation. Network names can contain any printable character other than a field delimiter, new line, or comment character.

The following example `networks` entry consists of a name, number, alias, and comment:

```
arpanet 10 arpa #just a comment
```

Consult your local network administrator for conventions to determine the proper host and network information.

Protocols

The `protocols` file contains information regarding the known protocols used in the internet. A single-line entry must be present for each protocol. Each entry contains the following information:

- official protocol name
- protocol number
- aliases (optional)

Protocol names can contain any printable character other than a field delimiter, new line, or comment character.

The following example `protocols` entry consists of a name, number, alias, and comment:

```
udp 17 UDP # user datagram protocol
```

Services

The `services` file contains information regarding known services available to your system. A service is a reserved port number for a specific application. For example, FTP is a service reserved at port 21. Each service also specifies the protocol it uses. Because each network can have a unique `services` file, networks can offer different services.

A single-line entry must be present in the `services` file for each service. Each entry contains the following information:

- official service name
- port number at which the service resides
- official protocol name
- aliases (optional)

Service names can contain any printable character other than a field delimiter, new line, or comment character.

The port number and protocol name are considered a single item; a slash character (`/`) separates them.

The following example services entry consists of a service name, port number, protocol name, alias, and comment:

```
shell 515/tcp cmd #no passwords used
```

To create a service, select a port number greater than 1024 (port numbers less than 1024 are reserved), a protocol, and a name; then add this information to the `services` file.

inetd.conf Configuration File

The `inetd.conf` file contains information regarding program services the `inetd` service daemon handles. `inetd` can currently take the place of `ftpd` and `telnetd`.

A single-line entry must be present in the `inetd.conf` file for each service available. Entries contains the following information:

- service name (program)
- socket type
- protocol
- flags
- user
- server path name (child process to fork)
- [additional arguments]

The following example `inetd.conf` entry consists of a service name, socket type, protocol, flags, user, and server path name.

```
ftp  stream tcp wait root ftpdc
```

resolv.conf Configuration File

A resolver finds the IP address of a host—given the host name—by using the Domain Name System (DNS). The `resolv.conf` file contains the necessary information to use DNS. Consult your local network administrator for local conventions.

The `resolv.conf` file contains three main sections:

- local domain name
- name server list
- optional domain search list

The following example `resolv.conf` listing contains these three sections.

```
# NAME RESOLUTION CONFIGURATION
#
# format: <keyword> <value>
# see keywork explanations for specific formatting
# requirements
#
# local domain name (1): domain <DomainName>
#
domain test.com
#
# ordered local nameserver list (1-3): nameserver
# <IPAddress>
#
nameserver1 192.1.1.1
nameserver2 192.1.1.2
nameserver3 192.1.1.3
nameserver4 fec0::3
#
# optional domain search list
# <Domain1> [<Domain2> ...]
#
search test.com
```

interfaces.conf Configuration File

The `interfaces.conf` file contains the hostname and interfaces to initialize when `ipstart` is run. The entry for the host name is a string and may be up to 64 characters long.

The entry for the interface list may contain the following information:

- interface name
- keyword and IP address of interface

- `broadcast` or `destaddr` keyword and broadcast or destination IP address
- `binding` keyword and device list
- optional values

```
[mtu <mtu>] [up|down] [netmask <mask>] [iff_broadcast]
[iff_pointopoint] [iff_nomulticast] [iff_nobroadcast]
[iff_nopointopoint] [prefixlen <len>]
```

The following example `interfaces.conf` entries contain the initialization information for an IPv4 SLIP device and an IPv6 Ethernet device:

- IPv4: `slip0 inet 10.0.0.1 destaddr 10.0.0.2 binding /spsl0`
- IPv6: `enet0 inet6 fec0::2374 destaddr fec0::01 prefixlen 12 binding /spde0/enet`

routes.conf Configuration File

The `routes.conf` file contains a static list of default, host, and network routes to be initialized when the stack is started. The entry for the route list contains a keyword and appropriate addresses as follows:

- default keyword and IP address of gateway network IP
- host keyword, host IP address, and gateway IP address
- network keyword, network IP address, gateway IP address, and optional network mask

There may be multiple host and network routes, but only one default route.

The following `routes.conf` entry contains the static host route entry to get to IP address 192.2.2.1 by going through router 192.1.1.2. Both IPv4 and IPv6 address styles can be used; however, you are limited in entering only one style of address per line.

```
host 192.2.2.1 192.1.1.2

net 10.0.0.0 192.2.3.3

net 172.16.40.0 192.2.3.3 255.255.255.0

host fec0::12:4 fec0::12:1 ffff::
```

rpc Configuration File

The `rpc` file contains a list of supported RPC services, associated program numbers, and the client program. Update this table to register services and program numbers with `portmap`.

The following rpc entry contains the service mapping for `rstatd`:

```
rstatd    100001    rup
```

B Error Messages

The following messages are extensions to the existing system messages and can be returned by socket access to the internet software. These messages are defined in the `errno.h` header file.

OS-9 Messages

For OS-9 for 68K, the indicated message number is constructed by separating the decimal representation of the upper and lower bytes of the error codes with a colon. For example, message number 007:001 corresponds to a hexadecimal value of 0x0701.

For OS-9, the indicated error number is constructed by separating the decimal representation of the upper and lower words of the error codes with a colon. For example, message number 007:001 corresponds to a hexadecimal value of 0x00070001.

Table B-1. Messages

Message Number	Description
007:001	EWOULDBLOCK (I/O operation would block) An operation that would cause a process to block attempted on a socket in non-blocking mode.
007:002	EINPROGRESS (I/O operation now in progress) An operation taking a long time to complete (such as connect()) attempted on a socket in non-blocking mode.
007:003	EALREADY (Operation already in progress) An operation was attempted on a non-blocking object with an operation in progress.
007:004	EDESTADDRREQ (Destination address required) The attempted socket operation requires a destination address.
007:005	EMSGSIZE (Message too long) A message sent on a socket is larger than the internal message buffer or some other network limit. Messages must be smaller than 32768 bytes.
007:006	EPROTOTYPE (Protocol wrong type for socket) A protocol is specified that does not support the semantics of the socket type requested. For example, an AF_INET UDP protocol as SOCK_STREAM is the wrong protocol type for the socket.
007:007	ENOPROTOOPT (Bad protocol option) A bad option or level is specified in getsockopt() or setsockopt().
007:008	EPROTONOSUPPORT (Protocol not supported) The requested protocol is not available or not configured for use.
007:009	ESOCKNOSUPPORT (Socket type not supported) The requested socket type is not supported or not configured for use.
007:010	EOPNOTSUPP (Operation not supported on socket) For example, accept() on a datagram socket.
007:011	EPFNOSUPPORT (Protocol family not supported)
007:012	EAFNOSUPPORT (Address family not supported by protocol)
007:013	EADDRINUSE (Address already in use) Only one use of each address is normally permitted. Wildcard use and connectionless communication are the exceptions.

Table B-1. Messages (Continued)

Message Number	Description
007:014	EADDRNOTAVAIL (Can't assign requested address) Results from an attempt to create a socket with an address not on this machine.
007:015	ENETDOWN (Network is down) The network hardware is not accessible.
007:016	ENETUNREACH (Network is unreachable) The network is unreachable. Usually caused by network interface hardware that is operational, but not physically connected to the network. This error can also be caused when the network has no way to reach the destination address.
007:017	ENETRESET (Network dropped connection on reset) The host you were connected to crashed and rebooted.
007:018	ECONNABORTED (Software caused connection abort) A connection abort was caused by the local (host) machine.
007:019	ECONNRESET (Connection reset by peer) A peer forcibly closed a connection. This normally results from a loss of the connection on the remote socket due to a time out or reboot.
007:020	ENOBUFS (No buffer space available) A socket operation could not be performed because the system lacked sufficient buffer space or a queue is full.
007:021	EISCONN (Socket is already connected) A `connect()` request was made on an already connected socket. Also caused by a `sendto()` request on a connected socket to a destination which is already connected.
007:022	ENOTCONN (Socket is not connected) A request to send or receive data is rejected because the socket is not connected or no destination is given with a datagram socket.
007:023	ESHUTDOWN (Can't send after socket shutdown)
007:024	ETOOMANYREFS (Too many references)
007:025	ETIMEOUT (Connection timed out) A `connect()` or `send()` request failed because the connected peer did not properly respond after a period of time. The time out period depends on the protocol used.
007:026	ECONNREFUSED (Connection refused by target) No connection could be established because the target machine actively refused it. This usually results from trying to connect to a service that is inactive on the target host.
007:027	EBUFTOOSMALL (mbuf too small for mbuf operation)
007:028	ESMODEXISTS (Socket module already attached)
007:029	ENOTSOCK (Path is not a socket)
007:030	EHOSTUNREACH (Host is unreachable; route not found)
007:031	EHOSTDOWN (Host is down)

C Legacy LAN Modules

This appendix lists the contents of the legacy LAN directory, BSD4, including the IP drivers and utilities.

This appendix is only helpful for users who are implementing the legacy stack. The remaining contents of this manual apply to the current stack.

The following sections are included:

Drivers

The legacy protocol drivers are located in the following directory:

`MWOS\OS9000\<processor>\CMDS\BOOTOBJS\SPF\BSD4`

The following drivers are included:

- `ip0`
- `raw0`
- `route0`
- `spip`
- `spraw`
- `sproute`
- `sptcp`
- `spudp`
- `tcp0`
- `udp0`

Utilities

The legacy utilities are located in the following directory:

`MWOS\OS9000\<processor\CMDS\BSD4`

The following utilities are included:

- `ftp`
- `ftpd`
- `ftpdc`
- `ifconfig`
- `netstat`
- `route`
- `telnet`
- `telnetd`
- `telnetdc`

Configuration Wizard Legacy Features

The following Wizard features are currently available for programmers using legacy LAN to build ROM images on their boards. Once the current LAN is used exclusively, these options will become unavailable.

- Revert to legacy stack option

  This check box can be made accessible by selecting `Configure -> Bootfile -> Network Configuration -> SoftStax Setup` from the Configuration Wizard screen.

- spf.ml

  This file is similar to the `lan.ml` file used for the current LAN. An `spf.ml` file exists for each board port relying on the legacy stack; it is located in the `BOOTS/INSTALL/PORTBOOT` directory. The contents of this file include a list of modules the Wizard can include in a bootfile.

D LAN Communications Stack Migration Reference

This appendix exists as an informational tool for customers migrating from the previously supported networking stack (that which supports the AF_INET address family) to the newly supported networking stack (that which supports both AF_INET and AF_INET6 address families).

It is recommended that you begin implementing the new stack with version 4.0 of OS-9; information regarding the previously supported stack will not be documented in subsequent releases.

Definitions

In this appendix, the following terms are used:

- "Legacy" refers to the LAN Communications stack that supports the `AF_INET` address family; this is the only network stack supported prior to OS-9 version 4.0.
- "Current" refers to the newly implemented LAN Communications stack, which supports both `AF_INET` and `AF_INET6` address families. This stack is supported for OS-9 version 4.0 or later.

References

For more information regarding the legacy LAN Communications stack or an overview of network addressing, refer to the following chapters of this manual:

- *Chapter 1, Networking Basics* (the Network Addressing section)
- *Appendix C, Legacy LAN Modules*

Migration from Legacy to Current Stack

This section discusses how the current stack has been implemented for OS-9. By default, utilities, tools, and libraries include `AF_INET6` support, where possible.

Utility Updates

Utilities for the current stack are ported with the NetBSD and OS-9 functionality in mind. The following list details how certain utilities have changed with implementation of the current stack.

- `telnet`, `telnetd`, and `telnetdc` are still based on the legacy OS-9 code, but have been enhanced to support IPv6 addresses. In addition, they each use the same command line parameters as the `AF_INET` family versions.
- `ftp`, `ftpd`, and `ftpdc` have been ported from NetBSD to OS-9 and, as such, have different command line parameters and require different system configuration. The NetBSD port includes additional functionality, including PASV support.
- `ping` is supported by the `AF_INET` family only. A new utility, `ping6`, has been created for `AF_INET6` support. `ping6` is currently ported from NetBSD sources.

The Code Base

The current stack is based on NetBSD 1.5.1 code. NetBSD IPv6 was first officially supported in version 1.5, based on the KAME Project NetBSD enhancements. The KAME Project is a joint effort to create single, solid software set, especially targeted at IPv6/Ipsec. The IPv6 code from this KAME Project was merged into NetBSD in June 1999. It is included in the NetBSD 1.5 official release.

Updates to Network Configuration Files

The current and legacy stacks contain the same IP address configuration. In addition, the configuration files live in the same directory and work for both the legacy and current LAN Communications stacks. However, when you add new IP addresses for the `AF_INET6` family, it is required that you use IPv6 keywords and addresses for the configuration files.

The network configuration is defined in a set of text configuration files that are compiled into a set of data modules, which are consumed when the network stack is started. The `AF_INET` family functionality works between legacy and current stacks.

In practice, host configuration activities are performed from within the OS-9 Configuration Wizard, which has been enhanced to support multiple Ethernet interfaces and IPv6 addressing requirements.

Router configuration is currently unsupported; thus, manual configuration is required.

Ethernet Drivers

Existing Ethernet drivers work under the current stack without any modifications. However, IPv6 Neighbor Discovery requires multicast support; if you intend on using the current stack, be sure that the drivers you are using support this feature.

The current stack uses mbuf packet chains; the legacy stack did not. The packet chain needs to be transmitted in a single Ethernet packet.

Do not loop on the packet chain to transmit separately. This will yield raw data without an IP header going out on the wire.

If the hardware does not support transmitting from multiple buffers, an enhancement to the driver to combine the mbuf packet chain to a single mbuf is recommended. If you do not have hardware driver source code to correct the mbuf packet chain issue, simply place the `spproto` driver above the hardware driver to gather an mbuf chain to a single Ethernet packet.

Miscellaneous Updates

- `netdb_local` has been eliminated. It is possible, however, to disable DNS lookup via other configuration methods.
- The legacy stack's headers and utilities have moved to subdirectories called `BSD4`. The `lancom.tpl` makefile template includes a switch to select the old or new headers. By default, compilations will use the current stack's headers and utilities at the directory locations. To use the legacy stack's headers and utilities, insert `LANCOM = 1` into the makefile definition.
- In general, legacy stack application binaries that are linked with the `netdb` trap library (`netdb.l`) should work with the current network stack; recompilation is not necessary. However, the current network stack may require additional stack space, which can be specified on the command line (e.g. `myftp foo #10`) or set permanently in the module (e.g. `fixmod -us=10 myftp`).

E Example Programs

This appendix contains a TCP and UDP socket example. Each example includes a client program and a server program. You may use these programs as templates for writing your own programs.

Source code for these examples resides in the following directory:

`MWOS/SRC/SPF/INET/EXAMPLES`

The following sections are included in this appendix:

Example One: Datagram Socket Operation for IPv4

The next two programs, `beam.c` and `target.c`, transfer data using UDP. The beam program sends a user-specified number of 1000-byte datagrams to a particular machine. By default, these are sent to UDP port 20000 but a new port may be specified using the optional third parameter to `beam`.

The `target` program binds to UDP port 20000 (or another port if specified on the command line) and receives datagrams. At the end of the transfer the number of packets and bytes received is printed. Because UDP does not guarantee end-to-end reliability of data delivery the number of packets `target` receives may be less than the number sent by `beam`.

Source code resides in the following directory:

```
MWOS/SRC/SPF/INET/EXAMPLES/AF_INET.UDP
```

beam.c

```
/* <<<<<<<<<<<<<<<<<<<<<< beam.c >>>>>>>>>>>>>>>>>>>> */
/*
** Syntax:    beam <target> <count> [<port>]
** Function:  sends packets over a datagram socket
*/

/* Header Files */
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <modes.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

/* Macro Definitions */
#define PKT_SIZE 1000/* packet size */
#define PORT20000/* udp port number */
#define START1/* packet types */
#define NORMAL2
#define END 3
```

```
/* Type Definitions */
struct packet {
    u_int32 type;
    u_int32 size;
    u_int32 count;
    char   buf[PKT_SIZE - 12];
} packet, *Packet;

void main(int argc, char* argv[], char* envp[])
{
    int s;
    int i;
    int count;
    struct hostent *host;
    struct sockaddr_in sockname;
    static struct packet pkt;
    /* check for proper number of arguments */
    if ((argc < 3) || (argv[1][0] == '-')) {
        printf("usage: beam <hostname|ip-address> <count> [<port>]\n");
        exit(0);
    }
    /* get number of packets to beam */
    count = atoi(argv[2]);
    /* open up datagram (UDP) socket */
    memset(&sockname, 0, sizeof(sockname));
    sockname.sin_family = AF_INET;
    sockname.sin_port = 0;
    sockname.sin_addr.s_addr = INADDR_ANY;
    if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "beam: socket call failed\n");
        exit(errno);
    }
    /* bind socket (let system pick our port number) */
    if (bind(s, (struct sockaddr*)&sockname, sizeof(sockname)) < 0) {
        fprintf(stderr, "bind failed to host\n");
        _os_close(s);
        exit(errno);
    }
```

```
/* get information concerning the host we'd like to beam to */
sockname.sin_port = 0;
sockname.sin_addr.s_addr = INADDR_ANY;
if ((host = gethostbyname(argv[1])) != (struct hostent *)0) {
    sockname.sin_family = host->h_addrtype;
    memcpy(&sockname.sin_addr.s_addr, host->h_addr, 4);
} else {
    u_int32 addr = inet_addr(argv[1]);
    sockname.sin_family = AF_INET;
    memcpy(&sockname.sin_addr.s_addr, &addr, 4);
}
endhostent();
if (argc > 3) {
    sockname.sin_port = htons(atoi(argv[3]));
} else {
    sockname.sin_port = htons(PORT);
}
/* set up socket for send */
#ifdef USE_CONNECT
    /* connected UDP socket -- we're only talking to this host */
    if (connect(s, (struct sockaddr *)&sockname, sizeof(sockname))
                < 0) {
        fprintf(stderr, "beam: cannot connect\n");
        _os_close(s);
        exit(errno);
    }
#endif
printf("beaming...\n");
/* set up packets for transfer and transfer them all */
pkt.size = htonl(PKT_SIZE);
for (i = 0; i <= count; i++) {
    if (i == 0) {
        pkt.type = htonl(START);
    } else if (i >= count) {
        pkt.type = htonl(END);
    } else {
        pkt.type = htonl(NORMAL);
    }
```

```
            pkt.count = htonl(i);
            /* send data to target */
            #ifdef USE_CONNECT
                if (send(s, &pkt, ntohl(pkt.size), 0) < 0) {
                    fprintf(stderr, "beam: send failed\n");
                    _os_close(s);
                    exit(errno);
                }
            #else
                if (sendto(s, (char*)&pkt, ntohl(pkt.size), 0,
                        (struct sockaddr*)&sockname, sizeof(sockname)) < 0) {
                    fprintf(stderr, "beam: sendto failed\n");
                    _os_close(s);
                    exit(errno);
                }
            #endif
        }
        _os_close(s);
        exit(0);
    }   /* end of main */
```

target.c

```
/* <<<<<<<<<<<<<<<<<<<<<<< target.c >>>>>>>>>>>>>>>>>>> */
/*
** Syntax:target
** Function:receives packets from beam over a datagram
**               socket and displays number of packets
*/

/* Header Files */
#include <stdio.h>
#include <stdlib.h>
#include <types.h>
#include <errno.h>
#include <string.h>
#include <modes.h>
#include <ctype.h>
#include <sys/types.h>
```

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

/* Macro Definitions */
#define PKT_SIZE 1000/* packet size */
#define PORT 20000/* udp port number */
#define START 1/* packet types */
#define NORMAL 2
#define END 3

/* Type Definitions */
struct packet {
   u_int32type;
   u_int32size;
   u_int32count;
   char   buf[PKT_SIZE - 12];
} packet, *Packet;

/* main() : initial program entry point */
void main(int argc, char* argv[], char* envp[])
{
   int s;
   int count;
   socklen_t namelen;
   int packetsrecv;
   u_int32 bytesrecv;
   static struct packet pkt;
   struct sockaddr_in name;
   if ((argc < 1) || (argc > 2) || ((argc == 2) &&
         (!isdigit(argv[1][0])))) {
      printf("usage: target [<port>]\n");
      exit(0);
   }
   /* open up datagram (UDP) socket */
   if ((s = socket(AF_INET, SOCK_DGRAM, 0)) == -1) {
      fprintf(stderr, "target: socket failed\n");
      exit(errno);
```

```
}
/* bind socket (pick proper "well-known" port number) */
memset(&name, 0, sizeof(name));
name.sin_family = AF_INET;
name.sin_addr.s_addr = INADDR_ANY;
if (argc >= 2) {
    name.sin_port = htons(atoi(argv[1]));
} else {
    name.sin_port = htons(PORT);
}
if (bind(s, (struct sockaddr*)&name, sizeof(name)) == -1) {
    fprintf(stderr, "target: bind failed to port
        '%d'\n",ntohs(name.sin_port));
    _os_close(s);
    exit(errno);
}
printf("Waiting for packets...\n");
/* wait for start packet */
while (1) {
    /* get a packet (and find out who sent it to us) */
    namelen = sizeof(name);
    if ((count = recvfrom(s, (char*)&pkt, sizeof(pkt), 0,
              (struct sockaddr*)&name,&namelen)) == -1) {
        fprintf(stderr, "target: recv failed\n");
        _os_close(s);
        exit(errno);
    }
    if (pkt.type != htonl(START)){
        printf("out of sequence packet received\n");
        continue;
    } else {
        break;
    }
}
bytesrecv = packetsrecv = 0;
/* loop until all packet are receved */
printf("Begin transfer\n");
do {
```

```
            namelen = sizeof(name);
            if ((count = recvfrom(s, (char*)&pkt, sizeof(pkt), 0,
                      (struct sockaddr*)&name,&namelen)) == -1) {
               fprintf(stderr, "target: recv failed\n");
               _os_close(s);
               exit(errno);
            }
            bytesrecv += count;
            packetsrecv++;
         } while (pkt.type == ntohl(NORMAL));
         /* if we didn't get and END packet, print error */
         if (pkt.type != ntohl(END)) {
            printf("expected an END packet\n");
         }
         /* print out summary */
         printf("Transfer complete\n");
         printf("    Packets received:  %d\n",packetsrecv);
         printf("    Bytes received:  %d\n",bytesrecv);
         /* cleanup and exit */
         _os_close(s);
         exit(0);
      }  /* end of main */
```

Example Two: Datagram Socket Operation for IPv6

The following programs, `beam6.c` and `target6.c`, transfer data using UDP on IPv6 addresses. The operation is similar to that of IPv4, with the exception of the IP version.

Source code for these files resides in the following directory:

`MWOS/SRC/SPF/INET/EXAMPLES/AF_INET.UDP6`

beam6.c

```
#if defined(_OSK) || defined(_OS9000)
_asm("_sysedit: equ 1");
#endif

/* <<<<<<<<<<<<<<<<<<<<<< beam.c >>>>>>>>>>>>>>>>>>>> */
/*
** Syntax:    beam <target> <count> [<port>]
** Function:  sends packets over a datagram socket
*/

/* Header Files */

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#if defined(_OSK) || defined(_OS9000)
#include <modes.h>
#endif

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

/* Macro Definitions */

#define PKT_SIZE1000/* packet size */
#define PORT20000/* udp port number */

#define START1   /* packet types */
#define NORMAL2
#define END3

#if !(defined(_OSK) || defined(_OS9000))
typedef unsigned int u_int32;
```

```
#define _os_close close
#endif

const struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;

/* Type Definitions */

struct packet {
    u_int32type;
    u_int32size;
    u_int32count;
    char   buf[PKT_SIZE - 12];
} packet, *Packet;


/* main() : initial program entry point */
void main(int argc, char* argv[], char* envp[])
{
    int s;
    int i;
    int count;
    struct hostent *host;
    struct sockaddr_in6 sockname;
    static struct packet pkt;

    /* check for proper number of arguments */
    if ((argc < 3) || (argv[1][0] == '-')) {
        printf("usage: beam <hostname|ip-address> <count> [<port>]\n");
        exit(0);
    }

    /* get number of packets to beam */
    count = atoi(argv[2]);

    /* open up datagram (UDP) socket */
    memset(&sockname, 0, sizeof(sockname));
    sockname.sin6_family = AF_INET6;
    sockname.sin6_port = 0;
```

```
    sockname.sin6_addr = in6addr_any;
    if ((s = socket(AF_INET6, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "beam: socket call failed\n");
        exit(errno);
    }

    /* bind socket (let system pick our port number) */
    if (bind(s, (struct sockaddr*)&sockname, sizeof(sockname)) < 0) {
        fprintf(stderr, "bind failed to host\n");
        _os_close(s);
        exit(errno);
    }

    /* get information concerning the host we'd like to beam to */
    sockname.sin6_port = 0;
    sockname.sin6_addr = in6addr_any;
    if ((host = (struct hostent *)gethostbyname2(argv[1], AF_INET6)) != 0
) {
        if( host->h_addrtype != AF_INET6 ){
            fprintf(stderr, "not support IPv4\n");
            _os_close(s);
            exit(errno);
        }
        sockname.sin6_family = host->h_addrtype;
        memcpy(&sockname.sin6_addr.s6_addr, host->h_addr, sizeof(struct
in6_addr));
    } else {
        if( !inet_pton(AF_INET6, argv[1], &sockname.sin6_addr) ) {
            fprintf(stderr, "not a valid presentation format\n");
            _os_close(s);
            exit(errno);
        }
        sockname.sin6_family = AF_INET6;
    }
    endhostent();
    if (argc > 3) {
        sockname.sin6_port = htons(atoi(argv[3]));
```

```
    } else {
        sockname.sin6_port = htons(PORT);
    }

    /* set up socket for send */

    #ifdef USE_CONNECT
        /* "connect" UDP socket -- we're only going to talk to this host */
        if (connect(s, (struct sockaddr *)&sockname, sizeof(sockname)) < 0)
{
            fprintf(stderr, "beam: cannot connect\n");
            _os_close(s);
            exit(errno);
        }
    #endif

    printf("beaming...\n");

    /* set up packets for transfer and transfer them all */
    pkt.size = htonl(PKT_SIZE);
    for (i = 0; i <= count; i++) {

        if (i == 0) {
            pkt.type = htonl(START);
        } else if (i >= count) {
            pkt.type = htonl(END);
        } else {
            pkt.type = htonl(NORMAL);
        }
        pkt.count = htonl(i);

        /* send data to target */
        #ifdef USE_CONNECT
            if (send(s, &pkt, ntohl(pkt.size), 0) < 0) {
                fprintf(stderr, "beam: send failed\n");
                _os_close(s);
                exit(errno);
            }
```

```
        #else
            if (sendto(s, (char*)&pkt, ntohl(pkt.size), 0,
                    (struct sockaddr*)&sockname, sizeof(sockname)) < 0) {
                fprintf(stderr, "beam: sendto failed\n");
                _os_close(s);
                exit(errno);
            }
        #endif
    }

    /* cleanup and exit */
    _os_close(s);
    exit(0);

}   /* end of main */
```

target6.c

```
#if defined(_OSK) || defined(_OS9000)
_asm("_sysedit: equ 1");
#endif

/* <<<<<<<<<<<<<<<<<<<<<< target.c >>>>>>>>>>>>>>>>>>> */
/*
** Syntax:target
** Function:receives packets from beam over a datagram
** socket and displays number of packets
*/

/* Header Files */

#include <stdio.h>
#include <stdlib.h>
#if defined(_OSK) || defined(_OS9000)
#include <types.h>
#endif
#include <errno.h>
#include <string.h>
#if defined(_OSK) || defined(_OS9000)
```

```
#include <modes.h>
#include <ctype.h>
#endif
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

#if !(defined(_OSK) || defined(_OS9000))
typedef unsigned int u_int32;
#define _os_close close
#endif

/* Macro Definitions */

#define PKT_SIZE1000/* packet size */
#define PORT20000/* udp port number */

#define START1   /* packet types */
#define NORMAL2
#define END3


/* Type Definitions */

struct packet {
   u_int32type;
   u_int32size;
   u_int32count;
   char   buf[PKT_SIZE - 12];
} packet, *Packet;

const struct in6_addr in6addr_any = IN6ADDR_ANY_INIT;

/* main() : initial program entry point */
void main(int argc, char* argv[], char* envp[])
```

```
{
    int s;
    int count;
    socklen_t namelen;
    int packetsrecv;
    u_int32 bytesrecv;
    static struct packet pkt;
    struct sockaddr_in6 name;

if ((argc < 1) || (argc > 2) || ((argc == 2) && (!isdigit(argv[1][0])))) {
        printf("usage: target [<port>]\n");
        exit(0);
    }

    /* open up datagram (UDP) socket */
    if ((s = socket(AF_INET6, SOCK_DGRAM, 0)) == -1) {
        fprintf(stderr, "target: socket failed\n");
        exit(errno);
    }

    /* bind socket (pick proper "well-known" port number) */
    memset(&name, 0, sizeof(name));
    name.sin6_family = AF_INET6;
    name.sin6_addr = in6addr_any;

    if (argc >= 2) {
        name.sin6_port = htons(atoi(argv[1]));
    } else {
        name.sin6_port = htons(PORT);
    }
    if (bind(s, (struct sockaddr*)&name, sizeof(name)) == -1) {
        fprintf(stderr, "target: bind failed to port
'%d'\n",ntohs(name.sin6_port));
        _os_close(s);
        exit(errno);
    }

    printf("Waiting for packets...\n");
```

```
/* wait for start packet */
while (1) {

    /* get a packet (and find out who sent it to us) */
    namelen = sizeof(name);
    if ((count = recvfrom(s, (char*)&pkt, sizeof(pkt), 0,
              (struct sockaddr*)&name,&namelen)) == -1) {
        fprintf(stderr, "target: recv failed\n");
        _os_close(s);
        exit(errno);
    }

    if (pkt.type != htonl(START)){
        printf("out of sequence packet received\n");
        continue;
    } else {
        break;
    }
}
bytesrecv = packetsrecv = 0;

/* loop until all packet are receved */
printf("Begin transfer\n");
do {
    namelen = sizeof(name);
    if ((count = recvfrom(s, (char*)&pkt, sizeof(pkt), 0,
              (struct sockaddr*)&name,&namelen)) == -1) {
        fprintf(stderr, "target: recv failed\n");
        _os_close(s);
        exit(errno);
    }
    bytesrecv += count;
    packetsrecv++;
} while (pkt.type == ntohl(NORMAL));

/* if we didn't get and END packet, print error */
if (pkt.type != ntohl(END)) {
    printf("expected an END packet\n");
```

```
    }

    /* print out summary */
    printf("Transfer complete\n");
    printf("    Packets received:  %d\n",packetsrecv);
    printf("    Bytes received:  %d\n",bytesrecv);

    /* cleanup and exit */
    _os_close(s);
    exit(0);

}   /* end of main */
```

Example Three: Stream Socket

The following programs, `tcpsend.c` and `tcprecv.c` use TCP to reliably transfer a file using either IPv4 or IPv6 as a transport. The `tcpsend` program connects to TCP port 27000 on the target machine and sends the specified file. The version of IP to use is determined by the type of address provided in the hostname parameter. If a machine name is specified rather than a numeric address, the first address returned from the resolver will be used. Blocking I/O will be used unless the `-n` parameter is specified.

The `tcprecv` program binds a socket to port 27000 and listens for incoming data connections. By default, IPv4 is used as a transport. The
-4 or -6 options may be used to force a version of IP to be used. When a connection occurs, the data is read from the network and written to the output file specified on the command line. Blocking I/O will be used unless the `-n` parameter is specified.

Source code resides in the following directory:

```
MWOS/SRC/SPF/INET/EXAMPLES/AF_INET.TCP
```

tcpsend.c

```
/* <<<<<<<<<<<<<<<<<<<<<<<<<<<< tcpsend.c >>>>>>>>>>>>>>>>>>>>>>>>>>>> */
/*
** Syntax: tcpsend [<opts>] <hostname> <filename>
** Function:  sends a file across a stream (TCP) socket
*/

/* Header Files */
#define _OPT_PROTOS

#include <stdio.h>
```

```
#include <stdlib.h>
#include <const.h>
#include <signal.h>
#include <types.h>
#include <errno.h>
#include <string.h>
#include <modes.h>
#include <sg_codes.h>

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/errno.h>
#include <sys/ioctl.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <netdb.h>
#include <SPF/spf.h>
#include <UNIX/ioctl.h>

/* Macro Definitions */

#define PORT_NUM"27000"
#define MAX_LOOPS200
#define FMODES_IREAD

#if defined(_OSK)
   #define _os_sleep(t,s) _os9_sleep(t)
#else
   signal_code sig = 0;
#endif

/* Type Definitions */
struct data {
   int code, count;
   char data[512];
};
```

```
/* Global Variables */
struct sockaddr_in ls_addr;
char msgbuf[64000];
char *ptr;

/* Function Prototypes */
void main(int, char **, char **);
void usage(void);

void main(int argc, char* argv[], char* envp[])
{
    int s;
    int flags = 0;
    int totbytes = 0;
    int noblock = 0;
    path_id ifile;
    u_int32 vflag = 0;
    u_int32 count;
    u_int32 wcount;
    u_int32 wsize;
    u_int32 tries;
    u_int32 tics;
    struct addrinfo hints;
    struct addrinfo *ai;
    error_code error;
    char *hostname = NULL;
    char *filename = NULL;
    while (--argc > 0) {
        if (*(ptr = *++argv) == '-') {
            while (*++ptr) {
                switch (*ptr|0x20) {
                    case 'n':
                    noblock = IO_ASYNC;
                    break;

                    case 'v':
                    vflag = 1;
                    break;
```

```
                case '?':
                default:
                usage();
            }
        }
    } else {
        if (hostname == NULL) {
            hostname = *argv;
        } else {
            if (filename == NULL) {
                filename = *argv;
            } else {
                usage();
            }
        }
    }
}

if (filename == NULL) {
    usage();
}

if ((errno = _os_open(filename,FMODE,&ifile)) != SUCCESS) {
    fprintf(stderr, "can't open file '%s'\n", filename);
    exit(errno);
}

memset (&hints, 0, sizeof(hints));
hints.ai_socktype = SOCK_STREAM;
if (error = getaddrinfo(hostname, PORT_NUM, &hints, &ai)) {
    fprintf(stderr, "getaddrinfo failed\n");
    _os_close(ifile);
    exit(error);
}
```

```
    if ((s = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol)) < 0)
{
        perror("socket");
        _os_close(ifile);
        exit(errno);
    }

    if (noblock) {
        printf("using non-blocking sockets\n");
        if (ioctl(s, FIONBIO, (caddr_t)&noblock)) {
            fprintf(stderr, "can't set socket nonblocking\n");
            _os_close(ifile);
            _os_close(s);
            exit(errno);
        }
    } else {
        printf("using blocking sockets\n");
    }

    if (noblock) {
        /*
        ** Non-blocking connect
        */
        tries = MAX_LOOPS;
        while (tries) {
            if (connect(s, ai->ai_addr, ai->ai_addrlen) < 0) {
                if (errno == EISCONN) {
                    break;
                }
                if (errno == EINVAL) {
                    int error;
                    u_int32 len;
                    len = sizeof(error);
                    if (getsockopt(s, SOL_SOCKET, SO_ERROR, &error,
                    &len) < 0) {
                    error = EINVAL;
                    }
                    errno = error;
```

```
                perror("connect");
                _os_close(s);
                _os_close(ifile);
                exit(1);
            }
            tics = 0x80000080; /* 1/2 second */
            _os_sleep(&tics, &sig);
        } else {
            break;
        }
        tries--;
    }

    if (tries == 0){
        errno = ETIMEDOUT;
        perror("connect");
        _os_close(s);
        _os_close(ifile);
        exit(1);
    }
} else {
    /*
    ** Blocking connect
    */
    if (connect(s, ai->ai_addr, ai->ai_addrlen) < 0) {
        perror("connect");
        _os_close(s);
        _os_close(ifile);
        exit(1);
    }
}
freeaddrinfo(ai);
printf("Connection established\n");
printf("Sending file '%s'...\n", filename);
```

```
if (noblock) {
    /*
    ** Non-blocking send
    */
    count = sizeof(msgbuf);
    while ((errno = _os_read(ifile,msgbuf,&count)) == SUCCESS) {
        wcount = 0;
        ptr = msgbuf;
        while (wcount < count){
            wsize = count - wcount;

            if ((wsize = send(s,ptr,wsize,flags)) == (u_int32)-1) {
                if (errno != EWOULDBLOCK) {
                fprintf(stderr, "socket write error\n");
                _os_close(s);
                _os_close(ifile);
                exit(errno);
                } else {
                if (vflag) printf("Write would block, sleeping...");
                tics = 0x80000040; /* 1/4 second */
                _os_sleep(&tics,&sig);
                if (vflag) printf("trying write again.\n");
                }
            } else {
                wcount += wsize;
                ptr += wsize;
                if (vflag) printf("wrote %d Bytes\n", wsize);
            }
        }
        totbytes += count;
        count = sizeof(msgbuf);
    }
    if (errno != EOS_EOF){
        fprintf(stderr, "read error on file\n");
        exit(errno);
    }
```

```
} else {
    /*
    ** Blocking send
    */
    count = sizeof(msgbuf);
    while ((errno = _os_read(ifile, msgbuf, &count)) == SUCCESS) {
        wcount = count;
        ptr = msgbuf;
        while ((count = send(s, ptr, count, flags)) != (u_int32)-1) {
            totbytes += count;
            if (count == wcount) {
                break;
            }
            /*
            ** For some reason all the data was not written,
            ** loop around and try to write the rest.
            */
            ptr += count;
            count = wcount - count;
            wcount = count;
        }
        if (errno){
            fprintf(stderr, "socket write error\n");
            _os_close(s);
            _os_close(ifile);
            exit(errno);
        }
        count = sizeof(msgbuf);
    }
    if (errno != EOS_EOF){
        fprintf(stderr, "read error on file\n");
        exit(errno);
    }
}

_os_close(s);
_os_close(ifile);
printf("sent %u bytes\n", totbytes);
```

```
    exit(0);
}

void usage()
{
    fprintf(stderr, "Syntax: tcpsend [<opts>] <hostname> <filename>\n"
                "Options:\n"
                "    -n   use non blocking sockets\n"
                "    -v   enable verbose output\n");
    exit(0);
}
```

tcprecv.c

```
/* <<<<<<<<<<<<<<<<<<<<<<<<< tcprecv.c >>>>>>>>>>>>>>>>>>>>>>>> */
/*
** Syntax: tcprecv [<opts>] <filename>
** Function:  receives a file across a stream (TCP) socket
*/

/* Header Files */
#define _OPT_PROTOS

#include <stdio.h>
#include <stdlib.h>
#include <const.h>
#include <signal.h>
#include <types.h>
#include <errno.h>
#include <string.h>
#include <modes.h>
#include <sg_codes.h>

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

```
#include <netdb.h>
#include <SPF/spf.h>
#include <UNIX/ioctl.h>

/* Macro Definitions */

#define PORT_NUM"27000"

#define INIT77/* commands */
#define DATA78
#define END79
#define FMODES_IWRITE

#if defined(_OSK)
   #define _os_sleep(t,s) _os9_sleep(t)
#else
    signal_code sig = 0;
#endif

/* Global Variables */
char msgbuf[20480];
char hostname[NI_MAXHOST];
char service[NI_MAXSERV];

/* Function Prototypes */
void main(int, char **, char **);
void usage(void);

void main(int argc, char* argv[], char* envp[])
{
   int s;
   int sx;
   u_int32 size;
   int totbytes = 0;
   int noblock = 0;
   path_id ofile;
   u_int32 vflag = 0;
   u_int32 count = 1;
```

```
u_int32 tics;
int family = AF_INET;
error_code error;
struct addrinfo hints;
struct addrinfo *ai;
struct sockaddr_storage peer;
char *filename = NULL;
char *ptr;

while (--argc > 0) {
    if (*(ptr = *++argv) == '-') {
        while (*++ptr) {
            switch (*ptr|0x20) {
                case '4':
                family = AF_INET;
                break;

                case '6':
                family = AF_INET6;
                break;

                case 'n':
                noblock = IO_ASYNC;
                break;

                case 'v':
                vflag = 1;
                break;

                case '?':
                default:
                usage();
            }
        }
```

```
            } else {
                if (filename == NULL) {
                    filename = *argv;
                } else {
                    usage();
                }
            }
        }

        if (filename == NULL) {
            usage();
        }

        if ((error = _os_create(filename, FMODE, &ofile,
                    S_IREAD|S_IWRITE)) != SUCCESS) {
            fprintf(stderr, "can't open file '%s'\n", filename);
            exit(error);
        }

        memset (&hints, 0, sizeof(hints));
        hints.ai_family = family;
        hints.ai_flags = AI_PASSIVE;
        hints.ai_socktype = SOCK_STREAM;
        if (error = getaddrinfo(NULL, PORT_NUM, &hints, &ai)) {
            fprintf(stderr, "getaddrinfo failed\n");
            _os_close(ofile);
            exit(errno);
        }

        if ((sx = socket(ai->ai_family, ai->ai_socktype, ai->ai_protocol)) <
0) {
            fprintf(stderr,"can't open socket\n");
            _os_close(ofile);
            exit(errno);
        }

        if (bind(sx, ai->ai_addr, ai->ai_addrlen) < 0) {
            fprintf(stderr,"can't bind socket\n");
```

```
        _os_close(sx);
        _os_close(ofile);
        exit(errno);
    }

    if (listen(sx, 1) < 0) {
        fprintf(stderr,"tcp_listen - failed!\n");
        _os_close(sx);
        _os_close(ofile);
        exit(errno);
    }

    size = sizeof(peer);
    if ((s = accept(sx, (struct sockaddr *)&peer, &size)) < 0) {
        fprintf(stderr, "accept failed\n");
        _os_close(s);
        _os_close(sx);
        _os_close(ofile);
        exit(errno);
    }
    _os_close (sx);
    freeaddrinfo(ai);

    if ((error = getnameinfo((struct sockaddr *)&peer, peer.ss_len,
hostname,
                sizeof(hostname), service, sizeof(service), 0)) != 0) {
        fprintf(stderr, "getnameinfo failed\n");
        _os_close(s);
        _os_close(sx);
        _os_close(ofile);
        exit(error);
    }

    printf("connected to %s, port %s\n", hostname, service);
```

```
    if (noblock) {
        printf("using non-blocking sockets\n");
        if (ioctl(s, FIONBIO, (caddr_t)&noblock)) {
            fprintf(stderr, "can't set socket nonblocking\n");
            _os_close(ofile);
            _os_close(s);
            _os_close(sx);
            exit(errno);
        }
    } else {
        printf("using blocking sockets\n");
    }

    do {
        count = sizeof(msgbuf);
        if ((count = recv(s,msgbuf,count, 0)) == (u_int32) -1) {

            if (errno == EOS_EOF) {
                break; /*at end of file*/
            }
            if (noblock & (errno == EWOULDBLOCK)) {
                if (vflag)
                    printf("Total Bytes read: %d. Read would block,
sleeping...",
                    totbytes);
                tics = 0x80000040; /* 1/4 second */
                _os_sleep(&tics, &sig);
                count = 1;/* force one more interation */
                if (vflag) printf("trying read again.\n");
            } else {
                fprintf(stderr, "can't recv (cnt=%d)\n", count);
                exit(errno);
            }
        } else if (count > 0) {
            if ((errno = _os_write(ofile, msgbuf, &count)) != SUCCESS) {
                fprintf(stderr, "can't write output\n");
                exit(errno);
            }
```

```
            totbytes += count;
        }
    } while (count);

    _os_close(s);
    _os_close(ofile);
    printf("read %d bytes\n", totbytes);

    exit(0);
}

void usage(void)
{
    fprintf(stderr, "Syntax: tcprecv [<opts>] <filename>\n"
                "Options:\n"
                "    -4    Use IPv4 addresses (default)\n"
                "    -6    Use IPv6 addresses\n"
                "    -n    Use non blocking sockets\n"
                "    -v    Verbose mode\n");
    exit(0);
}
```

Example Four: Sending Multicast Messages

The example programs `msend` and `mrecv` send multicast messages over the network, demonstrating the receipt of packets by multiple destinations. Since multicasting is a connectionless protocol, `mrecv` may not see every packet sent.

The `msend` program will send either the message entered at the command line or a default message once, then exit. The signal handler in `mrecv` processes any system signal as a signal to kill the process. The preferred method for executing this is to use <Ctrl C> or <Ctrl E> from the keyboard.

`msend` requires a route in the IP routing table that will return as a match for the multicast group being used. This is required even if the interface is explicitly named using the `-i` option. For example, consider the following routing table:

```
$ netstat -rn
```

Table E-1. Internet

Destination	Gateway	Flags	Refs	Use	Interface
127.0.0.1	127.0.0.1	UH	0	1	lo0
172.16	172.16.2.226	U	1	122	enet1
172.16.2.226	127.0.0.1	UHS	0	0	lo0

Table E-1. Internet (Continued)

Destination	Gateway	Flags	Refs	Use	Interface
192.168.3	192.168.3.19	U	0	0	enet0
192.168.3.19	127.0.0.1	UHS	0	0	lo0

Using `msend` to send to group `225.0.0.172` will result in an error since the routing table contains no route that will match that IP address. Running either of the following route commands will allow it to work:

```
route add -net default 172.16.2.250
```

-OR-

```
route add -net 225.0.0.0 172.16.2.226 240.0.0.0
```

Now `msend` will successfully transmit the packet on the `enet1` interface. If a '`-i 192.168.3.19`' command line option is added, the packet will be sent from the `enet0` interface instead.

Source code for both of these files is in the following directory:

```
MWOS/SRC/SPF/INET/EXAMPLES/MULTICAST
```

msend

Send multicast packet.

Syntax

```
msend [<opts>]
```

Options

```
[-v] [-l] [-t ttl] [-p port] [-g group] [-i interface] [-m message]
```

`-v`	Enable verbose mode. (Default: off)
`-l`	Enable loopback reception of packet. (Default: off)
`-t`	Set TTL of output packets. (Default: 1)
`-p`	Set port of output packets. (Default: 4433)
`-g`	Select destination group. (Default:225.0.0.172)
`-i`	Select outgoing interface. (Default: route table lookup of group)
`-m`	Select message to send. (Default: "This is a test message")

mrecv

Receive multicast packet.

Syntax

```
mrecv [<opts>]
```

Options

```
[-v] [-p port] [-g group] [-i interface]
```

`-v`	Enable verbose mode. (Default: off)
`-p`	Set port for selecting incoming packets. (Default: 4433)
`-g`	Select destination group. (Default: 225.0.0.172)
`-i`	Select receiving interface. (Default: route table lookup of group)

F Dynamic Configuration of the inetdb Module

This appendix contains the folloiwng section:

Sample inetdb Module

The `netdb.l` and `ndblib.l` libraries provide functions that enable direct updating of the `inetdb` module.

To create an `inetdb` module through an application, the function `ndb_create_ndbmod()` is used. (See library function for syntax.)

The following example creates an `inetdbX` module with room for additional entries:

```
/* <<<<<<<<<<<<<<<<<<< ndbcreate.c >>>>>>>> */
/*
**      Syntax:    ndbcreate <filename>
**      Function:  Create an inetdbX data module
*/
#include <stdio.h>
#include <errno.h>
#include <netdblib.h>

/*
** The following is the creation size array for the configuration
** files.
**
** hosts: 100 bytes (Up to 4 entries)
** hosts.equiv: 0 (not used)
** networks: 80 bytes (Up to 2 entries)
** protocols: 50 bytes (Up to 2 entries)
** services: 50 bytes (Up to 2 entries)
** inetd.conf: 50 bytes (Up to 1 entry)
** resolv.conf: 100 bytes (Up to 1 entry)
** host.config: 0 (not used)
** interfaces.conf: 400 bytes (Up to 2 entries)
** hostname string: 65 bytes (Up to 65 characters)
** routes.conf: 128 bytes (Up to 2 entries)
*/
int size_array[11] = { 100, 0, 80, 50, 50, 50, 100, 0, 400, 65, 128 };

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

void main(int argc, char* argv[], char* envp[])
{
```

```
    char *mod_name;
    int num_files;
    int perm =
         MP_OWNER_READ|MP_OWNER_WRITE|MP_GROUP_READ|MP_WORLD_READ;


    if (argc == 1) {
        fprintf(stderr, "Module name required (e.g. inetdb2)\n");
        exit(0);
    }
    mod_name = argv[1];
    num_files = 11; /* Create data module with configuration
                  file space */
    if (errno = ndb_create_ndbmod(mod_name, num_files, size_array,
            perm, 0)) {
        fprintf(stderr, "Can't create data module %s\n",mod_name);
        exit(errno);
    }
    exit(0);
}
```

The configuration entries can be updated with the following functions:

- hosts: `puthostent()`, `delhostent()`
- networks: `putnetent()`, `delnetent()`
- protocols: `putprotoent()`, `delprotoent()`
- services: `putservent()`, `delservent()`
- resolv.conf: `putresolvent()`, `delresolvent()`
- interfaces.conf: `putintent()`, `delintent()`
- hostname: `sethostname()`
- routes.conf: `putroutent()`, `delroutent()`

The routes and interfaces entries must be added before the IP stack is initialized. IP reads these entries only at that time. All other entries can be updated after the stack has been brought up.

Manipulating a Host Entry

Following is an example of how to manipulate a host entry:

```
/* <<<<<<<<<<<<<<<<<<<< hostent.c >>>>>>>>>>>>>>>>>>>>>>>>>>>> */
/*
**      Syntax:    hostent
**      Function:  Manipulate an inetdbX data module host entry
*/

#include <stdio.h>
#include <errno.h>
#include <netdb.h>

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

void main(int argc, char* argv[], char* envp[])
{
    struct hostent host, *gethost; /* host structure in inetdb */
    static struct in_addr addr; /* IP address of host structure */
    static struct in_addr *h_addrs[2] = {&addr,0}; /* List of IP
                addresses */
    static char *h_aliases[3] = {"alpha", "beta", 0};
    /*fill in the host entry*/
    addr.s_addr = inet_addr("10.0.0.1");
    host.h_name = "gamma";
    host.h_aliases = h_aliases;
    host.h_addr_list = (char **)h_addrs;
    host.h_length = sizeof(struct in_addr);
    host.h_addrtype = AF_INET;

    /*Insert Host Entry in InetdbX module*/
    if ((errno = puthostent(&host)) == -1) {
        fprintf(stderr,"Error in puthostent!\n");
        exit(errno);
    }

    /*Read Host Entry in InetdbX module*/
    if ((gethost = gethostbyname(host.h_name)) == NULL) {
```

```
        fprintf(stderr,"Error in gethostbyname!\n");
        exit(errno);
    }
    printf("Obtained host entry %s\n",gethost->h_name);

    /*Delete Host Entry in InetdbX module*/
    if ((errno = delhostbyname(host.h_name)) == -1) {
        fprintf(stderr,"Error in delhostbyname!\n");
        exit(errno);
    }
    exit(0);
}
```

Changing the DNS Client Entry

Following is an example of how to change the DNS client entry:

```
/* <<<<<<<<<<<<<<<<<<< resolvent.c >>>>>>>>>>>>>>>>>>>>>>>>>>>> */
/*
**      Syntax:    resolvent
**      Function:  Manipulate an inetdbX data module DNS client entry
*/
#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <resolv.h>
void main(int argc, char* argv[], char* envp[]);

void main(int argc, char* argv[], char* envp[])
{
    struct resolvent res, *getres; /* DNS client resolve structure in
                  inetdb */

    memset(&res, 0, sizeof(res));
    res.domain = "alphatest.com";
    res.nameservers[0] = "10.0.0.1";
    res.nameservers[1] = "10.0.0.2";
    res.search[0] = "alphatest.com";
    res.search[0] = "betatest.com";
    res.search[0] = "gammatest.com";
```

```
    /*Insert Resolve Entry in InetdbX module*/
    if ((errno = putresolvent(&res)) == -1) {
        fprintf(stderr,"Error in putresolvent!\n");
        exit(errno);
    }

    /*Read Resolve Entry in InetdbX module*/
    if ((getres = getresolvent()) == NULL) {
        fprintf(stderr,"Error in getresolvent!\n");
        exit(errno);
    }
    printf("Obtained resolve entry for domain %s\n",getres->domain);

    /*Delete Resolve Entry in InetdbX module*/
    if ((errno = delresolvent()) == -1) {
        fprintf(stderr,"Error in delresolvent!\n");
        exit(errno);
    }
    exit(0);
}
```

Adding an Interface Entry

Following is an example of how to add an interface entry:

These functions must be called before `ipstart` is run.

```
/* <<<<<<<<<<<<<<<<<<<<<<< interface.c >>>>>>>>>>>>>>>>>>>>>>> */
/*
**  Syntax:   interface
**  Function: Manipulate an inetdbX data module hostname and interface
*/
#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <net/if.h>

void main(int argc, char* argv[], char* envp[]);
```

```
void main(int argc, char* argv[], char* envp[])
{
    struct n_ifnet ifnet, *getint; /*Structure to add Interface*/
    struct n_ifaliasreq ifalias[1];
    struct sockaddr_in *sock_int;

    memset(&ifnet, 0, sizeof(ifnet));
    memset(&ifalias, 0, sizeof(ifalias));

    /*Build interface entry */
    strcpy(ifnet.if_name, "enet0");      /* Interface Name */
    strcpy(ifnet.if_stack_name,
                "/sple0/enet");    /* Device Path to open */
    ifnet.if_flags = IFF_UP|IFF_BROADCAST       /* Device Flags */
    /*ifnet.mw_flags = IFF_NOMULTICAST;*//* Override driver default */
    strcpy(ifalias[0].ifra_name, "enet0");

    /* Fill in interfaces address information */
    /* IP Address */
    sock_int = (struct sockaddr_in *) &ifalias[0].ifra_addr;
    sock_int->sin_family = AF_INET;
    sock_int->sin_addr.s_addr = inet_addr("10.0.0.1");
    /* Broadcast Address */
    sock_int = (struct sockaddr_in *) &ifalias[0].ifra_broadaddr;
    sock_int->sin_family = AF_INET;
    sock_int->sin_addr.s_addr = inet_addr("10.255.255.255");
    /* Subnet Mask */
    sock_int = (struct sockaddr_in *) &ifalias[0].ifra_mask;
    sock_int->sin_family = AF_INET;
    sock_int->sin_addr.s_addr = inet_addr("255.0.0.0");

    /* Insert Interface Entry in InetdbX module */
    if ((errno = putintent(&ifnet,ifalias,1)) == -1) {
        fprintf(stderr,"Error in putintent!\n");
        exit(errno);
    }
```

```
    /*Read Interface Entry in InetdbX module*/
    if ((getint = (struct n_ifnet *)getintent()) == NULL) {
        fprintf(stderr,"Error in getintent!\n");
        exit(errno);
    }
    printf("Obtained Interface entry for device name %s\n",
                getint->if_name);

    /* Delete Interface Entry in InetdbX module */
    if ((errno = delintbyname(getint->if_name)) == -1) {
        fprintf(stderr,"Error in delintbyname!\n");
        exit(errno);
    }
    exit(0);
}
```

Adding, Obtaining, and Deleting a Route Entry

Following is an example of how to add, get, and delete a route entry from the `inetdbx` data module.

These functions must be called before `ipstart` is run.

```
/* <<<<<<<<<<<<<<<<<<<<<<<<<<<<<< uproute.c >>>>>>>>>>>>> */
/*
**      Syntax:    uproute
**      Function:  Manipulate an inetdbX data module route entry
*/



#include <stdio.h>
#include <errno.h>
#include <netdb.h>
#include <net/route.h>

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

void main(int argc, char* argv[], char* envp[])
{
    struct rtreq route,*getroute;   /*Structure for Route Entry*/
```

```
memset(&route,0,sizeof(route));

/* The following example adds a host route entry */
route.req = RTM_ADD;            /*Add Route request*/
route.flags &= ~RTF_HOST;
route.flags |= RTF_UP|RTF_GATEWAY;

/*Fill in destination host route address*/
((struct sockaddr_in *) &route.dst)->sin_addr.s_addr =
             inet_addr("11.0.0.1");
((struct sockaddr_in *) &route.dst)->sin_family = AF_INET;
* (u_char *) &route.dst = sizeof(struct sockaddr_in);

/*Fill in gateway route address*/
((struct sockaddr_in *) &route.gateway)->sin_addr.s_addr =
             inet_addr("10.0.0.1");
((struct sockaddr_in *) &route.gateway)->sin_family = AF_INET;
* (u_char *) &route.gateway = sizeof(struct sockaddr_in);

/*Insert Route Entry in InetdbX module*/
if ((errno = putroutent(&route)) == -1) {
   fprintf(stderr,"Error in putroutent!\n");
   exit(errno);
}

/*Read Route Entry in InetdbX module*/
if ((getroute = getroutent()) == NULL) {
   fprintf(stderr,"Error in getroutent!\n");
   exit(errno);
}
printf("Obtained route entry for host %s\n",
   inet_ntoa(((struct sockaddr_in*) &getroute->dst)->sin_addr));
/*Delete Route Entry in InetdbX module*/
if ((errno = delroutent(&route)) == -1) {
   fprintf(stderr,"Error in delroutent!\n");
   exit(errno);
}
exit(0);
```

Initializing the IP Stack

Starting the IP stack via an application program can be done with the function `ip_start()` provided in `socket.l`.

The System Memory Buffer handler (`sysmbuf`) must be installed before starting the IP stack.

```
/* <<<<<<<<<<<<<<<<<<<<<<< ip_start.c >>>>>>>>>>>>>>>>>>>>>>>> */
/*
**      Syntax:    ip_start
**      Function:  Initialize IP stack
*/

#include <types.h>
#include <const.h>
#include <stdio.h>
#include <errno.h>

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

void main(int argc, char* argv[], char* envp[])
{
   error_code err;
   printf("Initializing IP Stack\n");
   if ((err = ip_start()) !=success) {
      fprintf(stderr,"Error returned from ip_start\n");
      exit(err);
   }
   printf("Initialization complete.\n");
   exit(0);
}
```